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

John D. Hobby

AT&T Bell Laboratories
Murray Hill, NJ 07974-2070

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 notion of smoothness and simplicity. 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 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 and fewer vertices. Filtering out the noise and reducing the vertex count makes the outlines more useful and speeds subsequent processing.

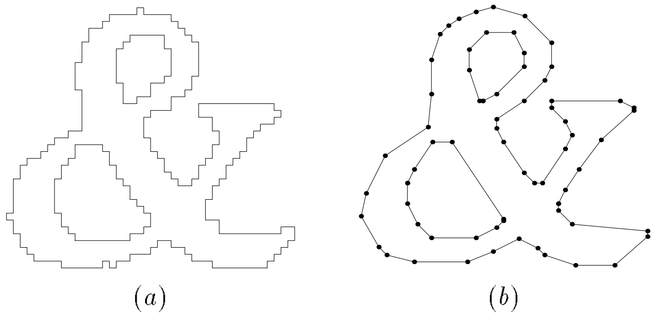
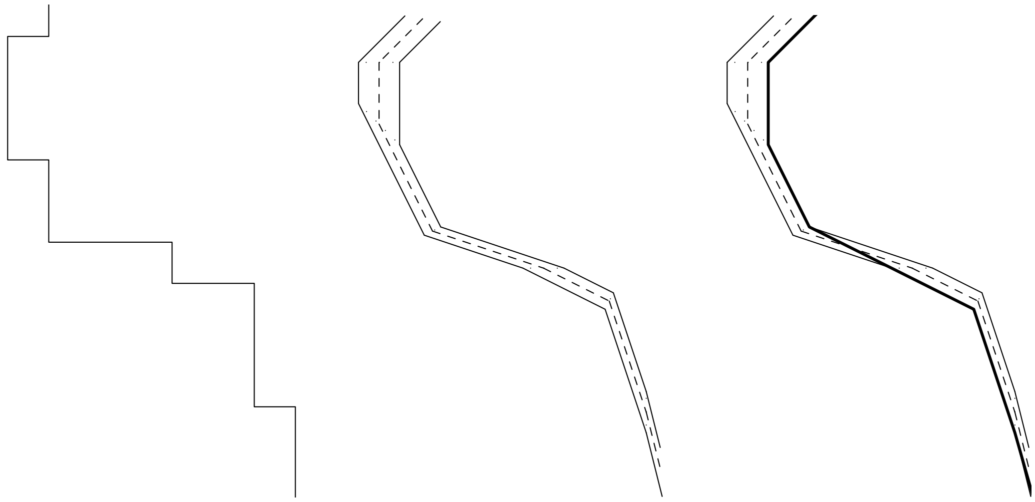


Figure 1: (a) A simulated character shape outline as might be obtained from a digital image; (b) a polygonal approximation with a smoother appearance.

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. The algorithm actually used to generate Figure 1b achieves such smoothness by minimizing the number of inflections in the output [6]; 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. (See [6] for a complete description of this algorithm and discussions of competing polygonal approximation algorithms.)

The only other approach that minimizes inflections is Montanari's idea of minimizing the perimeter subject to the error bounds [7]. (See also Sklansky [12, 13, 14].) 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 algorithm from [6]. For reasons that will become clear later, we refer to that algorithm as the *Stage 1* algorithm.



(a) (b) (c)

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 same output with the minimum perimeter path shown as a heavy line.

Figure 2 illustrates the data structures produced by Stage 1. These consist of a sequence of trapezoids that define a class of minimum-inflection curves that stay within a tolerance of the original input. The polygonal approximation shown in Figure 1b is formed by taking the center line through each trapezoid as indicated by the dashed line in Figure 2. Any curve that passes through the each trapezoid in the correct sequence and has no more inflections than the dashed line has the minimum possible number of inflections. Hence the trapezoid sequence contains all the information needed for a a spline approximation or a more concise polygonal approximation.

This trapezoid sequence facilitates a two-stage process, whereby Stage 1 attempts to find specifications for the "best possible" polygonal approximation, and Stage 2 considers compromises that reduce the vertex count and make it easier to store the output concisely; i.e., the vertices are restricted to a grid whose spacing is a parameter of Stage 2. The result is an algorithm that minimizes the number of inflections subject to the choice of error bound in Stage 1 and among all such approximations, chooses one that obeys the grid constraints imposed in Stage 2 and has nearly the minimum number of vertices allowed by these constraints.

This pipeline approach allows separate control over the approximation error in Stage 1 and the grid spacing in Stage 2. It also allows additional constraints to be imposed via an additional pipeline stage. For instance, it is possible to impose a secondary error tolerance that limits the amount by which the final output can deviate from Stage 1's midline approximation (the dashed line in Figure 2b). This is simply a matter of examining the trapezoids produced by Stage 1 and modifying them so that the parallel sides do not get too far apart.

Another possibility is to modify the input to Stage 1. The raw input can be any source of polygonal outlines where noise can be eliminated by finding approximations that minimize inflections. Outlines can be derived from edge detection in a gray-level image as is done by Rosin and West [10]. Such outlines may be considerably less noisy than Figure 1a, but edge detection is still a local process that provides relatively little control over features like inflections.

We do not consider spline approximations because they are outside the scope of this paper and there is evidence that they would be of marginal utility for the application to scanned document images. The character shapes in a 400 dot-per-inch 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 about five edges.

Section 2 discusses document processing and other applications in more detail. Section 3 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 4 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 5 discusses how to handle the competing demands of maximizing forward and backward visibility at an inflection; and Section 6 presents the main algorithm for Stage 2. Next, Section 7 discusses an optional intermediate pipeline stage that provides better control over the approximation error in Stage 2. Although dynamic programming techniques can be used to minimize the vertex count, the experimental results in Section 8 suggest that it is much better to settle for near minimum since this produces a reasonably fast algorithm. Finally, Section 9 gives a few concluding remarks and two appendices that give proofs that the casual reader may want to skip.

2 Applications

There are numerous applications for the high-quality polygonal approximations generated by the Stage 1 algorithm [6]. These 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 [2].

The primary motivation for this work is the need for compact representations of scanned documents in applications such as electronic libraries. For example, the RightPages project described by O'Gorman [8] involves an electronic library of scanned documents on which OCR is used to allow searching for keywords and other text. Since OCR is not reliable enough to generate complete page descriptions, the scanned page images need to be retained so that the document can be rendered on the user's display screen or printer. Several types of processing in the electronic library application benefit from availability of the outline representation. The simplification and data compression from the Stage 2 algorithm make it attractive to throw away the binary images and use the outline form as the primary representation.

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. Agazzi, Church, and Gale [1] have found that a good way to do the required rotations is to use the Stage 1 algorithm to convert to outlines, then rotate the outlines and scan-convert them to get another binary image.

It would be desirable 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]. The techniques of [1] would still be very useful, but it would surely improve the results to start with the outlines rather than the binary images derived by scan-converting them. The Stage 2 algorithm described here should make this feasible by reducing the space required to store the outlines enough to be competitive with the best available compression techniques for binary images.

Another possible benefit from retaining the outlines is that they could be used as input for the OCR process. Intense research and commercial interest in optical character recognition has lead to a wide variety of competing systems, but 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.

3 The Output of Stage 1

How does the sequence of trapezoids produced by the Stage 1 algorithm determine a class of inflection-minimizing polygonal approximations to the original input, and what properties does the trapezoid sequence have that might assist in constructing polygonal approximations with vertices confined to the output grid? These questions require an examination of the Stage 1 algorithm. In the following discussion, polygonal approximations with vertices confined to the output grid will be called *grid-restricted polygonal approximations.*

3.1 Grid Properties Useful in Stage 2

As explained in [6], the input is a 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

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

Subsequent removal of unnecessary x and y extrema can cause parts of the squares to be trimmed away to give "tolerance rectangles" of the form[1]

$$\{\, (x, y) \mid X_i - \epsilon < x \leq X_j + \epsilon,\ Y_k - \epsilon < y \leq Y_l + \epsilon,\ \}. \tag{2}$$

The points $(X_i \pm \epsilon, Y_j \pm \epsilon)$ at the corners of the tolerance rectangles are interesting because they appear in the output of the Stage 1 algorithm. An important benefit comes from choosing ϵ so that these points belong to a grid that is suitable for Stage 2. For instance, if the initial outlines come from binary images, the X_i, Y_i coordinates are integer numbers of pixel units. If the final output is to have vertices on a grid that is some integer factor K times finer than the pixel grid, it suffices to force ϵ to be a multiple of $1/K$ pixel units.[2]

[1] There is also another trimming step to handle "interfering quadrant boundaries." This trimming can be done to guarantee tolerance rectangles in the form of (2), but it is more natural to allow trimming at arbitrary grid coordinates.

[2] Actually, it is 2ϵ that needs to be a multiple of $1/K$. We use the more stringent condition because it allows better trimming in the context of the preceding note.

Forcing the corners of the tolerance rectangles (2) to be grid points causes the Stage 1 algorithm to begin with a trapezoid sequence

$$R_1R_2L_2L_1,\ R_2R_3L_3L_2,\ R_3R_4L_4L_3, \ldots,$$

where the trapezoid corners L_1, L_2, ... and R_1, R_2, ... are grid points. The ith trapezoid has parallel edges R_iR_{i+1} and $L_{i+1}L_i$, as shown in Figure 3. (Some of the trapezoid edges can be degenerate such as R_5R_6 and L_7L_6 which degenerate to points labeled R_{56} and L_{67} in the figure.)

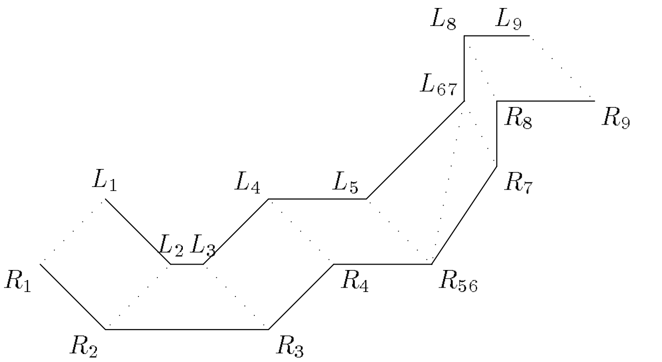


Figure 3: Part of a typical trapezoid sequence as derived from the tolerance rectangles used by the Stage 1 algorithm. Consecutive trapezoids touch at dotted lines and are bounded by parallel edges shown as solid lines.

The Stage 1 algorithm has two operations that change the trapezoid sequence: a "replacement step" that eliminates some trapezoid vertices without introducing any new ones, and a process of edge extension that is illustrated in Figure 4. Edge extension introduces trapezoid vertices that are generally not grid points, but the new vertices are always *convex trapezoid vertices*. A trapezoid vertex V is a convex trapezoid vertex if the trapezoids that meet at V have less than 180° total interior angle at V. The opposite of a convex trapezoid vertex is a *concave trapezoid vertex*. In Figure 4, L_2, L_3, L_6, and L_7 are convex trapezoid vertices and L_{45}, R_2, R_{34}, and R_{567} are concave trapezoid vertices. The following lemma is a consequence of the fact that Stage 1 introduces only convex trapezoid vertices.

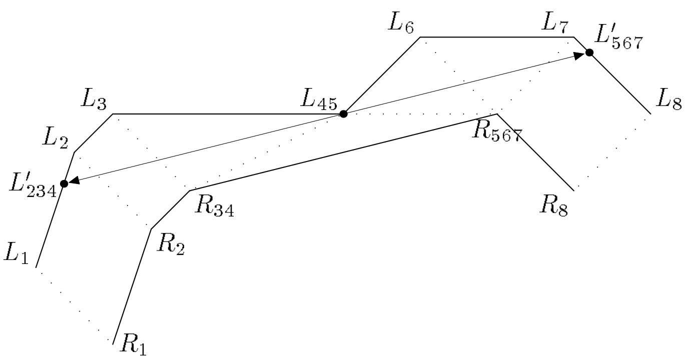


Figure 4: An example of how the Stage 1 edge extension step can alter the trapezoid sequence. New trapezoid vertices L'_{234} and L'_{567} replace L_2, L_3, L_{45}, L_6, and L_7.

Lemma 3.1 *If ϵ is a multiple of the grid spacing and and the input vertices for the Stage 1 algorithm are grid points, then output of Stage 1 has all concave trapezoid vertices at grid points.*

Another consequence of the edge extension step is that all nontrivial trapezoid edges lie on lines that pass through grid points. Segments produced by the edge extension step contain convex trapezoid vertices such as point L_{45} in Figure 4; other segments start and end at grid points. Edge extension can reposition the end points of existing segments but the resulting segments still lie on the same line. This proves the following lemma:

Lemma 3.2 *If ϵ is a multiple of the grid spacing and and the input vertices for the Stage 1 algorithm are grid points, then all nontrivial trapezoid edges lie on lines that pass through grid points.*

3.2 Ensuring there is a Grid-Restricted Path through the Trapezoids

The trapezoid sequence produced by Stage 1 defines a class of inflection-minimizing paths, but [6] does not guarantee that any such paths have vertices restricted to grid points as required by Stage 2. This turns out to be quite simple if we allow a few minor changes in how Stage 1 handles borderline cases. The idea is to make Montanari's minimum perimeter path satisfy the requirements.

Figure 2 illustrates the situation. From Lemma 3.1, the concave trapezoid vertices lie at grid points. The minimum perimeter path through the trapezoid sequence must have vertices only at concave trapezoid vertices since any other type of vertex would allow local changes that reduce the perimeter.

The only problem is that [6] uses strict inequalities in (1) and (2) and treats the trapezoid edges shown as solid lines in Figures 2–4 as out of bounds. Thus we need to modify the Stage 1 algorithm to allow error $\leq \epsilon$ instead of $< \epsilon$.

Tolerance squares and tolerance rectangles must be treated as including their boundaries while trimming to remove unnecessary x and y extrema. This involves changing a few $\leq$ and $\geq$ tests to $<$ and $>$ tests and being careful to cope with quadrant boundaries where tolerance rectangles have height 0 or width 0. The other change is to allow the replacement step to create degenerate trapezoids whose parallel sides are collinear. This is matter of changing one inequality and making sure that the edge extension step illustrated in Figure 4 is implemented carefully enough to cope with the degeneracies. We have the following lemma:

Theorem 3.3 *If the Stage 1 algorithm is modified to allow error $\leq \epsilon$ as explained above and the conditions of Lemma 3.1 are satisfied, then there is a minimum-inflection polygonal path through the trapezoid sequence with all vertices at grid points.*

4 The Best Visible Grid Point Subproblem

The Stage 2 algorithm takes a trapezoid sequence as described in Section 3 and tries to find a grid-restricted minimum vertex polygonal path that passes through the trapezoids in order. Begin by considering the simple case where no inflections are involved and we are given specific starting and ending points P_0 and P_{goal}. Without the grid constraints, we would just start at P_0, find an "optimal" visible point P_1 as shown in Figure 5, and use the same strategy to generate P_2, P_3, etc. until finding a P_i from which P_{goal} is visible. All such optimal points P_i lie on the outer boundary of the trapezoid sequence as does P_1^{opt} in the figure. ("Outer" and "inner" are defined relative to the curvature implied by the sequence of trapezoid directions.)

