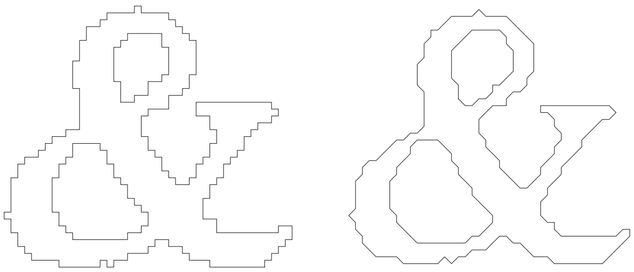
Polygonal Approximations that Minimize the Number of Inflections

John D. Hobby*

Abstract

Consider the problem of processing shape information derived from a noisy source such as a digital scanner. The object is to construct a polygon or a closed curve that matches the input polygon to within a fixed error tolerance and maximizes some intuitive notion of "smoothness and simplicity". Part of this goal should be to minimize the number of inflections.

The algorithm presented here finds an inflection-minimizing polygonal approximation and produces a data structure that characterizes a set of closed curves that fall within the error tolerance and minimize the number of inflections. The algorithm runs in linear time, is reasonably fast in practice, and can be implemented in low-precision integer arithmetic.



(a) (b)

Figure 1: Simulated character shape outlines as might be obtained from a digital image.

1 Introduction

Important practical problems in fields such as robotics, optical character recognition, and font generation often involve extracting shape information from a digital image. The data from the image can be readily converted into polygons, but the resulting polygons tend to have large numbers of short edges and many extraneous inflections. Depending on how the conversion is done, shape boundaries extracted from black-and-white images look like the polygons in Figure 1a or Figure 1b. It is also possible to extract similar information from gray-level images as explained by Rosin and West [27].

The "jaggies" in Figure 1 represent noise that complicates the polygons and tends to interfere with character recognition, shape matching, or whatever the shape information is to be used for. This paper presents an algorithm that gives superior results and is fast enough to be very practical. The algorithm has been implemented and is being tested on real-world applications.

The need for simple approximations that eliminate noise has motivated a lot of work on polygonal and spline approximation beginning with Freeman and Glass [10], Montanari [19], and Ramer [24]. None of these early papers give linear-time algorithms, and Montanari's algorithm takes time $\Omega(n^3)$ on an n-vertex polygon.

Subsequent work has focused on reducing running time, improving quality of approximation, and minimizing complexity of output. Tomek [33, 21] and Reumann and Witkam [25] developed linear-time algorithms that are fast in practice. They sequentially choose a subset of the input vertices based on a bound on the maximum pointwise error for the resulting polygonal approximation. Robergé [26] gives an improvement that reduces the number of vertices chosen.

A slightly better formulation of the greedy approach is the cone intersection method of Williams [35, 36], Sklansky and Gonzalez [30], Leung and Yang [17], and Badi'i and Peikari [4]. All but [4] run in linear time; [36] is unique in that the output vertices are not restricted to be a subset of the input.

Other linear-time algorithms include Wall and Danielsson [34] which measures error by keeping track of the area between the input polygon and the approximation. Algorithms by Davis [6], Hemminger and Pomalaza-Ráez [12], and Ansari and Delp [2] provide little control over the approximation error.

None of the linear-time algorithms achieves an optimal trade-off between the approximation error and the number of output vertices. Existing methods for doing this involve at least $\Omega(n^2)$ time. See Pavlidis [20], Williams [36], Kurozumi and Davis [15], Dunham [8], and Imai and Iri [14]. Some speed can be gained looking for local rather than global optima, but algorithms by Pavlidis [23, 21, 22], Dettori [7], and Ansari and Delp [2] are all significantly worse than linear.

Another important way a polygonal approximation algorithm can be optimal is to minimize the number of inflections subject to a bound on the error. In order to approach the intuitive notion of "the smoothest allow-

*AT&T Bell Laboratories, Murray Hill, NJ 07974

able approximation" it is surely necessary to minimize inflections. The only relevant algorithms are based on Montanari's idea of minimizing the perimeter subject to the error bounds [19]. (See also Sklansky [28, 29, 31].) All of these algorithms have the basic flaw that every output vertex is at the maximum allowable distance from the input. Hence, the minimum perimeter method does not yield approximation algorithms; it just computes the minimum number of inflections. An earlier paper by the present author [13] does claim to minimize inflections, but it only works for perfect, noise-free images.

When spline approximations are needed, they can be generated by extending polygonal approximation algorithms as Albano [1] and Gangnet [11] have done, or by postprocessing a polygonal approximation as Liao [18] and Rosin and West [27] do. The algorithm presented below is well-suited to such post-processing.

We begin in Section 2 with a summary of the applications that have motivated this work. The goal of the algorithm is to minimize inflections subject to a bound on the maximum error, and among all such polygonal approximations, to choose one that lies as close as possible to the input polygon. The general approach will be to maintain a data structure that keeps track of the set of allowable approximations with no more than a certain number of inflections. Section 3 discusses the sets of allowable curves and how to keep track of them. Section 4 presents the algorithm, and Section 5 shows that the algorithm runs in linear time and that the result obeys the error bounds and minimizes the number inflections. Section 6 presents performance data that demonstrate the practicality of the algorithm, and Section 7 gives some concluding remarks.

2 Applications

There are numerous application domains where high quality approximations are worth a little extra processing time.

Optical character recognition (OCR) uses a digital scanner to scan a page of text and then identifies letters and words. This is a problem of substantial research and commercial interest; existing hard-copy libraries would be more useful if available electronically. Processing time is a serious concern in OCR systems, but the major limitation is accuracy, especially when there is no prior knowledge about fonts. An OCR system can introduce so many "typographical errors" that fixing them is almost as expensive as retyping from scratch.

It is essential for an OCR system to extract accurately the character shapes from the scanned image. Contours like Figure 1 need to be approximated by simpler, less noisy polygons that can be processed more efficiently. The algorithm presented here is an ideal way to do this. Henry Baird of AT&T Bell Laboratories plans to incorporate our implementation into his OCR system. He is currently using the Wall and Danielsson algorithm [34]; he plans to continue to use that algorithm for easy cases, and to switch to our more elaborate algorithm when the system requires more accuracy. Figures 10 and 11 in Section 6 give examples of how the polygons produced by our algorithm compare to those from faster algorithms: they appear much smoother and they follow the input polygons more closely.