The region visible from P_0 is delimited by a tangent line through some point Q_t on the inner boundary. 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.

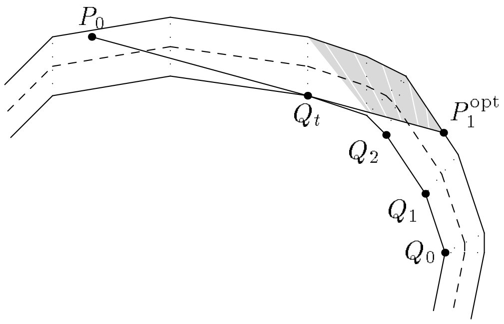


Figure 5: Input trapezoids for Stage 2 and the region to search when finding the best grid point visible from P_0. All points on a single white stripe are equally desirable.

The 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 is the area delimited by the tangent line. Part of this region is shaded in Figure 5 and the rankings are suggested by white stripes.

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 shaded region and stop when the scanning process has covered all points as that are as good as the current best grid point. Section 4.1 shows how suitable affine transformations can make this simple scanning process reasonably efficient.

Since this scanning process is inefficient when the grid is fine, Section 4.2 presents an alternative approach based on continued fractions. Section 4.3 generalizes this to cover a subproblem that will be useful when dealing with inflections.

4.1 Scanning a Simple Polygon for Good Grid Points

One way to find the best grid point visible from a point such as P_0 in Figure 5 is to scan a polygon such as the shaded region in the figure. If the polygon contains grid points, they can be ranked by finding tangent lines. If no grid points are found, the process starts over with a new polygon. The new polygon can be the entire region delimited by the line containing segment P_0Q_t, or it can be a subset delimited by a tangent line through Q_2, Q_t or the vertex between Q_2 and Q_t.

The important part of this process is 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 D_g in which scanning should proceed. (For the shaded region in Figure 5, D_g should be perpendicular to the white stripes near $P_1^{\rm opt}$ and pointed toward the rest of the polygon; i.e., it should be $P_1^{\rm opt} - Q_0$ rotated 90° counterclockwise.) Construct supporting lines for $\mathcal{P}$ perpendicular to D_g and complete a circumscribing parallelogram as shown in Figure 6. 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. [11].)

If the circumscribing parallelogram is $V_{AB}V_{BC}V_{CD}V_{AD}$ as shown in Figure 6, the area of the bounding box is

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

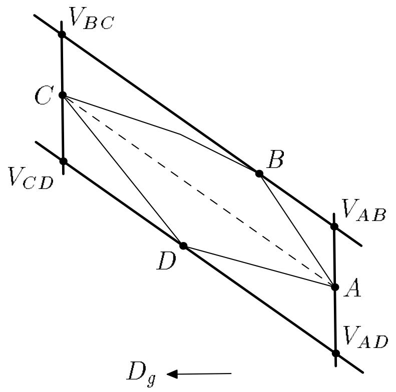


Figure 6: The construction for a parallelogram that circumscribes $\mathcal{P}$, has two edges perpendicular to D_g, and has area no more than twice the area of $\mathcal{P}$.

where $(x_1, y_1) = V_{AB} - V_{AD}$ and $(x_2, y_2) = V_{CD} - V_{AD}$. The following algorithm constructs a 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 (3) as much as possible.

Algorithm 4.1

1. Set $T_{11} = T_{22} = 1$ and $T_{12} = T_{21} = 0$ so T is the 2×2 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 $y_1 = y_1 + kx_1$, $y_2 = y_2 + kx_1$, $T_{21} = T_{21} + lT_{11}$, and $T_{22} = T_{22} + lT_{12}$.
4. Stop if $k = l = 0$; otherwise go to Step 2.

The purpose of Algorithm 4.1 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 constant fraction of its bounding box. This guarantees that an ordinary scan-conversion algorithm can find the grid points in the transformed $\mathcal{P}$ without considering more than $O(\sqrt{\text{area}(\mathcal{P})})$ scan lines. The efficiency of this process is a demonstrated by Lemmas A.1 and A.2 in Appendix A.

We need to show that this scan-conversion can be done in order of increasing D_g component. It turns out not to be all that important to get the ordering exactly right. In practice, we can just modify T to negate x and/or y so as to map D_g to be as compatible as possibly with ordinary left-to-right, top-to-bottom scanning. If the ordering by D_g component has to be exact, the following algorithm does the job. It assumes that the D_g component of (i, j) is $\alpha i + \beta j$ for fixed non-negative parameters α and β.

Algorithm 4.2

1. Use ordinary scan-conversion to generate a list of (i, a_i, b_i) triplets, where the desired integer grid points of the form (i, j) are those where $a_i \le j \le b_i$.
2. Sort the (i, a_i, b_i) triplets by $\alpha i + \beta a_i$, and set pointers p and q to the beginning of the list. Let $d_g()$ be the function that evaluates $\alpha i + \beta a_i$ for a given triple.

3. Advance q to the last triple that has $d_g(q) < 1 + d_g(p)$.

4. Output $(p{\rightarrow}i, p{\rightarrow}a_i)$ and advance $p{\rightarrow}a_i$. Delete triple p if $p{\rightarrow}a_i > p{\rightarrow}b_i$; otherwise insert p after q and advance q. Make sure p points to the head of the list.

5. If the list is not empty, go to Step 3.

Appendix A proves the following theorem:

Theorem 4.1 *Suppose Algorithm 4.1 is used to generate a transformation T for a convex polygon $\mathcal{P}$ and a direction D_g, and then Algorithm 4.2 scans $T(\mathcal{P})$ and its output is mapped by T^{-1}. This outputs the grid points in $\mathcal{P}$ ordered by D_g component and takes time*

$$O\left(\log\left(\frac{A_{B\mathcal{P}}}{A_{\mathcal{P}}}\right) + \sqrt{A_{\mathcal{P}}}\log(A_{\mathcal{P}}) + n + k\right), \tag{4}$$

where $A_{B\mathcal{P}}$ is the area of $\mathcal{P}$'s bounding box, $A_{\mathcal{P}}$ is the area of $\mathcal{P}$, n is the number of sides in $\mathcal{P}$, and k is the number of grid points produced.

4.2 Using Continued Fractions to Find the Best Visible Grid Point

The polygon searching algorithm in Section 4.1 is a useful tool, but it is not ideally suited to the problem of finding the best visible grid point as illustrated in Figure 5. 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, it is not clear how big to make the search polygon, and there is no unique D_g that is always consistent with the notion of ranking grid points according to the tangent line they lie on.

Continued fractions are useful for finding grid points because they make it efficient to scan the rational directions between two given directions. Figure 7a shows an example for $D_0 = (-1, 7)$ and $D_3 = (-11, 3)$. The 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 the rational direction vectors are "as short as possible" in the sense the shaded region in the figure contains no grid points. The main idea is to use this construction where D_0 is a multiple of the optimal tangent line direction $P_1^{\mathrm{opt}} - Q_0$, and the direction from P_1^{opt} to P_0 is D_k for some index k. Directions closest to D_0 are considered "best." The idea is to start at Q_0 in Figure 7b 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 R_1 because R_1' is out of bounds. The next step goes to R_2, and the process terminates there because R_2 is on the correct side of $\ell(P_1^{\mathrm{opt}}P_0)$. The argument that 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 q/p. Expressed in this form, the algorithm generates a sequence of approximations $\bar{D}_0$, $\bar{D}_1$, $\bar{D}_2$, ..., $\bar{D}_k$ 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

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

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

$$\bar{D}_{i-2} + \bar{D}_{i-1},\ \bar{D}_{i-2} + 2\bar{D}_{i-1},\ \bar{D}_{i-2} + 3\bar{D}_{i-1}, \ldots, \bar{D}_{i-2} + (c_i - 1)\bar{D}_{i-1} \tag{7}$$

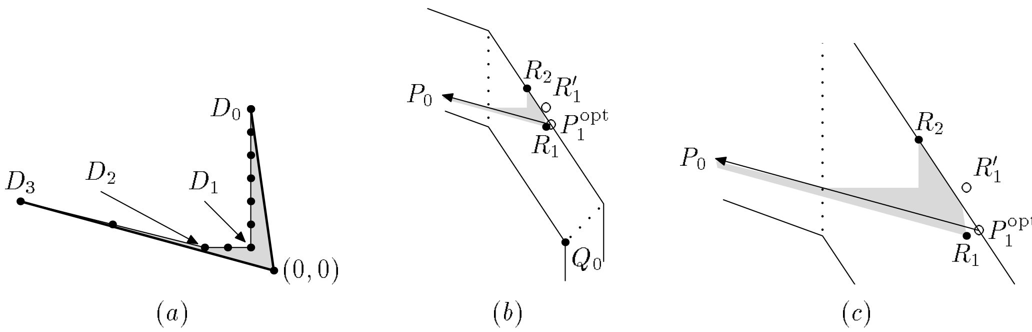


Figure 7: (a) Rational directions between $D_0 = (-1, 7)$ and $D_3 = (-11, 3)$ with a grid-point-free region shaded. Dots mark the ends of the direction vectors. (b) How to use the rational directions to find the best grid point R_2 that is visible from P_0. (Point P_0 should be some distance away in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

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***; i.e., the set of rational directions between $\bar{D}'$ and $\bar{D}''$ is

$$\{\, m\bar{D}' + n\bar{D}'' \mid m > 0,\ n > 0 \,\}.$$

The above properties allow the following algorithm to construct a sequence of rational directions between given initial and final directions $D_{\rm st}$ and $D_{\rm fin}$. Inserting intermediate directions from (7) would produce a sequence of CF-neighbors. The algorithm assumes that $D_{\rm st}$ belongs to the first quadrant; this assumption can be removed via rotations by a multiple of 90°.

Algorithm 4.3

1. Use the continued fraction algorithm to approximate $D_{\rm st}$.
2. If $D_{\rm fin}$ is clockwise from $D_{\rm st}$, output the entries of (5) in reverse order, stopping when some $\bar{D}_i$ is not clockwise from $D_{\rm fin}$. This may require extending (5) by prepending $(0,-1)$ or $(-1,0)$, $(0,-1)$.
3. If $D_{\rm fin}$ is counter-clockwise from $D_{\rm st}$,output the entries of (6) in reverse order, stopping when some $\bar{D}_i$ is not counter-clockwise from $D_{\rm fin}$. 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 a, b such that $D_{\rm fin} = a\bar{D}_j + b\bar{D}_i$. Then use the continued fraction algorithm to approximate (a, b) and output the entries of (5) transformed by the matrix T that maps $(1,0)$ to $\bar{D}_j$ and $(0,1)$ to $\bar{D}_i$.

The running time for Algorithm 4.3 is dominated by the continued fraction expansions of $D_{\rm st}$ and $D_{\rm fin}$. This is known to be

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

where $\|\cdot\|$ denotes any standard vector norm.

The output of Algorithm 4.3 is used as described above to make successive steps from a starting point Q_0 toward an initial goal $P_1^{\rm opt}$, turning as necessary to avoid crossing the outer boundary of the trapezoids. Call this outer boundary the *far path*. The directed line from $P_1^{\rm opt}$ toward P_0 is the *near line*; it defines a limit of visibility from P_0 that the stepping process must reach.

Let D_0, D_1, D_2, $\ldots$, D_n be the output of Algorithm 4.3 with $D_{\text{st}} = P_1^{\text{opt}} - Q_0$ and $D_{\text{fin}} = P_0 - P_1^{\text{opt}}$. The continued fraction algorithm produces additional integer parameters c_0, c_1, c_2, $\ldots$, c_{n-1} that determine the sequence CF-neighbors (7) between any D_i and D_{i+1}. They are

$$D_{i+1} + \frac{j(D_i - D_{i+1})}{c_i}, \quad \text{for} \quad 0 \le j \le c_i. \tag{8}$$

Using these to find the next step from some current grid point P_c requires a *test_pts*() function that takes a uniformly-spaced sequence of integer vectors, and finds the last such vector V for which $P_c + V$ is not outside of the far path. The function should return a failure code if there is no such vector. Here is the algorithm that uses *test_pts*() to find the best grid point visible from P_0.

Algorithm 4.4

1. Use Algorithm 4.3 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 $i = 0$, $P_c = Q_0$, $\ell_{\text{lim}} = \ell(Q_0 Q_1)$, and give P_b a null value.
2. Apply the *test_pts*() function to (8). 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 $\ell_{\text{lim}} = \ell(Q_0 P_b)$.
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

Algorithm 4.4 initializes the ℓ_{lim} line to $\ell(Q_0 Q_1)$ in order to ensure that Q_0 is the point of tangency for the tangent lines that determine which grid point is best. This corresponds to the heavy dashed line in Figure 8. The effect on Algorithm 4.4 is to limit the search region to the dark shaded region in the figure. If no grid points are found there, Algorithm 4.4 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 8. 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 9a 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 9b 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.

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. Algorithms 4.1 and 4.2 can do the job.

Run time bounds for Algorithm 4.4 would not be particularly informative because of unlikely but theoretically expensive operations such as the search in Step 4. Refer to Lemma A.3 in Appendix A for a proof that Algorithm 4.4 finds the correct grid point.

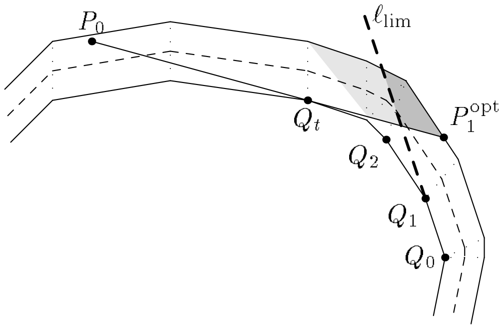


Figure 8: Input trapezoids for Stage 2 and the regions searched by Algorithm 4.4 when it is started at Q_0 (dark region) and Q_1 (lighter region).

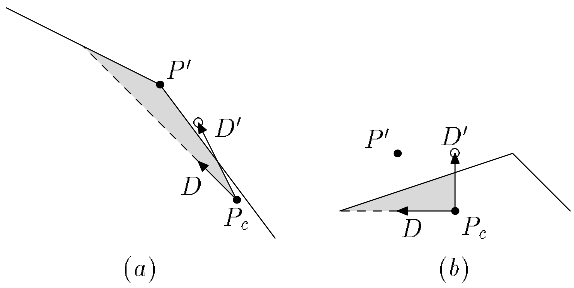


Figure 9: (a) A case where the far path direction crosses the D' ray from P_c with a direction in the D-D' sector, allowing the shaded region to contain a grid point P'. (b) a case where the far path direction avoids the D-D' sector and grid points such as P' need not be considered.

4.3 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 $\ell(Q_{t0}P_1^{\mathrm{opt}})$ that is tangent at Q_{t0} in Figure 10. Since Algorithm 4.4 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 10.

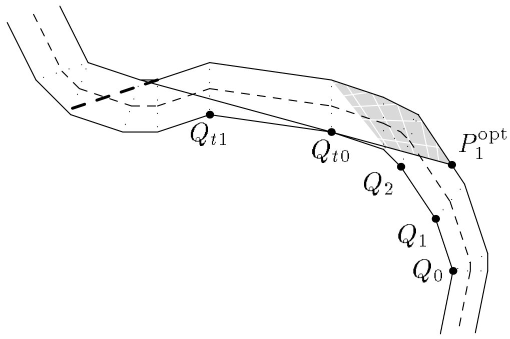


Figure 10: Input trapezoids for Stage 2 and the region to search when finding the trade-off between forward and backward visibility. All points on a single white stripe are equally desirable for the one direction or the other.

This time, two sets of tangent lines are used for ranking the grid points. The white stripes roughly parallel to $\ell(Q_{t0}P_1^{\mathrm{opt}})$ are tangent lines with points of tangency at Q_{t0} or at its predecessor Q_{t1}. Such tangent lines measure visibility back towards Q_{t1}, 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_{t0}. 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 approaches that of $\ell(Q_{t0}P_1^{\mathrm{opt}})$, 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 4.4 more than once. Each subsequent call to Algorithm 4.4 uses a near line based on the previous result as shown in Figure 11. First, Algorithm 4.4 finds a grid point P_1 that has best forward vision in the region delimited by the line $\ell(Q_{t0}P_1^{\mathrm{opt}})$. Next we restrict the region by using $\ell(Q_{t0}P_1)$ as the near line, and use Algorithm 4.4 to find a grid point P_2 that has the best forward vision in the restricted region. We could then find another grid point by using $\ell(Q_{t1}P_2)$ as the near line. In order for Algorithm 4.4 to choose P_1, the dark shaded region in Figure 11 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.

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_1 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}.

Slight modifications to Algorithm 4.4 are needed in order to get it to treat grid points on the near line as unacceptable. This could be done by reversing a few inequalities, but it is more efficient to take advantage of the lack of grid points better that P_{i-1} by initializing $P_c = P_{i-1}$ instead of $P_c = Q_0$ in Step 1 of Algorithm 4.4 and using P_{i-1} in place of P_1^{opt} when initializing D_{st} and D_{fin}. Thus in Figure 11, the search for P_2 starts at P_1 and immediately chooses a P_c increment

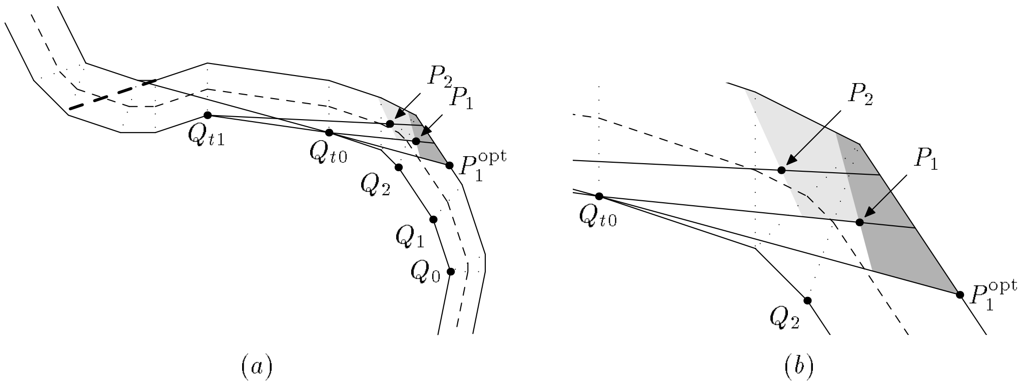


Figure 11: (a) Input trapezoids for Stage 2 and the near lines to use when finding the first two points P_1 and P_2 on trade-off between forward and backward visibility. (b) A close-up of the area near P_1 and P_2. The interiors of the shaded regions are free of grid points.

that keeps P_c above $\ell(Q_{t0}P_1)$. This produces the correct results because it is equivalent to running Algorithm 4.4 with the near line infinitesimally displaced from $\ell(Q_{t0}P_1)$ and the far path modified to run along the boundary of the dark shaded region to P_1; i.e., the modified far path runs along $\ell(Q_0P_1)$ until hitting the original far path, then it continues along the original far path. Generalizing the above argument gives the following theorem:

Theorem 4.2 *Suppose Algorithm 4.4 has found the grid point P_1 that has best forward visibility and lies in the region defined by a tangent line $\ell(Q_{t0}P_1^{\mathrm{opt}})$ as shown in Figure 11. Then running Algorithm 4.4 $j-1$ more times with Step 1 modified as explained above generates a sequence of grid points P_2, P_3, P_4, …, 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}.*

5 Computing the Trade-Off 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 12. 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 $P' \in \mathcal{R}'$.

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 4. 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 12, 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 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 and to provide a way of finding the whole trade-off if desired.

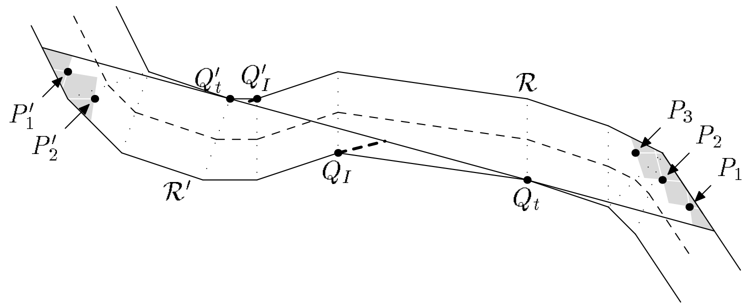


Figure 12: How to use the techniques of Section 4.3 to find a good pair of mutually visible points at an inflection. The $\ell(Q_t'Q_t)$ line is the inner common tangent, and the shaded regions are free of grid points.

5.1 Finding a Pair of Grid Points on the Visibility 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 edges incident on $Q_t'Q_I'$, where Q_I' is the last concave trapezoid vertex among the immediate successors of Q_t'. In the latter case, $P'P$ crosses an edge incident on Q_IQ_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 $Q_t'Q_I'$ obstacle and the Q_IQ_t obstacle.

Figure 12 suggests using the techniques of Section 4.3 to find the best few grid points in $\mathcal{R}$ chosen according to Theorem 4.2, and then find a similar list of points P_1', P_2', ... in $\mathcal{R}'$. In a situation similar to Figure 12, we might proceed as follows.[3]

First, find $P_1 \in \mathcal{R}$ with best forward visibility and $P_1' \in \mathcal{R}'$ with best backward visibility. Suppose $P_1'P_1$ crosses the $Q_t'Q_I'$ obstacle and $P_2'P_1$ crosses the Q_IQ_t obstacle. Note that P_2' is chosen to have the best backward visibility among all grid points whose visibility around the $Q_t'Q_I'$ 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 Q_IQ_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 $Q_t'Q_I'$ obstacle; and other grid points in $\Psi_{\mathcal{R}}(P_k)$ cannot see $P_{k'}'$ because of the Q_IQ_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 Q_IQ_t obstacle; and other grid points in $\Psi_{\mathcal{R}'}(P_{k'}')$ cannot see P_k because of the $Q_t'Q_I'$ obstacle.

[3] Figure 12 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 $Q_t'Q_I'$ and Q_IQ_t obstacles to interfere with mutual visibility as supposed in the following discussion.

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 the following algorithm:

Algorithm 5.1

1. Use Algorithm 4.4 to find the best grid point $P_1 \in \mathcal{R}$. Then set $k = 1$ and $k' = 0$.
2. Use Algorithm 4.4 to find the next best grid points $P'_{k'+1}$, $P'_{k'+2}$, $P'_{k'+3}$ in $\mathcal{R}'$ as explained in Section 4.3, stopping at the first $P'_{l'}$ for which $P'_{l'}P_k$ does not cross the $Q'_tQ'_I$ obstacle.
3. If $k > 1$, use Algorithms 4.1 and 4.2 to find grid points in the portion of $\Psi_{\mathcal{R}'}(P'_{l'}) \setminus \Psi_{\mathcal{R}'}(P'_{k'})$ for which the Q_IQ_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 4.4 to find the next best grid points P_{k+1}, P_{k+2}, P_{k+3} in $\mathcal{R}$ as explained in Section 4.3, stopping at the first P_l for which $P'_{k'}P_l$ does not cross the Q_IQ_t obstacle.
6. If $k' > 1$, use Algorithms 4.1 and 4.2 to find grid points in the portion of $\Psi_{\mathcal{R}}(P_l) \setminus \Psi_{\mathcal{R}}(P_k)$ for which the $Q'_tQ'_I$ 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.

The algorithm works by ensuring that Situation 1 holds at the start of Step 2 and Situation 2 holds at the start of Step 5. Regions $\Psi_{\mathcal{R}'}(P'_{k'})$ and $\Psi_{\mathcal{R}}(P_k)$ expand until reaching a pair of mutually visible grid points. The algorithm assumes that $\Psi_{\mathcal{R}'}(P'_0)$ and $\Psi_{\mathcal{R}}(P_0)$ are defined to be the empty set.

Theorem 5.1 *Algorithm 5.1 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 $\bar{P}$.*

Proof. Step 1 makes Situation 1 vacuously true. Step 2 then ensures that $P'_{l'}$ is the only grid point in $\Psi_{\mathcal{R}'}(P'_{l'})$ that has enough visibility around the $Q'_tQ'_I$ obstacle to see P_k. Step 4 halts if $P'_{l'}P_k$ does not cross the Q_IQ_t obstacle or if Step 3 has found a pair $(\bar{P}', \bar{P})$. If we can show that Step 3 finds such a pair whenever any grid point in $\Psi_{\mathcal{R}'}(P'_{l'})$ can see a grid point in $\Psi_{\mathcal{R}}(P_k)$, it follows that Situation 2 will hold at Step 5 if the algorithm gets there. Similarly, Situation 1 must be restored before returning to Step 2 if we can show that Step 6 finds a pair $(\bar{P}', \bar{P})$ whenever any grid point in $\Psi_{\mathcal{R}}(P_l)$ can see a grid point in $\Psi_{\mathcal{R}'}(P'_{k'})$.

Since Situation 1 holds at Step 2, Step 3 can safely exclude $\Psi_{\mathcal{R}'}(P'_{k'})$ and still be guaranteed of finding the best grid point $\bar{P}' \in \Psi_{\mathcal{R}'}(P'_{l'})$ that can see any point in $\Psi_{\mathcal{R}}(P_k)$. Thus Step 3 finds a pair $(\bar{P}', \bar{P})$ in the cases required above. By making $\bar{P}$ be the best grid point in $\Psi_{\mathcal{R}}(P_k)$ that can see $\bar{P}'$, it guarantees that the result $(\bar{P}', \bar{P})$ in Step 4 satisfies the theorem. A similar argument shows that Step 6 finds a pair $(\bar{P}', \bar{P})$ when required and any result returned in Step 7 also satisfies the theorem.

The algorithm must return some result because repeated failure will eventually cause P_k and $P'_{k'}$ to reach points such as Q'_I and Q_I where the Q_IQ_t and $Q'_tQ'_I$ obstacles cannot interfere. $\square$

Before going on, we need to clarify Steps 3 and 6 of Algorithm 5.1. In Step 3, requiring that the $Q_I Q_t$ 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 $Q_I Q_t$ 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 $Q'_t Q'_I$ and $Q_I Q_t$ 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}, $\ldots$, P_{j-1} for $1 \le i < j \le k$ as shown in Figure 13, or it falls between P_{j-1} and P_j for some $j \le k$ as shown in Figure 14. In the former case, the best grid point visible from $\bar{P}'$ must be P_i because Theorem 4.2 guarantees that P_i is the best grid point in $\mathcal{R}$ whose visibility around the $Q_I Q_t$ obstacle exceeds that of P_{i-1}.

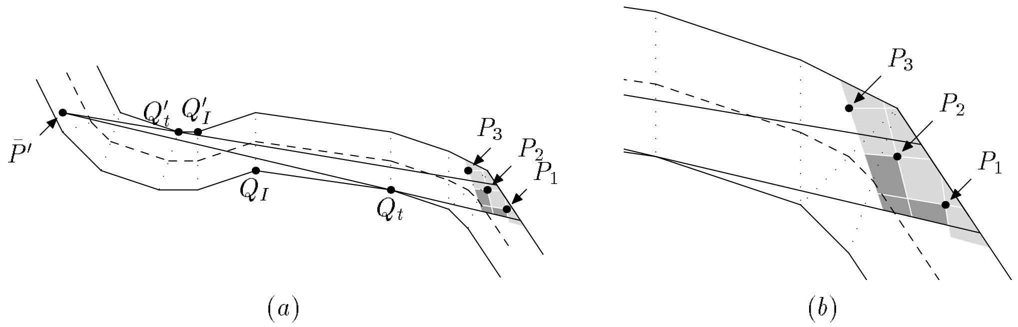


Figure 13: (a) The search area for $\bar{P}$ in Step 3 of Algorithm 5.1 (shaded) with grid-point-free regions shaded lightly; (b) A close-up showing that P_1 is the best grid point in the cone.

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 14, $j = 2$ and the quadrilateral is the dark shaded region. In general, it is the intersection of the visibility cone with

$$\Psi_{\mathcal{R}}(P_k) \setminus \Psi_{\mathcal{R}}(P_j). \tag{9}$$

The following algorithm summarizes the process of finding the best grid point in $\Psi_{\mathcal{R}}(P_k)$ that can see $\bar{P}'$.

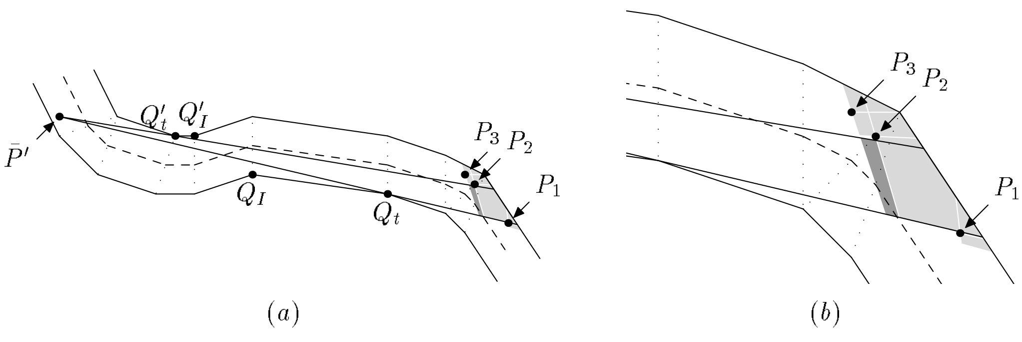


Figure 14: (a) The search area for $\bar{P}$ in Step 3 of Algorithm 5.1 (shaded) with grid-point-free regions shaded lightly; (b) A close-up showing how the visibility cone falls between P_1 and P_2.

Algorithm 5.2

1. Use binary search to locate the visibility cone for $\bar{P}'$ relative to $P_1, P_2, P_3, \ldots, P_k$. Let i be the smallest index for which does not hit $\bar{P}'P_i$ the $Q_I Q_t$ obstacle, and let j be the smallest index for which $\bar{P}'P_j$ hits the $Q'_t Q'_I$ obstacle.
2. If $i < j$ return P_i. Otherwise, use Algorithms 4.1 and 4.2 to search for grid points in the intersection of (9) with the visibility cone and return the best such grid point.