Another potential application domain is automated fingerprint processing. Sparrow explains [32] that it is very difficult to extract a storable feature set that allows isolated fingerprints to be recognized in the presence of common distortions (such as smudging, blurring, and stretching). He proposes some automatable techniques, but information loss during preprocessing led him to do some of the feature extraction by hand.

A promising approach to feature extraction combines our algorithm with an edge detection algorithm such as that of Lee et. al. [16]. The edge detection algorithm generates polygonal boundaries, which are then smoothed via our algorithm as shown in Figure 2. We have tested this approach on fingerprint images provided by Tony Russo of Bell Labs; the results look good but have yet to be fully evaluated.

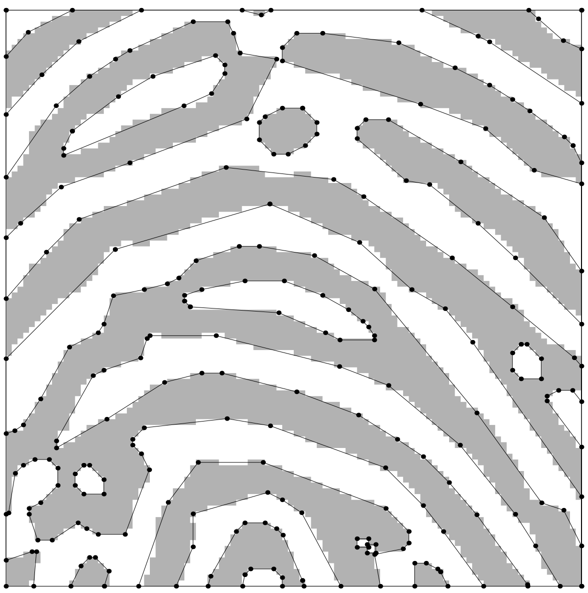


Figure 2: Part of a digitized fingerprint with the polygonal outlines as processed by our algorithm. An edge-detection algorithm produced a binary image and then found polygonal contours like those in Figure 1a.

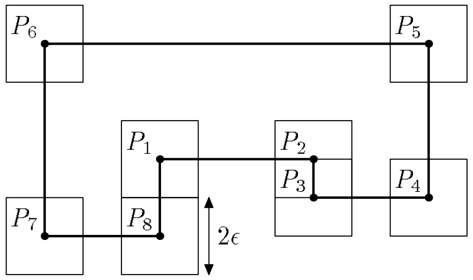


Figure 3: A polygon P with vertices numbered and tolerance rectangles shown with thinner lines.

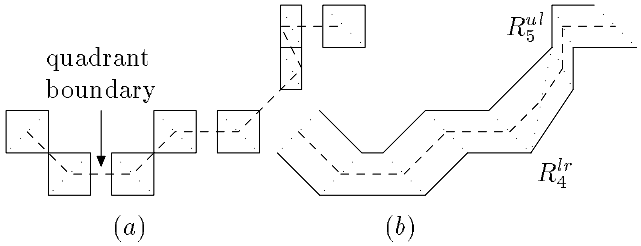


Figure 4: (a) Two quadrant specs with the relevant part of the original input path shown as a dashed line. (b) Corresponding general-purpose data structures.

Much work in computer vision involves extracting outlines from a digitized image and smoothing them while retaining important features. In the shape matching problem, a robot identifies parts coming down an assembly line by scanning images for shapes that match a set of templates. Davis [6] begins the shape matching process by finding a polygonal approximation to the shape outlines that preserves the important features while eliminating noise. Our algorithm could definitely be applied to this problem and is superior to Davis' algorithm in terms of control over the approximation error and the smoothness of the output (as measured by inflection count).

Another computer vision problem is digitizing engineering diagrams to be stored in CAD/CAM systems. (Some existing nuclear power plants are described only in hundreds of thousands of hard-copy drawings.) Smoothing polygons is an important step in reading such diagrams; see Bixler et. al. for more details [5].

3 Data Structures for Allowable Curves

In order for a polygon Q to *match* the input polygon P to within an error bound of ϵ, we require Q to pass P's vertices in order, each within ∞-norm distance ϵ. More precisely, if P has vertices $P_1, P_2, \ldots, P_n$, there must be points $Q(t_1), Q(t_2), \ldots Q(t_n)$ occurring in order on Q such that

$$\|P_i - Q(t_i)\|_\infty < \epsilon \quad \text{for } 1 \le i \le n.$$

Thus the input data may be thought of as a sequence of $2\epsilon \times 2\epsilon$ *tolerance rectangles* through which the approximating polygon must pass.

They are called tolerance rectangles instead of tolerance squares because it is sometimes desirable to restrict the class of allowable polygons Q by trimming off parts of some of the squares. In Figure 3, for instance, trimming off the upper half of the tolerance rectangles for P_1 and P_2 and the lower half of the tolerance rectangles for P_3 and P_4 eliminates a pair of y-extrema.

This suggests an initial data structure consisting of a circular list of tolerance rectangles $R_1, R_2, \ldots, R_n$, where the allowable polygons are those that pass through each R_i in order. Suitable trimming yields *quadrant specs* of the form $R_k, R_{k+1}, \ldots R_l$, where the R_i are monotonic in x and y. That is, if R_i^{ll} and R_i^{ur} are the lower-left and upper-right corners of R_i then for all vectors

$$\{\, \Delta \mid \Delta = R_{i+1}^{ll} - R_i^{ll} \text{ or } \Delta = R_{i+1}^{ur} - R_i^{ur} \text{ for } k \le i < l \,\},$$

no two of them can have x-coordinates of opposite sign and the same goes for the y-coordinates.

Further trimming makes the list of quadrant specs *non-interfering* in the sense that the last tolerance rectangle R_l of one quadrant spec and the first tolerance rectangle $R_{k'}$ of the next cannot have a nontrivial intersection.

The data structures can then be converted to the general form suggested by Figures 4a and 4b. Figure 4a shows the tolerance rectangles in a quadrant spec and Figure 4b illustrates the corresponding general-purpose data structures. The dotted lines crossing diagonally through the tolerance rectangles in Figure 4a are retained in Figure 4b and connected to form a series of parallelograms, triangles, and trapezoids through which any approximating path is required to pass. Call these *tolerance trapezoids.* One path through them is the dashed line joining the midpoints of the dotted lines. When a pair of connecting lines fail to be parallel, the offending quadrilateral is divided into two triangles as indicated by the dotted line between R_4^{lr} and R_5^{ul}.

Because the quadrant specs are non-interfering, consecutive ones are separated by a *quadrant boundary edge* like the bottommost horizontal edge in the figure.

More formally, the general data structures consist of a doubly-linked, cyclic list of *edge structures*, each of which contains a direction vector D, left and right side endpoints L and R, forward and backward links l^+ and l^-, and a few flags that will be discussed later. For an edge e, endpoints $e{\rightarrow}L$ and $e{\rightarrow}R$ are at the ends

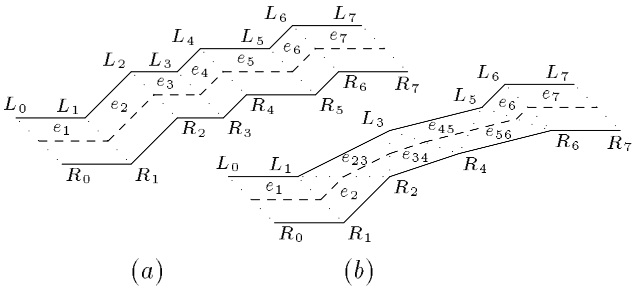


Figure 5: Edge structures before and after the transformation.

of one of the dotted lines in the figure, and $e{\rightarrow}D$ is the direction of the dashed line just before it reaches $(e{\rightarrow}L + e{\rightarrow}R)/2$, and vertices $e{\rightarrow}R$, $e{\rightarrow}L$, $e{\rightarrow}l^{-}{\rightarrow}L$, $e{\rightarrow}l^{-}{\rightarrow}R$ define the tolerance trapezoid.

4 The Algorithm

The algorithm removes inflections via a transformation called *the replacement step*. Section 4.1 describes the replacement step, Section 4.2 presents the main algorithm and Section 4.3 discusses post-processing.

4.1 The Replacement Step Let us start with an example. Figure 5a shows edges e_1, e_2, … e_7 with each $e_i{\rightarrow}L$ labeled L_i and each $e_i{\rightarrow}R$ labeled R_i. The replacement step changes the paths $L_1L_2L_3$, $R_2R_3R_4$, $L_3L_4L_5$, and $R_4R_5R_6$ into single lines; e.g., the edge e_{23} inserted between e_2 and e_3 has $L = L_3$, $D = L_3 - L_1$, $R = R_2$, while $e_2{\rightarrow}L$ becomes L_1. Thus e_{23} is the *replacement edge* that causes $L_1L_2L_3$ to be replaced by L_1L_3.

The inflections at e_2 and e_6 in Figure 5b are *removable inflections.* They could be eliminated by extending the right side of e_{23} backward and the left side of e_{56} forward.

In general, the replacement step involves examining a sequence of edge directions D_j, D_{j+1}, …, D_k and finding the directions D_{i_1}, D_{i_2}, … D_{i_l} where inflections occur. Let a, b, c, and d be consecutive inflection edges and suppose that directions turn leftward between $b{\rightarrow}D$ and $c{\rightarrow}D$ as in Figure 6. Then the right side of the resulting replacement edge is the outer common tangent of

$$a{\rightarrow}R \;\; a{\rightarrow}l^{+}{\rightarrow}R \;\; \cdots \;\; b{\rightarrow}l^{-}{\rightarrow}R$$

and

$$c{\rightarrow}R \;\; c{\rightarrow}l^{+}{\rightarrow}R \;\; \cdots \;\; d{\rightarrow}l^{-}{\rightarrow}R.$$

If this common tangent is $\alpha{\rightarrow}R$ $\beta{\rightarrow}R$, the replacement edge $e_{\alpha\beta}$ has $D = \beta{\rightarrow}R - \alpha{\rightarrow}R$ and this D determines where $e_{\alpha\beta}$ is inserted in the sequence b, $b{\rightarrow}l^{+}$, …, c. If it is inserted after some edge γ, edges α through γ get

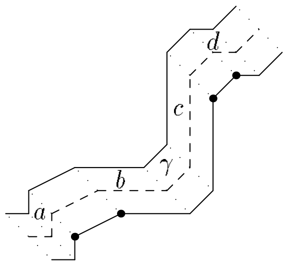


Figure 6: A situation where the replacement step is aborted because $\gamma{\rightarrow}L$ is not left of the common tangent for subpaths $a{\rightarrow}R \ldots b{\rightarrow}l^{-}{\rightarrow}R$ and $c{\rightarrow}R \ldots d{\rightarrow}l^{-}{\rightarrow}R$ whose ends are marked by dots.

their R fields reset to $\alpha{\rightarrow}R$ and edges $e_{\alpha\beta}$ through β get $R = \beta{\rightarrow}R$. Finally, set $e_{\alpha\beta}{\rightarrow}L = \gamma{\rightarrow}L$.

The replacement step consists of performing the above operation for each for each pair of consecutive inflection edges b and c. If directions turn right between $b{\rightarrow}D$ and $c{\rightarrow}D$, it is necessary to swap L and R and swap left and right in the above description. If as in Figure 6 the replacement edge $e_{\alpha\beta}$ would fail to have L left of R relative to the $e_{\alpha\beta}{\rightarrow}D$ direction, then the replacement step is aborted and the edge $\gamma{\rightarrow}l^{+}$ before which $e_{\alpha\beta}$ would have been inserted becomes a *break edge*.

It is not necessary that a and d be inflection edges, rather there must be no inflections between a and b or between c and d. Hence when doing the replacement step for edges e_0, $e_0{\rightarrow}l^{+}$, …, e_n, we can always use e_0 as the first a and e_n as the last d. Also note that the replacement step might cause some edges to become trivial in the sense that their L and R points both coincide with their predecessor's. Such edges should always be removed whenever they occur and consecutive edges with equivalent directions D should always be combined.

Another complication is that consecutive replacement edges must not interfere: the γ for one replacement edge cannot precede the previous one's β and the replacement edge before that one cannot have a β that follows the original edge's α. This is easily achieved by choosing replacement edges one at a time and restricting the range of allowable α and γ values to those that do not interfere.