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 5.1.

The contribution of Algorithm 5.2 and its variant to the run time of Algorithm 5.1 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 8 will show that they are small in practice.

The rest of the run time for Algorithm 5.1 is dominated by the calls to Algorithm 4.4 in Steps 1, 2 and 5. This is also difficult to bound theoretically, but the average number of calls to Algorithm 4.4 per invocation of Algorithm 5.1 typically ranges from 3 to 4.8 in practice. (See Section 8 for details.)

5.2 Finding the Rest of the Visibility Trade-Off

Theorem 5.1 guarantees that Algorithm 5.2 finds mutually-visible points $\bar{P}' \in \Psi_{\mathcal{R}'}(P'_{k'})$ and $\bar{P} \in \Psi_{\mathcal{R}}(P_k)$ for which the backward visibility from $\bar{P}'$ and the forward visibility from $\bar{P}$ cannot be simultaneously improved. How can we extend the algorithm to find the complete trade-off between backward visibility from $\bar{P}'$ and forward visibility from $\bar{P}$?

Step 3 of Algorithm 5.1 selects $\bar{P}'$ from a quadrilateral like the dark shaded region in Figure 15a and 15b. It then uses Algorithm 5.2 to decide whether each $\bar{P}'$ point can see any points $\bar{P} \in \Psi_{\mathcal{R}}(P_k)$. A simple way to extend the search would be to try all feasible $\bar{P}'$ points and rank them according to forward visibility from the point $\bar{P}$ returned by Algorithm 5.2. This would mean search the dark shaded region in the figure, as well as the lightly-shaded cone; i.e., searching the region between the common tangent $\ell(Q'_t Q_t)$ and the tangent line from P_{k-1} to the $Q_I Q_t$ obstacle. Theorem 5.1 was based on the argument that $\bar{P}'$ must lie in this region in order to have any chance of seeing a grid point $\bar{P} \in \Psi_{\mathcal{R}}(P_k)$ that is better than P_k.

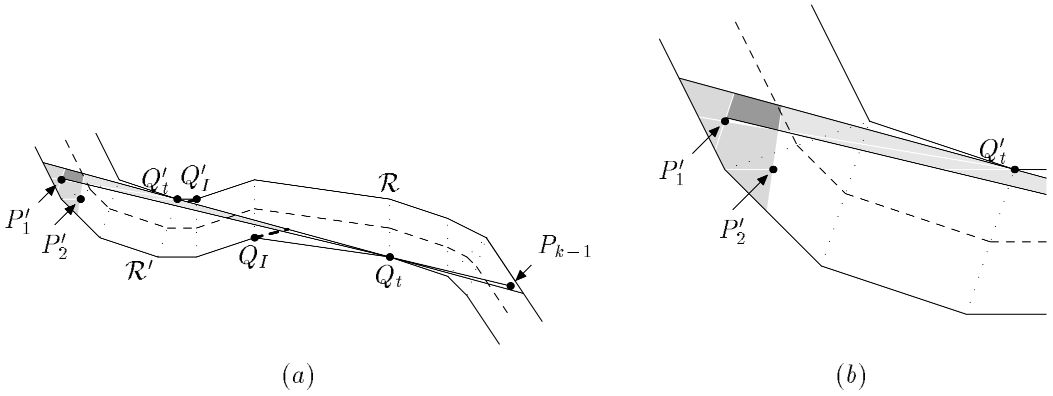


Figure 15: (a) The cone in $\mathcal{R}'$ to search for grid points that can see grid points in $\Psi_{\mathcal{R}}(P_k) \setminus \{P_k\}$. (b) A close-up of the actual search region.

For any given j, using P_{j-1} in place of P_{k-1} modifies the above procedure to search for pairs of grid points $(\bar{P}', \bar{P})$ with $\bar{P}' \in \mathcal{R}'$ and $\bar{P} \in \mathcal{R}$ better than P_j. Similarly, searching a cone based on the tangent from P'_{j-1} to the $Q'_t Q'_I$ obstacle and finding the best visible grid point in $\Psi'_{\mathcal{R}'}(P'_j)$ for each grid point in the cone yields grid points $(\bar{P}', \bar{P})$ with $\bar{P} \in \mathcal{R}$ and $\bar{P}' \in \mathcal{R}'$ better than P'_j. Hence we have a procedure for finding pairs of visible grid points $(\bar{P}', \bar{P})$ where $\bar{P}$ improves upon the result of Algorithm 5.1 at the expense of $\bar{P}'$, and there is a similar procedure that improves $\bar{P}'$ at the expense of $\bar{P}$. This gives the required trade-off.

6 The Main Algorithm for Stage 2

The main algorithm for Stage 2 depends on handling inflections as explained in Section 5 and using the ideas of Section 4 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 or Q_{j+1} as shown in Figure 16.

In Figure 16, 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 $1 \leq i < 4$.

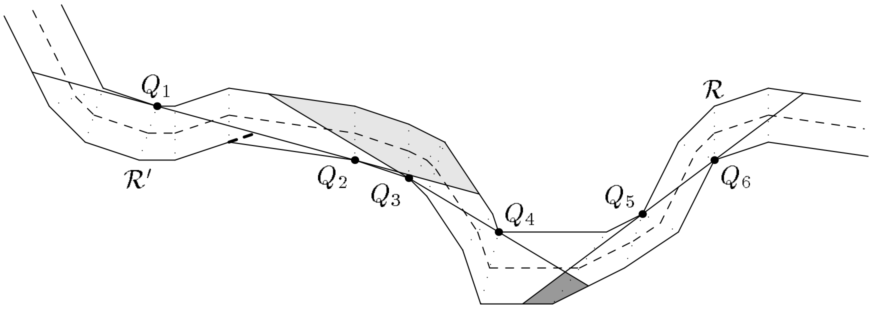


Figure 16: Regions to search when common tangents at successive inflections interfere.

A dynamic programming approach could be used in the case of Figure 16 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 in Section 6.1 performs almost as well in practice. Section 6.2 presents some comments about the dynamic programming approach, and Section 6.3 explains the empirical lower bound on the vertex count that shows dynamic programming is unnecessary in practice.

6.1 The Greedy Algorithm

How should the greedy algorithm handle the situation shown in Figure 16? Start by using Algorithms 4.1 and 4.2 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 4.4 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 $\ell(Q_iQ_{i-1})$ interferes with $\ell(Q_{i+2}Q_{i+3})$. This produces a whole series of regions like the shaded regions in Figure 16. 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, the algorithm might find that no grid points in the appropriate region can see P_{i-1}. This can be handled by simply ignoring the fact that $\ell(Q_iQ_{i+1})$ interferes with $\ell(Q_{i+2}Q_{i+3})$. For instance, this happens for $i = 3$ in the case of Figure 16 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 4.4 to select the best grid point visible from P_2 and then using the techniques of Section 5 to find a pair of mutually-visible grid points that span the Q_5Q_6 inflection.

The above ideas lead to the following algorithm. It uses the term *interference region* to refer to convex regions such as the shaded regions in Figure 16 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 6.1

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 4.4 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 such an l, set $\bar{P} = P_\Lambda$; otherwise set $l = k$, $l' = 1$, and use Algorithms 4.1 and 4.2 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 $i = 1$.
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 Algorithms 4.1 and 4.2 to find a grid point $\bar{P}'$ in the interference region that can see $\bar{P}$ and is as close to $\ell(Q_{2i'-1}, Q_{2i'})$ as possible. 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 4.4 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 4.4 to make $M^+[i]$ the grid point visible behind $M[i']$ with best backward visibility. If $M[i] = M[i'] = P_\Lambda$, use Algorithm 5.1 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 4.4 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, Algorithms 4.1 and 4.2 need a goal direction D_g. This should be chosen to favor points close to where $\ell(Q_{2k-1}Q_{2k})$ and $\ell(Q_1Q_2)$ intersect. Step 5 also uses Algorithms 4.1 and 4.2. 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 Algorithms 4.1 and 4.2.

Note that Algorithm 6.1 invokes Algorithms 4.1, 4.2, and 5.1 at most once per inflection, and it invokes Algorithm 4.4 at most once per output vertex. Sections 4.2 and 5.1 explain that worst case bounds for Algorithms 4.4 and 5.1 would not be very useful, but that they take nearly constant time in practice.

Letting $k = 1$ in Theorem 4.1 does give time bounds for Algorithms 4.1 and 4.2. If it were not for the dependence on the polygon and bounding box areas $A_{\mathcal{P}}$ and $A_{B\mathcal{P}}$, the total time for all invocations of Algorithms 4.1 and 4.2 would be proportional to the number of input trapezoids plus the number of output vertices. The $A_{\mathcal{P}}$ and $A_{B\mathcal{P}}$ terms could theoretically predominate, but square roots and logarithms in (4) make this unlikely in practice. It does not seem worthwhile to give detailed arguments on the size of the $A_{\mathcal{P}}$ and $A_{B\mathcal{P}}$ terms and how to modify the algorithms to limit them if necessary. The results in Section 8 will support the claim that Algorithm 6.1 is nearly linear in practice.

6.2 Dynamic Programming

Since Algorithm 6.1 uses a greedy heuristic that is not guaranteed to minimize the number of output vertices, it is worth considering how it could be modified to give such a guarantee. This involves finding the full trade-off between forward and backward visibility at an inflection as explained in Section 5.2 and using dynamic programming to find a similar trade-off for sequences of inflections where common tangent lines interfere as shown in Figure 16.

Suppose we have a sequence of k inflections where the common tangent line at the ith inflection is $\ell(Q_{2i-1}Q_{2i})$ and the common tangents for successive inflections interfere. The object is to find grid points in $\mathcal{R}'$ with maximal backward visibility and grid points in $\mathcal{R}$ with maximal forward visibility, where $\mathcal{R}'$ and $\mathcal{R}$ are the regions determined by $\ell(Q_1Q_2)$ and $\ell(Q_{2k-1}Q_{2k})$ as indicated in Figure 16.

The dynamic programming algorithm needs to maintain two pieces of information for each grid point in each interference region: which grid point in the previous interference region leads to the best point in $\mathcal{R}'$, and the tangent line direction that is used to rank the point in $\mathcal{R}'$. Maintaining this information requires finding all grid points in each interference region instead of stopping at the first one that meets the visibility requirement. It is also necessary to scan the previous interference region each time a new grid point is found. The net effect is to replace Steps 4–6 of Algorithm 6.1 by a dynamic programming process that requires additional time and additional space. The time and space for Steps 4–6 are multiplied by a factor roughly proportional to the square of the number of grid points in a typical interference region.

The process outlined above produces pairs of grid points (P_i', P_i) where $P_i' \in \mathcal{R}'$, $P_i \in \mathcal{R}$, and there is a k link grid-restricted path from P_i' to P_i for each i. They should be sorted so that as i increases, the backward visibility for P_i' declines and the forward visibility for P_i improves. Since common tangents at consecutive inflections do not necessarily interfere, there are a whole series of such trade-offs where the jth trade-off

$$\mathcal{T}_j = \big((P'_{1,j}, P_{1,j}),\ (P'_{2,j}, P_{2,j}), \ldots,\ (P'_{n_j,j}, P_{n_j,j})\big)$$

has k_j link paths between each pair of grid points $P'_{i,j}$ and $P_{i,j}$. The remaining task is to select some index i_j for each $\mathcal{T}_j$ that minimizes the total number of intermediate vertices needed between all pairs of points $P_{i_j,j}$ and $P'_{i_{j'},j'}$, where $\mathcal{T}_{j'}$ is the trade-off that follows $\mathcal{T}_j$.

Note that invoking Algorithm 4.4 once per intermediate vertex finds the minimum number of intermediate vertices between $P_{i_j,j}$ and $P'_{i_{j'},j'}$. Changing i_j or $i_{j'}$ can change the vertex count by at most 1, and increasing i_j or decreasing $i_{j'}$ cannot increase the vertex count.

This suggests another dynamic programming process where the auxiliary information for each pair in $\mathcal{T}_j$ is the optimal $\mathcal{T}_{j-1}$ index i_{j-1}, the minimum number of intermediate vertices on a path from $\mathcal{T}_1$, and the minimum index i_1 for which the path can start from $P'_{i_1,1}$. The primary goal is to minimize the number of intermediate vertices, and the secondary goal is to minimize i_1. Computing the minimum vertex count and the corresponding i_1 for a pair in $\mathcal{T}_{j+1}$ requires using Algorithm 4.4 to find intermediate vertices along a path back to $P_{n_j,j}$ and then using binary search to find the minimum i for which $P_{i,j}$ can be reached with the same intermediate vertices.

This dynamic programming algorithm replaces Steps 7 and 8 of Algorithm 6.1 by a process that invokes Algorithm 4.4 roughly $\bar{n}\log(\bar{n})$ times as often, where $\bar{n}$ is the average value of n_j for all trade-offs $\mathcal{T}_j$.

A full optimizing version of Algorithm 6.1 would also need a version of Step 2 that tries multiple possibilities for P_1 and selects the one that minimizes the vertex count. This multiplies the running time by the number of starting points. Hence, a full optimizing version of Algorithm 6.1 can be constructed by replacing various steps with dynamic programming versions that are slower various factors. One of these factors is quadratic in the number of grid points in an interference region, and the other two factors are linear or almost linear in the number of grid points in certain regions.

6.3 A Lower Bound on the Vertex Count

Section 6.2 gives a rough idea of how to find grid-restricted polygonal paths that really minimize the vertex count and how to estimate the time penalty for doing so. One way to tell whether this time penalty is worthwhile is to compute a lower bound on the vertex count.

We can find a lower bound on the vertex count by just ignoring the grid constrains. This produces an algorithm similar to that of Goodrich [3]. Guibas, Hershberger, Mitchell and Snoeyink also give algorithms for minimizing the vertex count under a variety of conditions [5]. They show that a dynamic programming approach to finding the exact minimum vertex count takes quadratic time even without grid constraints.

"Ignoring grid constraints" in Algorithm 6.1 amounts to replacing its calls to Algorithms 4.1, 4.2, 4.4, and 5.1 with simpler routines. In the terminology of Section 4.2, the replacement for Algorithm 4.4 only has to intersect the near line with the far path to find the point $P_1^{\rm opt}$. The replacement for Algorithm 5.1 can just find the common tangent line and take the segment of it that lies within or on the boundary of the trapezoids around the inflection. Instead of using Algorithms 4.1 and 4.2 to find grid points in an interference region, the modified Algorithm 6.1 just finds the intersection of the interfering common tangent lines.

The version of Algorithm 6.1 that ignores grid constraints can readily be implemented using floating point arithmetic, but some care is needed to ensure that rounding error does not cause it to over-estimate the vertex count. Since the purpose is to find a lower bound, under-estimates are harmless.