4.2 The Main Routine The algorithm uses a recursive routine $smooth(e_0, e_n)$ that eliminates unnecessary inflections on $e_0{\rightarrow}l^{+}$, …, $e_n{\rightarrow}l^{-}$ by applying the replacement step.

ALGORITHM 4.1.

1. Take the initial data structures, and trim the tolerance rectangles as necessary to eliminate as many x and y extrema as possible and produce non-interfering quadrant specs.

2. Convert the data structures to the general-purpose form by setting L and R fields to the appropriate vertices of the tolerance rectangles as explained in Section 3.

3. Call $smooth(p, q)$ for each pair of consecutive quadrant boundaries, letting p and q be the quadrant boundary edges.

4. Perform a final clean-up scan over the data structures to remove any removable inflections as will be explained in Section 4.3.

5. For each edge e, output the polygon vertex $(e{\rightarrow}L + e{\rightarrow}R)/2$.

The $smooth()$ routine uses l^f to mean "l^+ or l^- depending on the sign of f" and $LR(e, \sigma)$ to mean "$e{\rightarrow}L$ if $\sigma > 0$ and $e{\rightarrow}R$ if $\sigma < 0$". The replacement step fails for $e_0 \ldots e_n$ if there are less than two inflections between edges e_0 and e_n.

void $smooth(e_0,\ e_n)$
loop
  if (the replacement step fails for $e_0 \ldots e_n$) **return**;
  else Let $\bar{\gamma}$ be the break edge, if any, else $\bar{\gamma} = e_n$;
    Let c_1, c_k be the first, last replacement edges;
    Make $LR(\alpha_1, \sigma_1)$ and $LR(\bar{\beta}_k{\rightarrow}l^-, \sigma_k)$ the ends of their replacement segments;
  $emark(c_1,\ -\sigma_1,\ -1,\ \alpha_1)$;
  $emark(c_k,\ -\sigma_k,\ 1,\ \bar{\beta}_k)$;
  $smooth(e_0,\ \bar{\gamma})$;
  if $(\bar{\gamma} = e_n)$ **return**; **else** $e_0 = \bar{\gamma}{\rightarrow}l^-$;

void $emark(e,\ \sigma, f, e')$
  Delete edges $e{\rightarrow}l^f,\ e{\rightarrow}l^f{\rightarrow}l^f,\ \ldots,\ e'{\rightarrow}l^{-f}$;
  Flag edge e as "needing σ side f extension";

Note that c_1 and c_k could be degenerate in $smooth()$. This happens when a replacement edge has $\alpha{\rightarrow}R = \beta{\rightarrow}R$ so that $D = (0, 0)$. Such edges can normally be omitted from the data structures, but $smooth()$ must include them and give them nonzero directions when they appear as c_1 or c_k. Any direction between $\alpha{\rightarrow}l^+{\rightarrow}D$ and $\beta{\rightarrow}D$ is adequate, but it is better to match the direction of the adjacent replacement edge c_2 or c_{k-1} if possible.

One way to speed up $smooth()$ would be to add additional parameters e'_0 and e'_n that delimit a subrange of $e_0,\ e_0{\rightarrow}l^+, \ldots, e_n$ where inflection edges are possible. Initially $e'_0 = e_0$ and $e'_n = e_n$, but recursive calls could take advantage of the fact that the replacement step does not allow inflections before c_1 or after c_k.

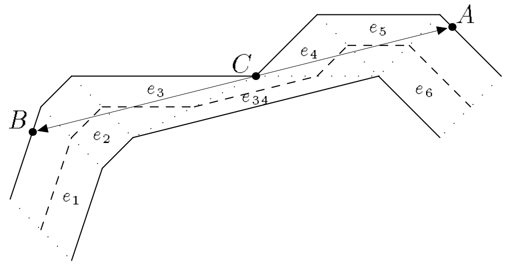


Figure 7: How the left side of an edge e_{34} might need to be extended point C to points A and B.

4.3 Eliminating Removable Inflections What does $smooth()$ do in the case of Figure 5a? It generates connecting the edges labeled e_{23}, e_{34}, e_{45}, and e_{56}, but then calls to $emark()$ delete e_2 and e_6. These gaps in the data structures need to be repaired but $smooth(e_0, e_n)$ cannot do it without clobbering edges outside of the $e_0{\rightarrow}l^+ \ \ldots\ e_n{\rightarrow}l^-$ range.

The marked edges should be thought of places where neighboring tolerance trapezoids are to be trimmed as shown in Figure 7. Extending the left side of e_{34} forward from C to A trims tolerance rectangles for e_4, e_5, e_6; extending back to B does similar trimming for e_3, e_2, e_1.

Implementing Step 4 of Algorithm 4.1 is simply a matter of scanning the edge structures and calling the following recursive routine for each edge e that is marked for σ side f extension. The return value tells where to resume scanning.

edge pointer $eextend(e,\ \sigma,\ f)$
  Let ℓ be the $e{\rightarrow}D$ directed line through $LR(e, \sigma)$;
  for $(e' = e{\rightarrow}l^f,\ e{\rightarrow}l^f{\rightarrow}l^f,\ e{\rightarrow}l^f{\rightarrow}l^f{\rightarrow}l^f,\ \ldots)$
    if (e' is marked for σ side f extension)
      $e' = eextend(e',\ \sigma,\ f)$;
    if ($LR(e', \sigma)$ is on σ side of ℓ) **break**;
  Intersect ℓ with segment $LR(e'{\rightarrow}l^f, \sigma)\ LR(e', \sigma)$;
  for $(\bar{e} = e, \ldots, e'{\rightarrow}l^{-f})$ Let $LR(\bar{e}, \sigma) =$ intersection;
  return $e'{\rightarrow}l^{-f}$;

A few comments about $eextend()$ are in order. The recursive call is needed to avoid losing track of the trimming for edge e' when trimming its tolerance trapezoid. The intersection point need not be stored or explicitly computed until it is actually needed in Step 5 of Algorithm 4.1 or when another invocation of $eextend()$ compares that point to the line ℓ.

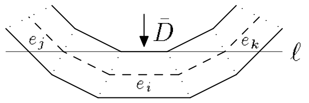


Figure 8: An extremal cut at edge e_i.

5 Theorems

Our first tasks are to show that Algorithm 4.1 minimizes inflections and that the output matches the input polygon. Let the tolerance trapezoid $TR(e)$ for edge e be the open-ended segments $e{\to}L$ $e{\to}R$ and $e{\to}l^-{\to}L$ $e{\to}l^-{\to}R$ and the interior of the trapezoid they define.

Figure 8 illustrates the main tool for recognizing inflections. A line ℓ is an *extremal cut* at edge e_i if there is a direction $\bar{D}$ perpendicular to ℓ and an edge sequence e_j, …, e_k containing e_i where the following hold: 1) directions $e_j{\to}D, \ldots, e_i{\to}D$ have nonnegative $\bar{D}$ components; 2) directions $e_i{\to}l^+{\to}D$, …, $e_j{\to}D$ have nonpositive $\bar{D}$ components; 3) $TR(e_i) \cap TR(e_i{\to}l^+)$ is on the $\bar{D}$ side of ℓ; 4) $TR(e_j) \cap TR(e_j{\to}l^-)$ and $TR(e_k) \cap TR(e_k{\to}l^+)$ are both on the $-\bar{D}$ side of ℓ.

Lemma 5.1. *Every quadrant boundary edge and the predecessor of every break edge discovered by Algorithm 4.1 has extremal cut.*

Proof. If e is a quadrant boundary edge, then e_i is e and the edge sequence e_j, …, e_k is formed by the quadrants before and after e. If $e{\to}l^+$ is a break edge, then e is the γ for a failed replacement edge and the replacement edge construction makes $\alpha{\to}l^+ \ldots \beta$ the edge sequence for an extremal cut.

To see that a subsequent replacement edge cannot destroy an extremal cut, consider the possible locations for $\alpha{\to}L$ and $\beta{\to}L$ relative to ℓ and observe that their difference always has an appropriate $\bar{D}$ component. A similar argument works for the σ side of edge e after *eextend*(e, σ, f).

To get a lower bound on the number of inflections, we need monotonicity conditions in order to prevent tricks such as replacing 90° right turns by 270° left turns. The x and y monotonicity conditions implied by the quadrant specs fulfill this purpose.

Theorem 5.1. *No polygonal path that matches the quadrant specs for the input polygon can have fewer inflections than the result of Algorithm 4.1.*

Proof. An extremal cut at e_i can be classified as leftward or rightward according to the type of turn between $e_i{\to}D$ and $e_i{\to}l^+{\to}D$. A polygonal path Q that passes through the tolerance trapezoids in order and is monotone in ℓ's direction while on the $\bar{D}$ side of ℓ must have a turn that agrees with the classification. If Q has proper monotonicity at the extremal cuts given by Lemma 5.1, it must have as many inflections as the result of the algorithm.

Let P be a polygonal path that passes through the tolerance rectangles in each quadrant spec while obeying x and y monotonicity. Successively transforming P by replacing x, y monotone subpaths with line segments cannot add inflections. Such transformations yield a path that can serve as Q.

Definition 5.1. A polygon Q *matches* edge structures e_1, e_2, …, e_N if the following hold: 1) Q passes sequentially through $Q(t_1)$, …, $Q(t_N)$, where $Q(t_i) \in TR(e_i) \cap TR(e_i{\to}l^+)$; 2) $Q(t)$ passes sequentially through $TR(e_{i_1})$, $TR(e_{i_2})$, …, where $e_{i_{j+1}}$ is $e_{i_j}{\to}l^+$ or $e_{i_j}{\to}l^-$.

Lemma 5.2. *Suppose* $TR(e_i) \cup \cdots \cup TR(e_j) \subseteq TR(e'_{i'}) \cup \cdots \cup TR(e'_{j'})$, *where* $TR(e_k) \cap TR(e_l) \neq \emptyset$ *if and only if* $|k-l| \leq 1$ *and similarly for* $TR(e'_k)$ *and* $TR(e'_l)$. *Then any polygon that matches the edge structures with* $e_i, \ldots, e_j$ *replaced by* $e'_{i'}, \ldots, e'_{j'}$ *also matches the original edge structures.*

Proof. Let $S = TR(e_i) \cup \cdots \cup TR(e_j)$ and $S' = TR(e'_{i'}) \cup \cdots \cup TR(e'_{j'})$. The condition on $TR(e_k) \cap TR(e_l)$ guarantees any polygonal path through S satisfies Condition 2 of Definition 5.1. Hence portions of Q passing through S' obey Condition 2 for the original edge structures and thus all of Q does too.

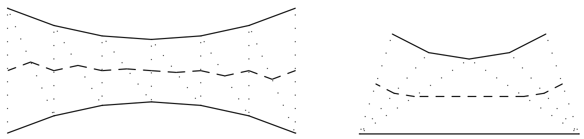
Condition 1 ensures there exist i', j', where $Q(t'_{i'}) \in TR(e'_{i'}) \cap TR(e'_{i'}{\to}l^-)$, $Q(t'_{j'}) \in TR(e'_{j'}) \cap TR(e'_{j'}{\to}l^+)$, and $Q(t)$ passes through S' between $t'_{i'}$ and $t'_{j'}$. This portion of $Q(t)$ must cross each $TR(e_k) \cap TR(e_k{\to}l^+)$ for $i < k < j$ as required by the lemma.

Theorem 5.2. *Any polygonal path that matches the edge structures computed by Algorithm 4.1 also matches the input polygon.*

Proof. We shall prove the invariant that any path matching the current edge structures matches the input polygon. This is true initially because $TR(e) \cap TR(e{\to}l^+)$ is a subset of the tolerance rectangle from which the edge e was derived.

Inserting a replacement edge $e_{\alpha\beta}$ preserves the invariant because of Lemma 5.2. Tolerance rectangles $TR(\alpha{\to}l^+)$, …, $TR(\beta)$ are replaced by a sequence whose union is a subset. Non-consecutive tolerance trapezoids cannot intersect because each $TR(\alpha{\to}l^+)$ through $TR(\gamma)$ are circularly ordered around $\alpha{\to}R$ or $\alpha{\to}L$ and $TR(\gamma{\to}l^+)$ through $TR(\beta)$ are ordered around $\beta{\to}R$ or $\beta{\to}L$.

A similar argument allows Lemma 5.2 to be applied to the actions of *eextend*(). In this case the circular ordering is around the new vertices that are labeled A



(a) (b)

Figure 9: (a) An hourglass; (b) A zero inflection hourglass.

and B in Figure 7. Hence the algorithm preserves the invariant as required.