If we want a lower bound theorem instead of a lower bound algorithm, there are two ways to proceed: we could try define parameters that characterize the input well enough to allow for a reasonably tight close-form expression; or we could try to get a bound on the amount by which the vertex count from Algorithm 6.1 exceeds that of the modified version outlined above. The first option seems quite difficult and it would not readily provide an answer to the question of whether or not to use dynamic programming. The second option might work, but it would probably provide a bound that is too weak to be very useful in practice. Hence, we only state it as a conjecture:

Conjecture 6.1 *Modifying Algorithm 6.1 to ignore grid constraints as described above reduces the number of output vertices by at most a factor of two.*

7 Possible 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. This 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: transform each trapezoid $R_iR_{i+1}L_{i+1}L_i$ so that the modified version $R'_iR'_{i+1}L'_{i+1}L'_i$ satisfies

$$\begin{aligned} &d_\infty(\ell(R'_iR'_{i+1}),\, \ell(R_iR_{i+1})) = d_\infty(\ell(L'_iL'_{i+1}),\, \ell(L_iL_{i+1})) \\ &d_\infty(\ell(R'_iR'_{i+1}),\, \ell(L'_iL'_{i+1})) = \min\bigl(2\epsilon_2,\, d_\infty(\ell(R_iR_{i+1}),\, \ell(L_iL_{i+1}))\bigr), \end{aligned} \tag{10}$$

where $\ell(AB)$ is the directed line containing segment AB. Figure 17 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 17 are unaltered 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 do not belong to the modified version of the neighboring trapezoid. 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 $R_2R_3L_3L_2$.

7.1 The Simple Inflection-Free Case

Let us see how to construct a refined trapezoid sequence in a simple case. Suppose we can choose j and k so that the direction sequence

$$R_{i+1} - R_i + L_{i+1} - L_i, \text{ for } i = j-1,\ j,\ j+1, \ldots, k \tag{11}$$

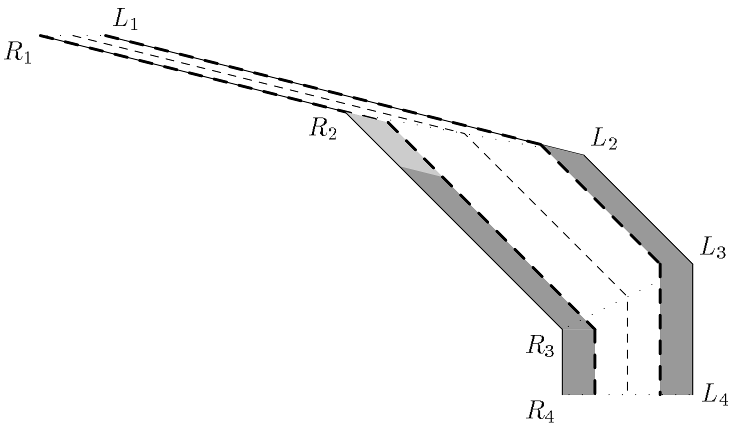


Figure 17: Part of a trapezoid sequence with regions that violate the secondary tolerance ϵ_2 shaded. The heavy dashed lines are at ∞-norm distance ϵ_2 from the midline approximation (light dashed line).

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}, $\ldots$, L_{k+1} and R_{j-1}, R'_j, R_{j+1}, $\ldots$, 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 $\ell(R_iR_{i+1})$.

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

$$\ell(R'_{j-1}R'_j),\ \ell(R'_jR'_{j+1}), \ldots, \ell(R'_kR'_{k+1})$$

and

$$\ell(L'_{j-1}L'_j),\ \ell(L'_jL'_{j+1}), \ldots, \ell(L'_kL'_{k+1})$$

satisfying (10), 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 $\mathcal{L}_{jk}$. This almost works, but we must relax one of these regions to ensure that concave trapezoid vertices lie at grid points. Otherwise, the exclusion of regions like the lightly shaded parallelogram in Figure 17 would make it difficult to guarantee the existence of a suitable grid-restricted polygonal approximation.

If as in Figure 17, the directions (11) 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 (11) 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 $\mathcal{L}'_{jk}$.

Algorithm 7.1 gives a simple iterative routine that constructs the boundary of $\mathcal{L}'_{jk}$ or $\mathcal{R}'_{jk}$ by treating directed lines ℓ_{j-1}, ℓ_j, ℓ_{j-1}, $\ldots$, ℓ_k as the boundaries of half planes to be intersected. With $\ell_i = \ell(R'_i, R'_{i+1})$, it gives the boundary of $\mathcal{R}'_{jk}$, and setting $\ell_i = \ell(L'_i, L'_{i+1})$ gives the boundary of $\mathcal{L}'_{jk}$.

Algorithm 7.1

1. Initialize $i = j$ and set the boundary B to ℓ_{j-1}.
2. If ℓ_i intersects the last segment of B, terminate that segment at the intersection point and append a new segment with ℓ_i's direction. If there is no intersection, remove the last segment and repeat this step.

3. Advance i and terminate if $i > k$. Otherwise go back to Step 2.

We also need an algorithm that computes the convex hull of the grid points in $\mathcal{L}'_{jk}$ or $\mathcal{R}'_{jk}$. This can be done with the help of 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 18 gives an example where grid points are marked by dots and the desired convex hull boundary shown as a heavy line. The part before P on line $\bar{\ell}$ is shown dashed because it is not used by the algorithm below.

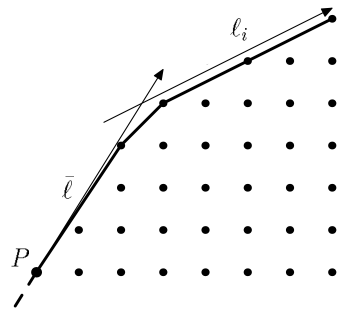


Figure 18: Directed lines $\bar{\ell}$ and ℓ_i and the convex hull of the grid points in the region to the right of both lines. The directed lines are shown as arrows and the convex hull boundary is marked by a heavy line.

Algorithm 4.4 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 $P_1^{\rm opt}$ the point where $\bar{\ell}$ and ℓ_i cross. These inputs guarantee that Step 4 will never need to do a search.

Theorem 7.1 *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.4 in Appendix A. □

We can use the grid hull subroutine to generalize Step 2 of Algorithm 7.1 and obtain an algorithm for the boundary of the convex hull of the grid points to the right of directed lines ℓ_{j-1}, ℓ_j, ℓ_{j+1}, $\ldots$, ℓ_k. This algorithm assumes that each line ℓ_i has a rational direction and that the directions turn right as i increases. The left-turning case can be handled analogously.

Algorithm 7.2

1. Shift ℓ_{j-1} to its right until the resulting line passes through grid points. Then set the boundary B to this line and initialize $i = j$.

2. While ℓ_i does not intersect the last segment of B, remove the last segment and repeat this step.

3. Call the grid hull subroutine with $\bar{\ell}$ equal to the line containing the last segment of B and P equal to the starting point of that segment. (If $\bar{\ell}$ is the only segment in B, P can be any grid point on $\bar{\ell}$ to the right of ℓ_i.) Then remove the portion of B before P and append the result of the grid hull subroutine.

4. Advance i and terminate if $i > k$. Otherwise go back to Step 2.

We can now summarize the process of finding a modified trapezoid sequence for the R_i and L_i that appear in (11) assuming no inflections and directions confined to a 180° sector. If the directions turn right as i increases, use the $\ell(L'_i, L'_{i+1})$ lines from (10) as input for Algorithm 7.1 and use the output to replace L_i for $j-1 \le i \le k+1$. Then use the $\ell(R'_i, R'_{i+1})$ lines as input for Algorithm 7.2 and use the output to replace R_i for $j-1 \le i \le k+1$. Since Algorithm 7.2 can produce segments whose directions do not belong to (11), the two lists of output segments need to be interleaved according to their direction angles and null segments need to be inserted where one list skips over a segment direction from the other list. This produces sequences of matching segments $R''_i R''_{i+1}$ and $L''_i L''_{i+1}$ that define the modified trapezoids. If the directions in (11) turn left as i increases, the procedure is the same except that Algorithm 7.2 is used in place of Algorithm 7.1 and vice versa.

Lemma B.1 in Appendix B shows that the trapezoids produced by this process are a refinement of the input trapezoids and they contain the Stage 1 midline approximation.

7.2 Finishing the Refined Trapezoid Sequence

Before we can extend the techniques of the previous section to handle the entire trapezoid sequence, we have to deal with the restriction that the direction sequence (11) is confined to a 180° sector. This restriction is partly illusionary because algorithms 7.1 and 7.2 only operate on the last few segments in the boundary path being built. The main purpose of the 180° limitation is to allow reasoning about the algorithms using on the basic properties of convex sets rather than a more complicated theory such the Guibas-Ramshaw-Stolfi theory of polygonal tracings [4].

In practice, the way around the 180° limitation is simply to ignore it. Instead of generalizing Lemma B.1 to cover this situation, it is more convenient to subdivide the trapezoid sequence into subsequences where the segment directions span no more than 180°. In the trapezoid sequence of Figure 19a, subdividing at $R_5R_6L_6L_5$ yields subsequences

$$R_1R_2L_2L_1, \ldots, R_5R_6L_6L_5 \quad \text{and} \quad R_5R_6L_6L_5, \ldots, R_9R_{10}L_{10}L_9$$

that obey the 180° limitation. Refining each subsequence as suggested in Section 7.1 gives Figure 19b.

Figure 20 illustrates the process of merging trapezoid subsequences after subdividing at $R_iR_{i+1}L_{i+1}L_i$ and refining separately. As in Figure 19, the segment directions turn right as the index increases. Figure 20a is based on the subsequence that ended at $R_iR_{i+1}L_{i+1}L_i$. As explained in Section 7.1, this trapezoid has been extended to infinity in the direction implied by the arrows. The heavy dashed lines are the segments of the refined trapezoid boundaries that cross the dotted line R_iL_i or lie behind it.

Figure 20b is analogous to Figure 20a, except it shows the initial segments of the result of refining the subsequence starting at $R_iR_{i+1}L_{i+1}L_i$. The idea behind the merging process is that Figure 20a shows what happens before reaching the $R_{i+1} - R_i + L_{i+1} - L_i$ direction, Figure 20b shows what happens after passing that direction, and the intersection of the two is a good way to treat the $R_iR_{i+1}L_{i+1}L_i$ in the merged sequence.

The dashed lines in Figure 20c are computed by finding the shaded regions in Figures 20a and 20b, intersecting the lightly-shaded regions, and finding the convex hull of the grid points in the intersection of the darker regions. The lightly shaded regions in Figures 20a–20c belong to the half plane $\mathcal{S}^l_i$ on or to the left of the midline

$$\ell\left(\frac{L_i + R_i}{2}\ \frac{L_{i+1}, R_{i+1}}{2}\right) \tag{12}$$

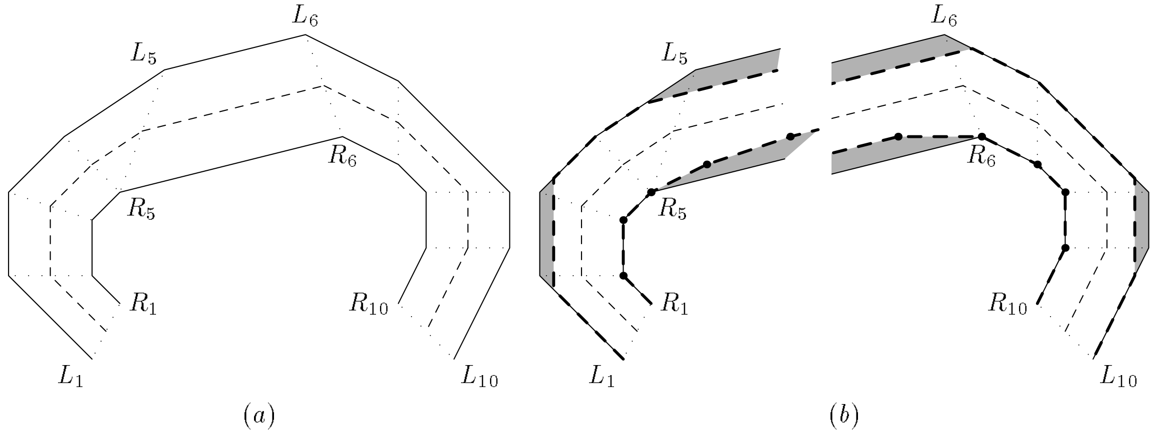


Figure 19: (a) Part of a trapezoid sequence as produced by Stage 1; (b) The result of splitting at $R_5R_6L_6L_5$ and computing separate refinements for each half. Heavy dashed lines show the boundaries of the refined trapezoids and the shaded areas are the regions removed by the refinement process.

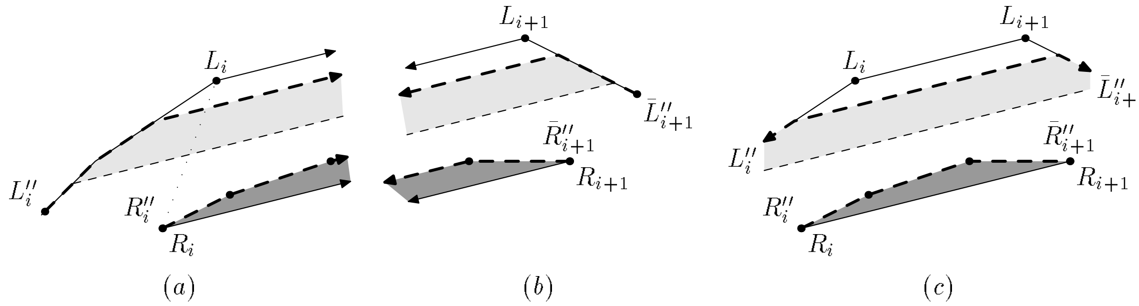


Figure 20: (a) The last trapezoids from separately refining a subsequence ending at $R_iR_{i+1}L_{i+1}L_i$; (b) the first trapezoids resulting from separate refinement starting at $R_iR_{i+1}L_{i+1}L_i$; (c) the result of merging the two refined subsequences.

and the darker regions belong to the half plane $\mathcal{S}_i^r$ on or to the left of the directed line $\ell(R_iR_{i+1})$.

These definitions of $\mathcal{S}_i^l$ and $\mathcal{S}_i^r$ are appropriate when the segment directions turn right as you advance through the trapezoid sequence. If they turn left, we have the *left turn case* where $\mathcal{S}_i^l$ is the region on or to the right of $\ell(L_iL_{i+1})$ and $\mathcal{S}_i^r$ is the region on or to the right of (12). Intuitively, $\mathcal{S}_i^l$ is the region far enough from the "inside edge" to give possible locations for the refined trapezoid boundary $L_i'L_{i+1}'$, and $\mathcal{S}_i^r$ is far enough from the "inside edge" to give possible locations for $R_i'R_{i+1}'$.

The following algorithm assumes that the running Algorithms 7.1 and 7.2 with input based on a sequence of trapezoids ending at $R_iR_{i+1}L_{i+1}L_i$ produced output for which L_i'' is the last left-side vertex not in $\mathcal{S}_i^l$ and R_i'' is the last right-side vertex not in $\mathcal{S}_i^r$. Running the algorithms with input from a trapezoid sequence starting at $R_iR_{i+1}L_{i+1}L_i$ is assumed to have produced $\bar{L}_{i+1}''$ and $\bar{R}_{i+1}''$ as the first left-side vertex not in $\mathcal{S}_i^l$ and the first right-side vertex not in $\mathcal{S}_i^r$.