LEMMA 5.3. *Steps 1, 2, 4, and 5 of Algorithm 4.1 each take time linear in their input size.*

Proof. Step 1 is linear because it involves three simple passes through the input: pass 1 finds non-removable x extrema; pass 2 does the same for y; and pass 3 trims the tolerance rectangles. Step 2 is clearly linear in the input size and Step 5 is clearly linear in the output size.

It only remains to show that Step 4 is linear. This follows because *eextend*()'s return value allows edges to be scanned without backtracking. Hence the time to scan everything is linear.

Before we can show that *smooth*() is linear, we need to deal with situations like Figure 9a where the replacement step makes little or no progress. Part of the problem is that an edge can have a null right side ($e{\rightarrow}R = e{\rightarrow}l^{-}{\rightarrow}R$) or a null left side ($e{\rightarrow}L = e{\rightarrow}l^{-}{\rightarrow}L$). Let a sequence of two or more edges e_1, e_2, $\ldots$, e_k where the directions $e_i{\rightarrow}D$ turn monotonically left or right be a *left turn sequence* or a *right turn sequence*. If a replacement edge is to be useful, its $\alpha{\rightarrow}l^{+} \ldots \beta$ range must contain a left turn sequence with more than one non-null right side or a right turn sequence with more than one non-null left side. Such left or right turn sequences are called *vulnerable*. If as in Figure 9a, a sequence of edges contains no vulnerable left or right turn sequence then it is called an *hourglass*. We have the following lemma.

LEMMA 5.4. *Inserting a replacement edge $e_{\alpha\beta}$ alters the sequence of non-null left sides or the sequence of non-null right sides if and only if the* span *$\alpha{\rightarrow}l^{+}$, $\alpha{\rightarrow}l^{+}{\rightarrow}l^{+}$, $\ldots$, β is not contained in an hourglass.*

This suggests that *smooth*() should be modified to keep track of hourglasses and avoid considering pairs of inflections that belong to the same hourglass when looking for replacement edges. The hourglasses need to be updated before the recursive call to *smooth*(), but this can be done quickly because any part of an hourglass is an hourglass and the definition allows hourglasses to be recognized in linear time.

It may seem that since the span of any replacement edge contains two inflection edges, *smooth*() only needs to keep track of hourglasses that have at least two inflections. In fact, zero inflection hourglasses (0-hourglasses) such as the one in Figure 9b are also dangerous because they could contain many edges and yet have very few that are affected by the replacement step.

What should *smooth*() do with a 0-hourglass? Assume without loss of generality that it is a left turn sequence and thus has at most one edge with a non-null right side. The trick is to avoid computing immediately where in the sequence the non-null right side belongs. For right turning 0-hourglasses, a non-null left side is kept out of sequence. Either way, this out of sequence side is called the *free side*. When the position of the free side is needed at the end of Step 3 of Algorithm 4.1, it can readily be derived from the edge directions.

LEMMA 5.5. *Performing the replacement step on a sequence of edges and 0-hourglasses takes $O(H+E+\bar{H})$ time, where H is the number of 0-hourglasses, E is the number of edges outside of 0-hourglasses, and $\bar{H}$ is the number of edges removed from 0-hourglasses.*

Proof. Since 0-hourglasses are devoid of inflections, the inflections can be found in $O(H+E)$ time. After the replacement step, there are $O(H+E)$ edges outside of known 0-hourglasses, hence it takes $O(H+E)$ time to find maximal-size 0-hourglasses in the output.

Once the inflections are found, how much time is spent for each 0-hourglass in the span of a replacement edge? Assume without loss of generality that the 0-hourglass G is a left turn sequence so that its edges have non-null left sides and its free side is on the right. If the replacement edge effects right sides, it takes constant time to replace G's free side if necessary. Otherwise the time is proportional to the number of edges in both G and the span of the replacement edge. Since all these edges have non-null left sides, at most one of them can remain in G. Hence the time is proportional to the contribution to $\bar{H}$. Thus the total time for the replacement step is $O(H+\bar{H})$ within 0-hourglasses. Adding this to the $O(E)$ time outside of 0-hourglasses gives the required total time $O(H+E+\bar{H})$.

LEMMA 5.6. *Let E, H, and $\bar{H}$ be as in Lemma 5.5. After performing the replacement step on a sequence of edges and 0-hourglasses, the following total is at least $H+\bar{H}+E-2$: Count 1 for each edge that has a non-null left side and is in the span of a left side replacement edge and 1 for each non-null right side in the span of a right side replacement edge.*

Proof. Let $\bar{T}$ be the total in the statement of the lemma. It is convenient to think of non-inflection edges as trivial 0-hourglasses, thereby reducing E and increasing H without affecting $E+H$.

The span of a left-side replacement edge includes two inflection edges and the right turn sequence between them; a right side replacement edge spans two inflections and a left turn sequence. The free side of any hour glass must belong to such a left or right turn sequence. Since we maintain maximum-size 0-hourglasses, two consecutive ones with no inflections between them must both have non-null free sides. Thus the contribution to $\bar{T}$ from free sides is H.

Since replacement edges span consecutive inflections, all but the first and last inflection belong to two replacement edge spans and thus have a non-null side in at least one of them. This contributes $E-2$ to the total $\bar{T}$.

The span of a left-side replacement edge can include left turn sequences and the same goes for right-side replacement edges and right turn sequences. All edges removed from 0-hourglasses belong to this class and there are at least $\bar{H}$ such edges. Thus $\bar{T} \geq H+E-2+\bar{H}$ as required.

THEOREM 5.3. *When using the four parameter version of smooth() defined at the end of Section 4.2 with modifications to keep track of hourglasses, the running time for Algorithm 4.1 is linear in the number of input vertices.*

Proof. For accounting purposes, let the cost of an edge be 1 if it has a null side and 2 otherwise. By Lemma 5.3, it suffices to show that the time for $smooth(e_0, e_n, e'_0, e'_n)$ is proportional to the cost of the edges between e'_0 and e'_n. The proof is by induction on the total cost. When the cost is small enough, there must be less than two inflections so that the replacement step fails immediately. In this case, the time is proportional to the number of edges scanned, hence proportional to the cost.