Algorithm 7.3

1. If this is the left turn case, apply Algorithm 7.2 to the list formed by taking the segments following L_i'' in order followed by the segments preceding L_{i+1}''. Otherwise, apply Algorithm 7.1 to the same list.
2. Use the segments just computed to replace everything after L_i'' and everything before L_{i+1}''.
3. If this is the left turn case, apply Algorithm 7.1 to the list formed by taking the segments following R_i'' in order followed by the segments preceding R_{i+1}''. Otherwise, apply Algorithm 7.2 to the same list.
4. Use the segments just computed to replace everything after R_i'' and everything before R_{i+1}''.
5. Scan the vertices between L_i'' and L_{i+1}'', and construct trapezoids by pairing them with with vertices between R_i'' and R_{i+1}''. Choose the interleaving so that the the segment directions change monotonically.

Algorithm 7.3 emphasizes clarity over efficiency. Steps 1 and 3 begin by running Algorithms 7.1 and 7.2 on the last few segments of their own output. The main effect of this is to restore the segments as originally computed but put them at risk for removal in Step 2 of Algorithm 7.1 or Step 3 of Algorithm 7.2. Another possible improvement is to quit early in Algorithm 7.1 or 7.2 if it becomes clear that the rest of the output will be the same as the input. It may also be possible to optimize Step 5 of Algorithm 7.3 by making use of the pairings in the trapezoid sequences being merged.

We also need a version of Algorithm 7.3 that works when there is an inflection at trapezoid $R_iR_{i+1}L_{i+1}L_i$ as illustrated in Figure 21. This allows any trapezoid sequence to be refined by breaking it into inflection-free subsequences and then merging them together.

The shaded regions in Figures 21a and 21b are analogous to the shaded regions in Figures 20a and 20b. Instead of intersecting them, we merge them 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.

In the following algorithm, B_L^- and B_R^- are the left- and right-side boundaries from running Algorithms 7.1 and 7.2 with input based on a sequence of trapezoids ending at $R_iR_{i+1}L_{i+1}L_i$; similarly, B_L^+ and B_R^+ are the left- and right-side boundaries from running the algorithms with input based on a sequence of trapezoids starting at $R_iR_{i+1}L_{i+1}L_i$. For each vertex V in B_L^-, there is a unique vertex $\rho(V)$ in B_R^- that belongs to the same pair of trapezoids; similarly, each vertex V' in B_R^- has a unique partner $\rho(V')$ in B_L^-, and the same goes for B_L^+ and B_R^+.

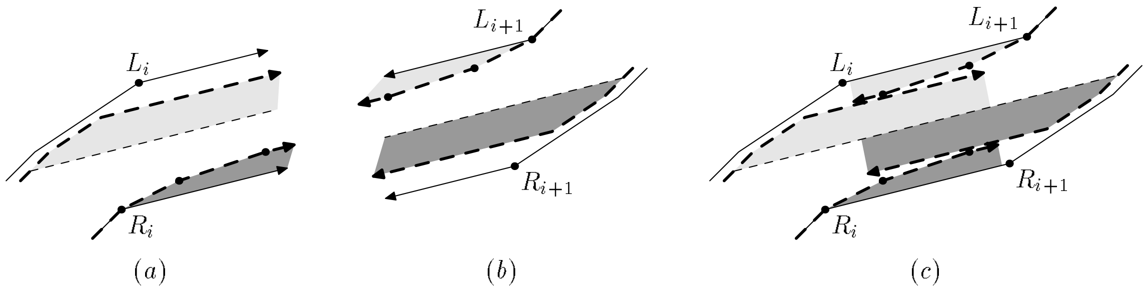


Figure 21: (a) The last trapezoids from separately refining a subsequence ending at $R_iR_{i+1}L_{i+1}L_i$; (b) the first trapezoids resulting from separate refinement starting at $R_iR_{i+1}L_{i+1}L_i$; (c) the two refined subsequences just prior to merging.

Algorithm 7.4

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 after 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 after V_4 in B_R^+ with the segment V_3V_4.
3. If $V_1 \neq V_2$, $V_3 \neq V_4$, and segments V_1V_2 and V_3V_4 are parallel, stop.
4. If $\rho(V_1)$ precedes V_3, let $V_{l1} = V_1$ and $V_{r1} = \rho(V_1)$; otherwise let $V_{r1} = V_3$ and $V_{l1} = \rho(V_3)$. If $\rho(V_2)$ follows V_4, let $V_{l2} = V_2$ and $V_{r2} = \rho(V_2)$; otherwise let $V_{r2} = V_4$ and $V_{l2} = \rho(V_4)$.
5. Assign $\rho(V) = V_{r1}$ for all vertices V strictly between V_{l1} and V_{l2} and $\rho(V) = V_{l2}$ for all V' strictly between V_{r1} and V_{r2}.

An informal restatement of Steps 3–5 of Algorithm 7.4 might be "pair up all the vertices so as to define a trapezoid sequence with a single inflection at V_1V_2 or V_3V_4."

Lemmas B.2 and B.4 in Appendix B show that Algorithms 7.3 and 7.4 preserve the key properties of the refined trapezoid sequences being merged: the region covered by the refined trapezoids is a subset of the similar region for the original trapezoids, and it contains the Stage 1 midline approximation. Appendix B uses these lemmas to prove the following theorem.

Theorem 7.2 *Suppose a trapezoid sequence produced by Stage 1 is broken into inflection-free subsequences whose directions are confined to sectors of $\leq 180°$, and that these are refined separately as explained in Section 7.1 and then merged using Algorithms 7.3 and 7.4. If the region covered by the original and refined trapezoids are R_I and R_O, then the Stage 1 midline approximation is a subset of R_O and $R_O \subseteq R_I$. The same relation holds for any number of contiguous subsequences if the R_I and R_O are modified by having their first and last trapezoids extended to infinity as in Lemma B.1.*

7.3 Choosing the Secondary Tolerance

Sections 7.1 and 7.2 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 (10) limits the ∞-norm deviation to ϵ_2, but this is subsequently relaxed by up to 1 grid unit so as not to destroy the grid point property from Lemma 3.1:

Theorem 7.3 *If a refined trapezoid sequence is computed as in Theorem 7.2, the result will have all concave trapezoid vertices at grid points.*

Proof. Section 7.1 introduces concave trapezoid vertices only via Algorithm 7.2, and that algorithm forces all vertices to be at grid points. Algorithms 7.3 and 7.4 do not create any new concave trapezoid vertices. □

It may be a good idea to try to adjust ϵ_2 and the grid spacing so that the $\ell(R_i' R_{i+1}')$ and $\ell(L_i' L_{i+1}')$ lines from (10) are likely to pass through grid points. 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. Then Lemma 3.2 guarantees that if $R_i \neq R_{i+1}$ and $L_i \neq L_{i+1}$, lines $\ell(R_i R_{i+1})$ and $\ell(L_i L_{i+1})$ will pass through points on the course grid and therefore their bisector will pass through points on the fine grid. Making ϵ_2 a multiple of the fine grid spacing then forces $\ell(R_i' R_{i+1}')$ and $\ell(L_i' L_{i+1}')$ to pass through grid points.

8 Experimental Results

The Stage 1 and Stage 2 algorithms were implemented in C++ and tested on binary images of pages of scanned text from standard database of test documents [9]. Stage 1 converts outlines into trapezoid sequences, and Stage 2 converts them into grid-restricted outlines. Table 1 gives statistics that show how the algorithms perform on a fairly typical sample image.[4] 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 error tolerance and grid spacing. This must be close to the best possible, because number of output vertices ranges from 2.0% to 4.8% more than the lower bound from Section 6.3.

	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
§6.3 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. 6.1	1600	1620	1360	2291	2370	2254	1867	2259
Alg. 4.4 invocations	17194	17167	19477	25830	23951	24063	28742	31478
number from Alg. 5.1	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
outline bytes	29100	33111	35661	47269	43578	44859	51965	57872
CCITT-g4 bytes	47072	47072	47072	47072	47072	47072	47072	47072

Table 1: Statistics for the Stage 2 algorithm on input from a 300 dpi binary image of a page of text from [9]. Columns *A*-*H* give statistics for various settings of the grid spacing, the Stage 1 error tolerance, and the Section 7 secondary tolerance.

[4] The table is based on journal page image a002 from [9]. It is a fairly clean image scanned at 300 dots per inch. It includes text and mathematical formulas, but no drawings or halftone images.

A simple binary file format was developed in order to estimate the space required to store the outlines. Each outline was encoded as a pair of starting coordinates followed by a vertex count and a list of $(\Delta x, \Delta y)$ pairs. All numbers were 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. 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.

Table 1 also gives statistics that are relevant to the design of Algorithms 4.4 and 6.1. Since Algorithm 6.1 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.

Section 4.2 explained that Algorithm 4.4 has to be restarted 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 is never more than 7.1% of the total invocations. The table also shows that the polygon searches in Step 4 of Algorithm 4.4 are also not critical because they are an even smaller percentage of the total invocations. The next row of the table shows that the visibility cone searches used in Steps 3 and 6 of Algorithm 5.1 and defined in Algorithm 5.2 are extremely rare. The practical cost of Algorithm 5.1 is that it accounts for 40% to 57% of the Algorithm 4.4 invocations that are listed in the table. The average number of Algorithm 4.4 invocations per invocation of Algorithm 5.1 ranges from 3.1 for Column H to 4.8 for Column E.

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 reducing the Stage 1 or secondary tolerance or reducing the grid spacing increases the outline byte count, and Figures 22b–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 22a from being eliminated.

Testing the algorithm on all 979 journal page images from [9] produced the timing and data compression statistics in Table 2. 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 optional refinement increased this to a factor of 3. These numbers may be somewhat higher than necessary because no effort has been made to optimize the Stage 2 implementation.

Table 2 shows that the outline byte count generally compares favorably to TIFF bitmaps with CCITT-g4 compression when the Stage 1 tolerance is 1 pixel and the grid spacing and secondary tolerance is $\frac{1}{2}$ pixel. Smaller tolerances would increase the ratio of outline bytes to bytes in the TIFF files as suggested by Table 1. Refer to Figure 23 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 [9] has separate flags for the presence of drawings and halftone images, Figure 23 makes a similar distinction. The algorithm does particularly well on simple line drawings, but drawings containing shaded areas are a difficult case.

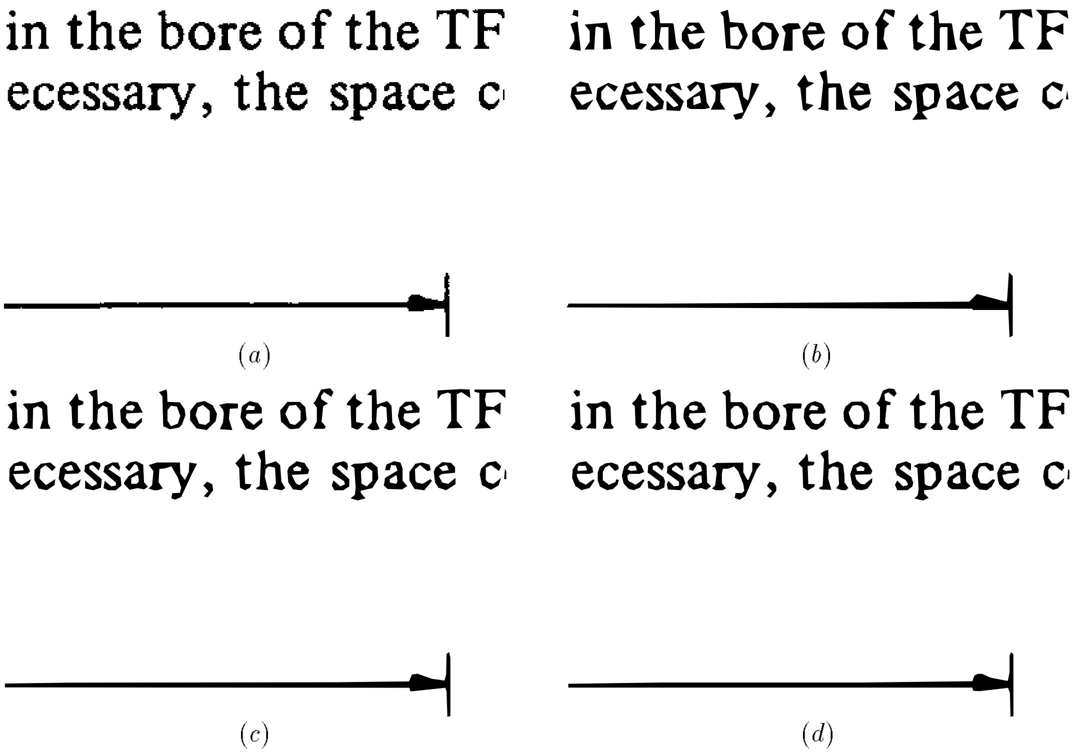


Figure 22: A magnified portion of test document `h047` reproduced by various methods. (a) pixel replication from the raw image; (b) from outlines generated with grid spacing and tolerances as in column C of Table 1; (c–d) the same for columns D and H, respectively.

	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
§6.3 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 2: Statistics for runs of the algorithm on images of journal pages from [9]. 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.

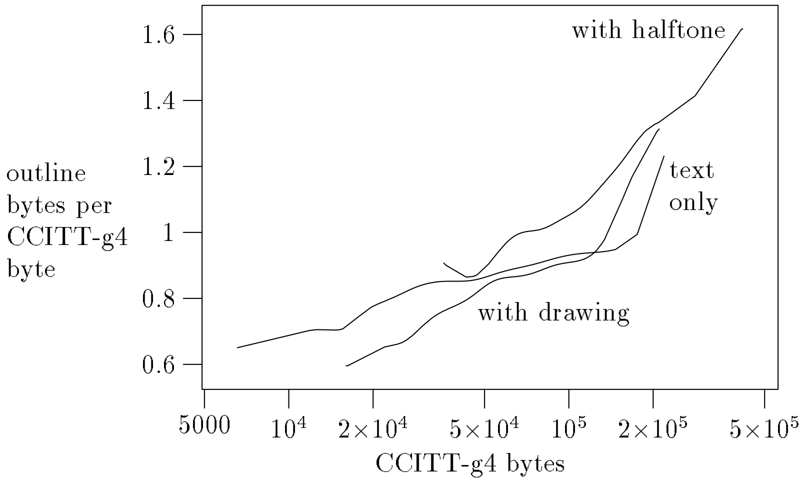


Figure 23: Average ratio of outline bytes to length of the CCITT-g4-compressed TIFF file as a function of the TIFF file size.

9 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 8 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 7. By using the trapezoid sequence data from Stage 1, it avoids complications that Guibas, Hershberger, et. al. [5] encountered in deciding what order the output path hits the input data points.

Appendices

A Validity and Timing Results for Stage 2

The first task is to show that algorithms defined in Section 4.1 allow the grid points in a convex polygon to be scanned efficiently in a specified order.

Lemma A.1 *If $V_{AB}V_{BC}V_{CD}V_{AD}$ is circumscribing parallelogram constructed for a convex polygon $\mathcal{P}$ as explained in Section 4.1, invoking Algorithm 4.1 with $(x_1, y_1) = V_{AB} - V_{AD}$ and $(x_2, y_2) = V_{CD} - V_{AD}$ produces a transformation matrix T that maps $\mathcal{P}$ into a polygon whose area is at least one quarter of its bounding box area.*

Proof. The construction of $V_{AB}V_{BC}V_{CD}V_{AD}$ is based on support points A, B, C, and D as shown in Figure 6. This defines a quadrilateral $ABCD \subseteq \mathcal{P}$ whose area is half of $V_{AB}V_{BC}V_{CD}V_{AD}$. Since affine transformations do not change area ratios, it suffices to show that the transformed parallelogram has at least half the area of its bounding box; i.e.,

$$|x_1y_2 - x_2y_1| \tag{13}$$

must be at least half of

$$(|x_1| + |x_2|)(|y_1| + |y_2|). \tag{14}$$

When Algorithm 4.1 terminates, we must have

$$\min\big(|x_1 + y_1| + |x_2 + y_2|\,,\ |x_1 - y_1| + |x_2 - y_2|\big) \geq \max\big(|x_1| + |x_2|\,,\ |y_1| + |y_2|\big) \tag{15}$$

in order for Steps 2 and 3 to select $k = 0$ and $l = 0$. We will use this to show that (14) is no more than twice (13).

There are some symmetry transforms that do not affect (13), (14), or either side of (15): swapping (x_1, y_1) and (x_2, y_2) or negating either one, or swapping (x_1, x_2) and (y_1, y_2) or negating either one. These allow us to assume without loss of generality

$$x_2 \geq x_1 \geq 0, \quad x_1 + x_2 \geq |y_1| + |y_2|, \quad \text{and} \quad y_1 + y_2 \leq 0. \tag{16}$$

We can also assume that $y_1 + y_2 \neq 0$ since (13) is equal to half of (14) when (16) holds and $y_1 + y_2 = 0$. These assumptions restrict (y_1, y_2) to the dark shaded region in Figure 24a since it would violate (15) to have (y_1, y_2) in the lightly shaded region.

With these assumptions, $|x_1+y_1| + |x_2 + y_2| \geq |x_1| + |x_2|$ from (15) can be written

$$\max\big(\pm(x_1 + y_1) \pm (x_2 + y_2)\big) \geq x_1 + x_2,$$

where the maximum is over the four choices of signs. The only choice that does not conflict with (16) or $y_1 + y_2 \neq 0$ is $-(x_1 + y_1) + (x_2 + y_2) \geq x_1 + x_2$. Combining this with (16) gives

$$\begin{aligned} 2x_1 &\leq y_2 - y_1 \leq x_1 + x_2 \\ -x_1 - x_2 &\leq y_1 + y_2 \leq 0. \end{aligned} \tag{17}$$

which corresponds to the dark area in Figure 24a. Thus

$$x_1y_2 - x_2y_1 = \frac{(x_1 + x_2)(y_2 - y_1) + (x_1 - x_2)(y_1 + y_2)}{2} \geq \frac{2x_1(x_1 + x_2) + 0}{2} \geq 0 \tag{18}$$

so that the absolute value in (13) is unnecessary.

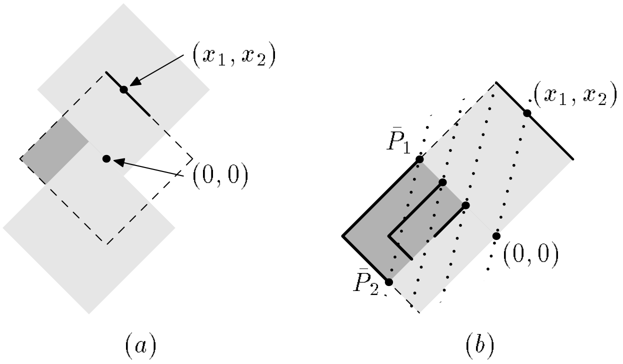


Figure 24: Illustrations for the proof of Lemma A.1. The dark shaded region in (a) gives the possible (y_1, y_2) values. Equation (16) restricts it to the left side of the dashed diamond, and (15) disallows the lightly shaded diamonds. Dotted lines in (b) are (y_1, y_2) values for particular values of α; heavy solid lines in the dark area show where (14) achieves particular values.

Treat (x_1, x_2) as fixed and consider the set S_α of (y_1, y_2) points that obey (15) and (16) and have (13) equal to some value α, where

$$x_1(x_1 + x_2) \le \alpha \le \frac{(x_1 + x_2)^2}{2}. \tag{19}$$

The set S_α is the segment of the line $x_1 y_2 - x_2 y_1 = \alpha$ that satisfies (17). Call it $\mathcal{S}_\alpha$. It has an end point at

$$\left(\frac{-\alpha}{x_1 + x_2}, \frac{\alpha}{x_1 + x_2} \right) \tag{20}$$

as can easily be verified by substituting this for (y_1, y_2) in $x_1 y_2 - x_2 y_1 = \alpha$ and (17).

Figure 24b suggests that the maximum of (14) on $\mathcal{S}_\alpha$ occurs at an end point where $y_1 = -y_2$. If so, (20) must be that end point, and evaluating (14) there gives 2α. Thus (14) is at most twice (13) if α obeys (19). It cannot violate the lower bound in (19) because of (18). Since (16) limits $|y_1| + |y_2|$ to at most $x_1 + x_2$, (14) cannot be more than twice the maximum α allowed by (19). Thus the lemma is true if (20) is the end point of $\mathcal{S}_\alpha$ where (14) is maximized.

The argument that (14) is maximized at (20) is based on the points

$$\bar{P}_1 = \left(-\frac{x_1 + x_2}{2}, \frac{x_1 + x_2}{2} \right) \quad \text{and} \quad \bar{P}_2 = \left(-\frac{3x_1 + x_2}{2}, \frac{x_1 - x_2}{2} \right)$$

that are shown in Figure 24b. Since $x_1 y_2 - x_2 y_1$ has the same value at $(y_1, y_2) = \bar{P}_1$ and $(y_1, y_2) = \bar{P}_2$, it has the same value at $f\bar{P}_1$ and $f\bar{P}_2$, for any multiplier f. Choosing $f = 2\alpha(x_1 + x_2)^{-2}$ makes $f\bar{P}_1$ equal to (20) so that $f\bar{P}_2$ is the point on the line containing $\mathcal{S}_\alpha$ where the value of (14) matches that at (20). Hence it suffices to show that $\mathcal{S}_\alpha$ lies entirely between $f\bar{P}_1$ and $f\bar{P}_2$. This is true because $\mathcal{S}_\alpha$ is a subset of region defined by (17), and this region belongs to the cone defined by $O\bar{P}_1$ and $O\bar{P}_2$ where O is the origin. □

Lemma A.2 *Algorithm 4.1 runs in time proportional to the logarithm of the ratio by which (14) is reduced.*

Proof. Let $X = (x_1, x_2)$ and $Y = (y_1, y_2)$, and let $\|\cdot\|_1$ be the 1-norm so that $\|X\|_1 = |x_1| + |x_1|$ and the function minimized in Step 2 is $\|X + kY\|_1$. This unimodal function of k achieves its minimum at $-x_1/y_1$ or $-x_2/y_2$, or it is minimal for the entire interval $-x_1/y_1 \le k \le -x_2/y_2$. Thus the minimum over the integers can be found in constant time, and a similar argument applies to Step 3.

If Step 2 does not leave $\|X\|_1 < \|Y\|_1$, we have

$$\min(\|X+Y\|_1, \|X-Y\|_1) \ge \|X\|_1 \ge \|Y\|_1$$

so that Step 3 will find that $\|Y+lX\|_1$ is minimized $l=0$. Thus the next iteration of Steps 2 and 3 will find $k=l=0$ and the algorithm will terminate. Similarly, the algorithm terminates after one more iteration if Step 3 leaves $\|Y\|_1 \ge \|X\|_1$.

After Step 2, we always have

$$\min(\|X+Y\|_1, \|X-Y\|_1) \ge \|X\|_1 .$$

Thus if Step 3 finds $l = \pm 1$, the new Y will have $\|Y\|_1 \ge \|X\|_1$ and the algorithm will terminate after one more iteration. A similar argument shows that the algorithm will terminate in one more iteration if Step 2 finds $k=\pm 1$ after Step 3 has executed at least once.

Now consider an iteration of Steps 2 and 3 that is not the first and not the last or second-to-last. If X_0 and Y_0 are the values of X and Y before Step 2 and X_1 and Y_1 are the values after Step 3, we have seen that

$$|k| \ge 2, \quad |l| \ge 2, \quad \text{and} \quad \|Y_1\|_1 < \|X_1\|_1 . \tag{21}$$

Since $Y_0 = Y_1 - lX_1$ and $X_0 = X_1 - kY_0$, we have

$$X_0 = (1+kl)X_1 - kY_1. \tag{22}$$

Thus

$$\|X_0\|_1 \ge |1+kl| \cdot \|X_1\|_1 - |k| \cdot \|Y_1\|_1 \ge (|1+kl| - |k|)\,\|Y_1\|_1 , \tag{23}$$

where (21) implies

$$\begin{aligned} |1+kl| - |k| &= 1 + |k|\,(|l|-1) \ge 3 \quad \text{if } kl > 0; \\ |1+kl| - |k| &= -1 + |k|\,(|l|-1) \ge 2 \quad \text{otherwise} \end{aligned} \tag{24}$$

unless $k = -l = \pm 2$.

When $k=-l=\pm 2$, let $\sigma_k = \pm 1$ be the sign of k so that $k = 2\sigma_k$ and $l = -2\sigma_k$. This makes (22) reduce to $X_0 = -3X_1 - 2\sigma_k Y_1$ so that

$$\begin{aligned} 3(X_1 - \sigma_k Y_0) &= 3(X_1 - \sigma_k(Y_1 - lX_1)) = (3 + 3l\sigma_k)X_1 - 3\sigma_k Y_1 \\ &= -3X_1 - 3\sigma_k Y_1 = X_0 - \sigma_k Y_1 \end{aligned}$$

and $\|X_0\|_1 \ge 3\,\|X_1 - \sigma_k Y_0\|_1 - \|Y_1\|_1$. Since the optimality of k implies $\|X_1 - \sigma_k Y_0\|_1 \ge \|X_1\|_1$, (21) gives

$$\|X_0\|_1 \ge 3\,\|X_1 - \sigma_k Y_0\|_1 - \|Y_1\|_1 \ge 3\,\|X_1\|_1 - \|Y_1\|_1 > 2\,\|Y_1\|_1 . \tag{25}$$

We conclude from (25) or from (23) and (24) that $\|Y_1\|_1 \le \frac{1}{2}\,\|X_0\|_1$. Since Step 2 leaves $\|X_1\|_1 < \|Y_0\|_1$, each iteration of Steps 2 and 3 other than the first or the last two reduces the product of $\|X\|_1$ and $\|Y\|_1$ by at least a factor of two. Since this product is (14), the lemma follows. □

Now we are ready to prove Theorem 4.1. Step 1 of Algorithm 4.2 computes the correct grid points and the rest of the algorithm keeps them ordered by D_g component.

Lemma A.1 guarantees that $T(\mathcal{P})$ has height $\le 2\sqrt{A_\mathcal{P}}$ or width $\le 2\sqrt{A_\mathcal{P}}$. Hence Step 1 generates at most $2\sqrt{A_\mathcal{P}}$ triples and takes time $O(n + \sqrt{A_\mathcal{P}})$. Step 2 takes

$$O\big(\sqrt{A_\mathcal{P}}\log(A_\mathcal{P})\big),$$

and the primitive operations in the other steps are all executed at most once for each output point or once for each triple. Finally, Lemma A.2 gives a time bound for Algorithm A.2 that matches the first term of (4). This gives the time bound required by Theorem 4.1.

Lemma A.3 *Algorithm 4.4 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 4.*

Proof. Note that points on the $\ell_{\lim}$ 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 area in Figures 25a. 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 (8) 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.

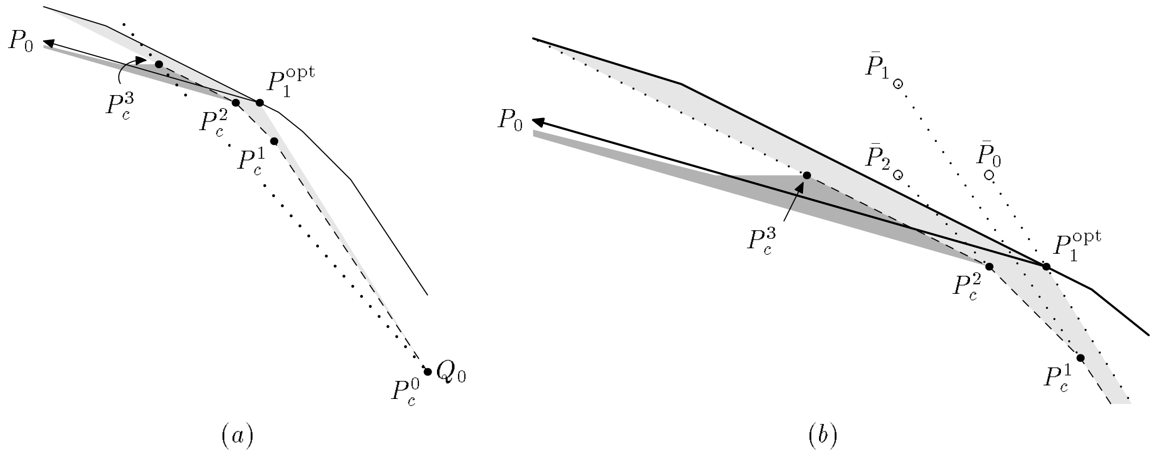


Figure 25: (a) The successive P_c values from Algorithm 4.4 (dashed line), with the grid-point-free region shaded. (b) A close-up of the region near P_c^3.

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 25b.) 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 25a and 25b 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 $Q_0 P_1^{\text{opt}}$, 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 $Q_0 P_c^m$ (the dotted line in Figure 25a), $P_c^{0,m}$ (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 25a and 25b corresponds to Figure 7a. It is a union of triangles of the form

$$P_c^{m-1} \quad P_c^{m-1}+D_j \quad P_c^{m-1}+D_{j+1}. \tag{26}$$

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 (26) 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. □

The following corollary is simply a restatement of some the intermediate results in the above proof. It is needed to prove that Section 7.1 implements the grid hull subroutine correctly.

Corollary A.4 *Let $\bar{P}_c^{0,m}$ be the polygonal line determined by the successive P_c values computed by Algorithm 4.4 as in the proof of Lemma A.3. 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, all interior vertices of $\bar{P}_c^{0,m}$ lie on grid points, and there are no inflection segments on $\bar{P}_c^{0,m}$.*

B The Validity of the Refined Trapezoid Sequence

The purpose of this appendix is to show that the techniques of Section 7 produce a refined trapezoid sequence that includes the Stage 1 midline approximation. In other words, we prove Theorem 7.2.