Between consecutive multi-inflection hourglasses G_1 and G_2, the replacement step generates one replacement edge for each inflection pair in the sequence of inflections starting with G_1's last inflection and ending with G_2's first one. Lemma 5.4 guarantees each replacement edge replaces at least two existing non-null edge sides so that the total cost reduction is at least half the quantity $\bar{T}$ in Lemma 5.6. Since G_1 and G_2 would have been merged if there were no vulnerable left or right turn sequence between them, we must have $\bar{T} \geq 1$. Thus the time given by Lemma 5.5 is proportional to the cost reduction.

The above argument also works for edges before the first multi-inflection hourglass or after the last one. If there are no such hourglasses, the argument applies to the entire input. If the entire $e'_0, \ldots, e'_n$ range is a single hourglass, the calls to *emark*() clearly produce a cost reduction proportional to their running time.

6 Performance

The algorithm has been implemented in C++ and tested on a variety of images. The program uses 32-bit integer arithmetic; floating point is used only to generate the output polygon. The implementation includes runtime checks for the invariants needed to prove Theorem 5.2, but these checks were disabled during timing tests. The modifications discussed in Section 5 were not implemented because large "hourglasses" are very rare in practice.

When comparing the performance of polygonal approximation algorithms, we must expect a trade-off between speed and the quality of the results. The fastest conceivable algorithm is to select every kth vertex of the input polygon, where k is some fixed parameter. This is probably much faster than any published algorithm, but it provides no real control over the quality of the approximation. No published algorithm is quite so simple-minded, but some of them do use simple rules to select a subset of the input vertices.

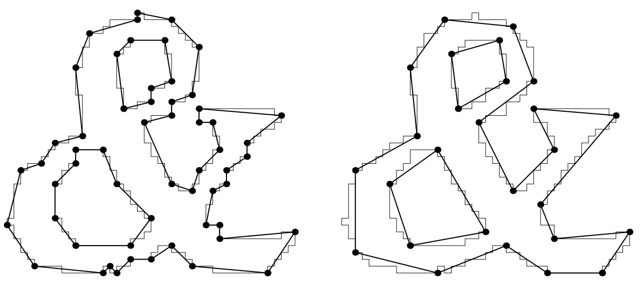
Table 1 gives comparative timings for a number of algorithms of this type [35, 30, 4, 34, 26, 17] as well as a number of other algorithms. By listing comparative timing data from available literature, the table gives a rough idea of the relative speed of a wide range of competing algorithms. All except for Pavlidis [23] produce a subset of the input vertices. The output for [23] is a sequence of disjoint line segments. Most of the sources agree that Robergé [26] gives the fastest algorithm, although Fahn et. al. [9] claim their's runs a little faster.

The comparative timings were done with Robergé's "extension factor" parameter equal to 1; Robergé suggests larger values for which his timings are 1.5 to 1.6 times slower. Thus Wall and Danielsson [34] is a reasonable choice for speed comparisons. An optimized C implementation was readily available, and this algorithm is popular in certain application areas. The implementation used one pass of Ramer's algorithm [24] to improve the output. Figure 10 shows the results of improved Wall and Danielsson, and Figure 11 shows how the results of Algorithm 4.1 compare.

Relative timings for Algorithm 4.1 and improved Wall and Danielsson appear in Table 2. Algorithm 4.1 is 9.5 to 14.5 times slower, but is still quite fast on an absolute scale.

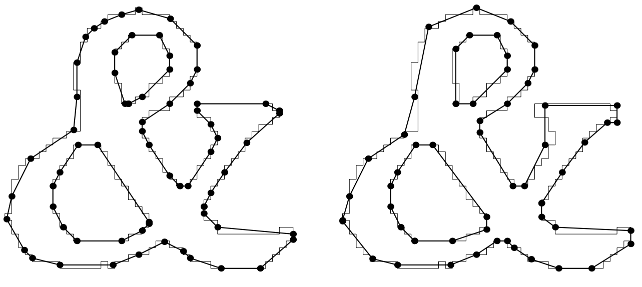
			Time (ms./vertex) for each algorithm												
Source	N	T	[24]	[23]	[35]	[6]	[30]	[4]	[22]	[34]	[26]	[8]	[9]	[17]	[3]
[26]	1000	1.6λ			.31						.10				
Cyber	1000	16λ			.28						.09				
[8]	402	1	75	600	75	66	82	118	88		66	241			
T.I.	402	4	67	376	94	64	105	74	86		55	1300			
[9]	292	1								5.4	5.0	12.6	4.2		
VAX	292	4								4.5	4.1	11.2	3.2		
[17]	522	≈ 1									.06			.12	
Sun 4	522	≈ 7									.04			.07	
[3]	1337	big			8.3										20.2
Apollo	1750	big			9.0										7.3

Table 1: Published computation times in milliseconds per input vertex. The N and T columns give the number of input vertices and the error tolerance in pixels or as a multiple of the average input segment length λ. Robergé [26] used a CDC Cyber 6000; Dunham [8] used a T.I. Professional Computer; Fahn et. al. [9] used a VAX 11/780; Leung and Yang [17] used a Sun 4/280; and Aoyama and Kawagoe [3] used an Apollo DN3000.



(a) (b)

Figure 10: Results from improved Wall and Danielsson with tolerances (a) 1 pixel; (b) 2 pixels. Dots mark output vertices and thin lines show the input path.



(a) (b)

Figure 11: Results from Algorithm 4.1 with tolerances (a) 1 pixel; (b) 2 pixels. Dots mark output vertices and thin lines show the input path.

Input	N	T	Algorithm 4.1	Wall & D.
"&"	196	1	67	7.0
		2	70	6.7
		4	75	7.2
"B•C•"	848	1	61	6.4
		2	61	6.0
		4	65	6.7
fingerprint	31536	1	91	6.7
		2	89	6.3
		4	89	6.1

Table 2: timings in *microseconds* per input vertex on a 40Mhz. MIPS R3000 processor. The N and T columns give input vertices and error tolerance. Inputs were the "&" in Figure 1a, 102-pixel-high outlines for the letters "B•C•", and fingerprint outlines excerpted in Figure 2.

7 Conclusion

We have seen a polygonal approximation algorithm with guaranteed smoothness properties and superior aesthetics. It is designed to preserve input features as well as possible and only throw away noise. Keeping track allowable deviations from the output polygons would allow for post-processing to minimize the vertex count or compute spline approximations. There is a time penalty for all this, but the implementation is fast enough to be very practical: up to 15,000 vertices per second on a MIPS R3000. (A page of text scanned at 400 dots/inch produces about 100,000 input vertices).