Lemma B.1 *Suppose trapezoids $R_{j-1}R_jL_jL_{j-1}$ and $R_kR_{k+1}L_{k+1}L_k$ are elongated by withdrawing R_{j-1} and L_{j-1} to infinity along $\ell(R_{j-1}R_j)$ and $\ell(L_{j-1}L_j)$ and withdrawing R_{k+1} and L_{k+1} to infinity along $\ell(R_kR_{k+1})$ and $\ell(L_kL_{k+1})$. Let R_I be the region covered by trapezoids $R_iR_{i+1}L_{i+1}L_i$ for $j-1 \le i \le k$ after the elongation and let R_O be the region similarly covered by the trapezoids from the output of Algorithms 7.1 and 7.2 as outlined at the end of Section 7.1. Then $R_O \subseteq R_I$ and the midline approximation from $(R_{j-1}+L_{j-1})/2$ to $(R_{k+1}+L_{k+1})/2$ is a subset of R_O.*

Proof. Section 7.1 defined semi-infinite regions $\mathcal{L}'_{jk}$ and $\mathcal{R}'_{jk}$ as intersections of half planes, where $\mathcal{L}'_{jk}$ uses half planes bounded by $\ell(L'_iL'_{i+1})$ for $j-1 \le i \le k$, and $\mathcal{R}'_{jk}$ uses half planes bounded by $\ell(R'_iR'_{i+1})$.

Assume without loss of generality that the directions (11) turn right as i increases. (This is the situation in Figure 17.) By construction, R_O is the set difference

$$\mathcal{L}'_{jk} \setminus CV_g(\mathcal{R}'_{jk}), \tag{27}$$

where $CV_g(R)$ denotes the interior of the convex hull of the grid points contained in any region R.

If the relevant portion of the midline approximation is $\mathcal{M}_{jk}$, we can show that $\mathcal{M}_{jk} \in \mathcal{L}'_{jk}$ by exhibiting a set of half planes that contain $\mathcal{M}_{jk}$ and have $\mathcal{L}'_{jk}$ as their intersection. The half planes will be the regions to the right of directed lines $\ell(L'_iL'_{i+1})$ for $j-1 \le i \le k$. Their intersection is $\mathcal{L}'_{jk}$ by definition, and (10) guarantees that segment i of $\mathcal{M}_{jk}$ is parallel to $\ell(L'_iL'_{i+1})$ and to its right. Since the segment directions in $\mathcal{M}_{jk}$ turn monotonically and cover at most 180°, any segment of $\mathcal{M}_{jk}$ is a supporting line. Hence $\mathcal{M}_{jk}$ belongs to the $\ell(L'_iL'_{i+1})$ half plane for each i and thus it belongs to $\mathcal{L}'_{jk}$, the intersection of all such half planes.

Next, we need to show that $\mathcal{M}_{jk}$ and $CV_g(\mathcal{R}'_{jk})$ are disjoint. Any point in $\mathcal{M}_{jk}$ belongs to a segment AB where $A = (R_i + L_i)/2$ and $B = (R_{i+1} + L_{i+1})/2$ for some i. Since $\mathcal{R}'_{jk}$ was constructed to lie to the right of directed lines such as $\ell(R'_i R'_{i+1})$, (10) guarantees that $\mathcal{R}'_{jk}$ lies to the right of AB. Hence, AB is disjoint from $CV_g(\mathcal{R}'_{jk})$ and therefore $\mathcal{M}_{jk}$ and $CV_g(\mathcal{R}'_{jk})$ are disjoint as required. Thus $\mathcal{M}_{jk} \subseteq R_O$ as required by the lemma.

In order to show that $R_O \subseteq R_I$, let

$$R_I = \mathcal{L}_{jk} \setminus \mathcal{R}_{jk}, \tag{28}$$

where $\mathcal{L}_{jk}$ is the region on or to the right of directed lines $\ell(L_i L_{i+1})$, for $j-1 \le i \le k$ and $\mathcal{R}_{jk}$ is the region to the right of $\ell(R_i R_{i+1})$, for $j-1 \le i \le k$. From (10), we immediately have $\mathcal{L}'_{jk} \subseteq \mathcal{L}_{jk}$ and $\mathcal{R}_{jk} \subseteq \mathcal{R}'_{jk}$. Thus

$$CV_g(\mathcal{R}_{jk}) \subseteq CV_g(\mathcal{R}'_{jk}).$$

From Lemma 3.1, we have $\mathcal{R}_{jk} = CV_g(\mathcal{R}_{jk})$ and thus

$$\mathcal{R}_{jk} \subseteq CV_g(\mathcal{R}'_{jk}) \quad \text{and} \quad \mathcal{L}'_{jk} \subseteq \mathcal{L}_{jk}.$$

Using (27) for R_O and combining this with (28) gives $R_O \subseteq R_I$ as required by the lemma. □

The next three lemmas show that using Algorithm 7.3 or Algorithm 7.4 to merge separately refined trapezoid subsequences preserves relationships similar to that in Lemma B.1.

Lemma B.2 *Suppose trapezoid sequences*

$$R_{j-1}R_jL_jL_{j-1}, \ldots, R_iR_{i+1}L_{i+1}L_i \quad \text{and} \quad R_iR_{i+1}L_{i+1}L_i, \ldots, R_kR_{k+1}L_{k+1}L_k \tag{29}$$

are refined as explained in Section 7.1, and (11) has no inflections so that Algorithm 7.3 can be used to merge the results. Let $\mathcal{S}_i = \mathcal{S}_i^r \cup \mathcal{S}_i^l$ and let R_I^0 and R_O^0 be the regions covered by the original and trapezoids and the merged refinements. Then

$$(\mathcal{M}_{jk} \cap \mathcal{S}_i) \subseteq (R_O^0 \cap \mathcal{S}_i) \subseteq (R_I^0 \cap \mathcal{S}_i), \tag{30}$$

where $\mathcal{M}_{jk}$ is the midline approximation corresponding to R_I.

Proof. Recall from Section 7.2 that $\mathcal{S}_i^r \cup \mathcal{S}_i^l$ is a half plane bounded by $\ell(L_iL_{i+1})$ or $\ell(R_iR_{i+1})$ and defined so that all points in $R_I^0 \cap \mathcal{S}_i$ lie on or between these lines.

Assume without loss of generality that the directions (11) turn right as the indices increase, and apply Lemma B.1 to the two subproblems. The results can be written

$$\begin{aligned}\mathcal{R}_{ji} \subseteq CV_g(\mathcal{R}'_{ji}) \subset \mathcal{I}(\mathcal{M}_{ji}) \subseteq \mathcal{L}'_{ji} \subseteq \mathcal{L}_{ji},\\ \mathcal{R}_{ik} \subseteq CV_g(\mathcal{R}'_{ik}) \subset \mathcal{I}(\mathcal{M}_{ik}) \subseteq \mathcal{L}'_{ik} \subseteq \mathcal{L}_{ik},\end{aligned}$$

where $\mathcal{I}(\mathcal{M}_{ji})$ and $\mathcal{I}(\mathcal{M}_{ik})$ are the convex regions bounded by the midline approximations for the subproblems, $\mathcal{L}_{ji} \setminus \mathcal{R}_{ji}$ and $\mathcal{L}_{ik} \setminus \mathcal{R}_{ik}$ are the regions covered by the input trapezoids for the two subproblems, and $\mathcal{L}'_{ji} \setminus CV_g(\mathcal{R}'_{ji})$ and $\mathcal{L}'_{ik} \setminus CV_g(\mathcal{R}'_{ik})$ are the corresponding regions after refinement.

Combining this with

$$\begin{aligned}\mathcal{R}_{ik} \cap \mathcal{S}_i &= \mathcal{R}_{ji} \cap \mathcal{R}_{ik} \cap \mathcal{S}_i\\ CV_g(\mathcal{R}'_{ik}) \cap \mathcal{S}_i &= CV_g(\mathcal{R}'_{ji}) \cap CV_g(\mathcal{R}'_{ik}) \cap \mathcal{S}_i\\ \mathcal{I}(\mathcal{M}_{ik}) \cap \mathcal{S}_i &= \mathcal{I}(\mathcal{M}_{ji}) \cap \mathcal{I}(\mathcal{LM}_{)||}) \cap \mathcal{S}_{)}\\ \mathcal{L}'_{ik} \cap \mathcal{S}_i &= \mathcal{L}'_{ji} \cap \mathcal{L}'_{ik} \cap \mathcal{S}_i\\ \mathcal{L}_{ik} \cap \mathcal{S}_i &= \mathcal{L}_{ji} \cap \mathcal{L}_{ik} \cap \mathcal{S}_i\end{aligned}$$

gives $\mathcal{R}_{jk} \subseteq CV_g(\mathcal{R}'_{jk}) \subset \mathcal{I}(\mathcal{M}_{jk}) \subseteq \mathcal{L}'_{jk} \subseteq \mathcal{L}_{jk}$ which is equivalent to (30). □

The correctness lemma for Algorithm 7.4 uses properties of a function $\Phi(A, B)$ on semi-infinite closed convex sets A and B with oriented boundaries. It requires an inner common tangent $P_A P_B$ of A and B with $P_A \in A$ and $P_B \in B$ chosen so that P_A is as early as possible on the boundary of A and P_B is as late as possible on the boundary of B. The result of $\Phi(A, B)$ is the closed region that contains A and has as its boundary the boundary of A up to P_A, the segment $P_A P_B$, and the part of B's boundary that follows P_B. This is the lightly shaded region in Figure 26 together with the darker region labeled "A." The definition can be extended to handle borderline cases as follows: if $A \cap B$ is a point or a line segment, use that as $P_A P_B$; if $A \cap B$ has interior points the function is undefined.

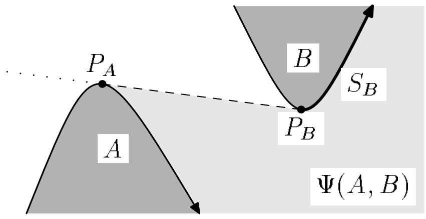


Figure 26: An illustration of $\Phi(A, B)$ for the proof of Lemma B.3.

Lemma B.3 *Let Φ be the function defined above and assume $\Phi(A_1 B_1)$ and $\Phi(A_2 B_2)$ are defined. If $A_1 \subseteq A_2$ and $B_2 \subseteq B_1$ then $\Phi(A_1 B_1) \subseteq \Phi(A_2 B_2)$.*

Proof. Let S_B be the points on the boundary of B from P_B onward and consider the union of all lines tangent to B at points in S_B. This region contains A and has as its boundary the union of S_B and the ray from P_B through P_A. This is the same as $\Phi(A, B)$ except that it contains points outside of A for which the tangent line passes through A before reaching B. (This region is delimited by the dotted line in Figure 26.) Eliminating this region produces an alternative definition for $\Phi(A, B)$: it is the union of all rays that start in A and are tangent to B at some point in S_B. Because $P_A P_B$ is a common tangent, rays from A tangent to B have points of tangency in S_B if and only if the direction of the ray matches B's boundary orientation at the point of tangency. This makes $\Phi(A, B)$ is the union of all rays tangent to B that start in A and agree with B's boundary orientation at the point of tangency.

Enlarging A provides more starting points for the tangent rays, so it can only add points to $\Phi(A, B)$. Since $\Phi(B, A)$ is the complement of the interior of $\Phi(A, B)$, the same argument shows that removing points from B also adds to $\Phi(A, B)$. $\square$

Lemma B.4 *Suppose trapezoid sequences (29) are refined as explained in Section 7.1, and (11) has an inflection so that Algorithm 7.4 can be used to merge the results. If R_I^0 and R_O^0 be the regions covered by the original trapezoids and the merged refinements, then*

$$\mathcal{M}_{jk} \subseteq R_O^0 \subseteq R_I^0, \tag{31}$$

where $\mathcal{M}_{jk}$ is the midline approximation corresponding to R_I.

Proof. Let R_I^- be the region covered by the trapezoid sequence ending at $R_i R_{i+1} L_{i+1} L_i$, and let R_O^- be the union of the corresponding refined trapezoids. Similarly, let R_I^+ and R_O^+ be the regions covered by the original and refined versions of the trapezoids starting at $R_i R_{i+1} L_{i+1} L_i$. Since Section 7.1 expresses the original and refined trapezoids as the difference of convex sets, we have

$$R_I^- = R_{I1} \setminus R_{I2}, \qquad R_O^- = R_{O1} \setminus R_{O2},$$
$$R_I^+ = R_{I3} \setminus R_{I4}, \qquad R_O^+ = R_{O3} \setminus R_{O4},$$

where R_{I1} through R_{I4} and R_{O1} through R_{O4} are convex regions. We can also define a convex regions $\mathcal{I}(\mathcal{M}_{ji})$ and $\mathcal{I}(\mathcal{M}_{ik})$ whose boundaries are respectively $\mathcal{M}_{ji}$ and $\mathcal{M}_{ik}$, the midline approximations for R_I^- and R_I^+.

This allows us to rewrite the results

$$\mathcal{M}_{ji} \subseteq R_O^- \subseteq R_I^- \quad \text{and} \quad \mathcal{M}_{ik} \subseteq R_O^+ \subseteq R_I^+$$

of Lemma B.1 as

$$R_{I2} \subseteq R_{O2} \subseteq \mathcal{I}(\mathcal{M}_{ji}) \subseteq R_{O1} \subseteq R_{I1}$$

and

$$R_{I4} \subseteq R_{O4} \subseteq \mathcal{I}(\mathcal{M}_{ik}) \subseteq R_{O3} \subseteq R_{I3}.$$

Lemma B.3 immediately gives

$$\begin{aligned} \Phi(R_{I2}, R_{I3}) \subseteq \Phi(R_{O2}, R_{O3}) &\subseteq \Phi(\mathcal{I}(\mathcal{M}_{ji}), \mathcal{I}(\mathcal{M}_{ik})) \\ &\subseteq \Phi(R_{O1}, R_{O4}) \subseteq \Phi(R_{I1}, R_{I4}). \end{aligned} \tag{32}$$

Since

$$\begin{aligned} R_O^0 &= \Phi(R_{O1}, R_{O4}) \setminus \Phi(R_{O2}, R_{O3}), \\ R_I^0 &= \Phi(R_{I1}, R_{I4}) \setminus \Phi(R_{I2}, R_{I3}), \end{aligned}$$

and the boundary of $\mathcal{M}_{jk}$ is $\Phi(\mathcal{I}(\mathcal{M}_{ji}), \mathcal{I}(\mathcal{M}_{ik}))$, (32) is equivalent to (31) as required by the lemma. □

Lemmas B.2 and B.4 make it easy to prove Theorem 7.2. The proof is by induction on the total number of calls to Algorithms 7.3 and 7.4 Lemmas B.2 and B.4 each use the containment relations guaranteed by Lemma B.1, but their proofs still work when this is replaced by the induction hypothesis.

Atts.
References
Appendices A and B

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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

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

[10] Paul L. Rosin and Geoff A. W. West. Segmentation of edges into lines and arcs. *Image and Vision Computing*, 7(2):109–114, 1989.

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

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

[13] 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.

[14] 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.

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

John D. Hobby

AT&T Bell Laboratories
Murray Hill, NJ 07974-2070

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 notion of smoothness and simplicity. 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 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.