References

[1] A. Albano. Representation of digitized contours in terms of conic arcs and straight-line segments. *Comput. Gr. and Image Proc.*, 3(1):23–33, Mar. 1974.

[2] N. Ansari and E. J. Delp. On detecting dominant points. *Pattern Recog.*, 24(4):441–451, 1991.

[3] H. Aoyama and M. Kawagoe. A piecewise linear approximation method preserving visual feature points of original figures. *CVGIP Gr. Models Image Proc.*, 53(5):435–446, Sept. 1991.

[4] F. Badi'i and B. Peikari. Functional approximation of planar curves via adaptive segmentation. *Int. J. Syst. Sci.*, 13(6):667–674, June 1982.

[5] J. P. Bixler, L. T. Watson, and J. P. Sanford. Spline-based recognition of straight lines and curves in engineering line drawings. *Image Vis. Comput.*, 6(4):262–269, Nov. 1988.

[6] L. S. Davis. Shape matching using relaxation techniques. *IEEE Trans. Patt. Anal. and Machine Intel.*, PAMI-1(1):60–72, Jan. 1979.

[7] G. Dettori. An on-line algorithm for polygonal approximation of digitized plane curves. In *Proceedings of the 6th International Joint Conference on Pattern Recognition*, volume 2, pages 739–741, 1982.

[8] J. G. Dunham. Optimum uniform piecewise linear approximation of planar curves. *IEEE Trans. Patt. Anal. and Machine Intel.*, PAMI-8(1):67–75, Jan. 1986.

[9] C.-S. Fahn, J.-F. Wang, and J.-Y. Lee. An adaptive reduction procedure for the piecewise linear approximation of digitized curves. *IEEE Trans. Patt. Anal. and Machine Intel.*, 11(9):967–973, Sept. 1989.

[10] H. Freeman and J. M. Glass. On quantization of line-drawing data. *IEEE Trans. Syst. Sci. and Cybernetics*, SSC-5(1):70–79, Jan. 1969.

[11] M. Gangnet. Approximation of digitized contours. In R. A. Earnshaw, editor, *Theoretical Foundations of Computer Graphics and CAD*, pages 833–855. Springer Verlag, 1988.

[12] T. L. Hemminger and C. A. Pomalaza-Ráez. Polygonal represenctaion: A maximum likelihood approach. *Comput. Vision Gr. and Image Proc.*, 52(2):239–247, Nov. 1990.

[13] J. D. Hobby. Smoothing digitized contours. In R. A. Earnshaw, editor, *Theoretical Foundations of Computer Graphics and CAD*, pages 777–793. Springer Verlag, 1988.

[14] H. Imai and M. Iri. Computational-geometric methods for polygonal approximation of a curve. *Comput. Vision Gr. and Image Proc.*, 36(1):31–41, Oct. 1986.

[15] Y. Kurozumi and W. A. Davis. Polygonal approximation by the minimax method. *Comput. Gr. and Image Proc.*, 19(3):248–264, July 1982.

[16] D. Lee, G. W. Wasilkowski, and R. Mehrotra. A new zero-crossing-based discontinuity detector. *IEEE Trans. Image Proc.*, to appear.

[17] M. K. Leung and Y.-H. Yang. Dynamic strip algorithm in curve fitting. *Comput. Vision Gr. and Image Proc.*, 51(2):145–165, Aug. 1990.

[18] Y.-Z. Liao. A two-stage method of fitting conic arcs and straight line segments to digitized contours. In *Proceedings of the IEEE Computer Society Conference on Pattern Recognition and Image Processing*, pages 224–229, Aug. 1981.

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

[20] T. Pavlidis. Polygonal approximations by Newton's method. *IEEE Trans. Comput.*, C-26(8):801–807, Aug. 1977.

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

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

[23] T. Pavlidis and S. L. Horowitz. Segmentation of plane curves. *IEEE Trans. Comput.*, C-23(8):860–870, Aug. 1974.

[24] U. Ramer. An iterative procedure for the polygonal approximation of plane curves. *Comput. Gr. and Image Proc.*, 1(3):244–256, Nov. 1972.

[25] K. Reumann and A. P. M. Witkam. Optimizing curve segmentation in computer graphics. In A. Gunther, B. Levrat, and H. Lipps, editors, *International Computing Symposium*, pages 467–472. American Elsevier, 1974.

[26] J. Robergé. A data reduction algorithm for planar curves. *Comput. Vision Gr. and Image Proc.*, 29(2):168–195, Feb. 1985.

[27] P. L. Rosin and G. A. W. West. Segmentation of edges into lines and arcs. *Image Vis. Comput.*, 7(2):109–114, 1989.

[28] J. Sklansky. Recognition of convex blobs. *Pattern Recog.*, 2(1):3–10, Jan. 1970.

[29] J. Sklansky, R. L. Chazin, and B. J. Hansen. Minimum perimeter polygons of digitized silhouetts. *IEEE Trans. Comput.*, C-21(3):260–268, Mar. 1972.

[30] J. Sklansky and V. Gonzalez. Fast polygonal approximation of digitized contours. *Pattern Recog.*, 12(5):327–331, 1980.

[31] J. Sklansky and D. F. Kibler. A theory of nonuniformity in digitized binary pictures. *IEEE Trans. Syst., Man, and Cybernetics*, SMC-6(9):637–647, Sept. 1976.

[32] M. K. Sparrow. *Topological Coding of Single Fingerprints.* PhD thesis, University of Kent at Canterbury, Aug. 1985.

[33] I. Tomek. Two algorithms for piecewise-linear continuous approximation of functions of one variable. *IEEE Trans. Comput.*, 23(4):445–448, Apr. 1974.

[34] K. Wall and P.-E. Danielsson. A fast sequential method for polygonal approximation of digitized curves. *Comput. Vision Gr. and Image Proc.*, 28(2):220–227, Nov. 1984.

[35] C. M. Williams. An efficient algorithm for the piecewise-linear approximation of planar curves. *Comput. Gr. and Image Proc.*, 8(2):286–293, Oct. 1978.

[36] C. M. Williams. Bounded straight-line approximation of digitized planar curves and lines. *Comput. Gr. and Image Proc.*, 16(4):370–381, Aug. 1981.