TR 85_024

TRIANGULATION ALGORITHMS FOR SIMPLE, CLOSED, NOT NECESSARILY CONVEX, POLYGONS IN THE PLANE

— I I —

JOHN H. HALTON

The University of North Carolina
Department of Computer Science
New West Hall 035 A
Chapel Hill, NC 27514 USA

JULY 21, 1985

TRIANGULATION ALGORITHMS FOR SIMPLE, CLOSED, NOT NECESSARILY CONVEX POLYGONS IN THE PLANE

II

John H. Halton
Professor of Computer Science
The University of North Carolina
Chapel Hill, NC 27514 USA

ABSTRACT

This paper, a sequel to an earlier work of the same title, refines one algorithm, of an iterative character, presented there, and describes a new algorithm, recursive in nature. Experimental results are given for 22 polygons, together with annotated listings of four "C" programs implementing the algorithms. The time complexity of both algorithms is fully analysed and shown to be $O(n^2)$ where n is the number of vertices in the polygon to be triangulated. Counts of arithmetic operations are bounded by $54(n^2 - \frac{9}{2}n + 5)$ and $\frac{27}{2}(n^2 - 3n + 2)$, respectively. The new algorithm is therefore preferable when recursion is possible without too much labor.

Keywords: Algorithms; data structures; triangulation; polygons; graphics {computers; techniques; performance analysis; complexity}

July 1985

Triangulation Algorithms for Simple, Closed, Not Necessarily Convex Polygons in the Plane, II.

John H. Halton, Chapel Hill, North Carolina, USA

1. Introduction

The present paper is a sequel to the earlier one of the same title [1]. This deals with the classic problem, in which we are given n points P_1, P_2, ..., P_n in the Euclidean plane (with $P_0 = P_n$ and, in general, $P_j = P_{j+kn}$, for any integers j and k), ordered so as to define the *polygon*

$$\mathfrak{P} \equiv P_1P_2 \dots P_n, \tag{1}$$

with vertices P_j ($j = 1, 2, \dots, n$), consisting of the n line-segments P_jP_{j+1} ($j = 1, 2, \dots, n$); which is *simple*, in the sense that all the P_j are distinct and no two sides P_iP_{i+1} and P_jP_{j+1} have any points in common, except when $i = j$ [of course] or $i = j - 1$ [only P_j in common] or $i = j + 1$ [only P_i in common]. We now seek to identify a set of *triangles*, whose interiors are disjoint, and whose union is the interior and boundary of the polygon $\mathfrak{P}$. The triangulation we seek is *economical*, in the sense that there are at most $n - 2$ triangles in it, and the vertices of these triangles are all vertices P_j of the polygon $\mathfrak{P}$.

We adopt the following convention. The removal of the polygon $\mathfrak{P}$ from the Euclidean plane in which it lies leaves two connected components, each an open set. That which is *bounded* is the *interior* $I_{\mathfrak{P}}$; the other is the *exterior* $E_{\mathfrak{P}}$. We now assume that the vertices of $\mathfrak{P}$ are numbered so that, in traversing the polygon in the order P_1, P_2, ..., P_n, the interior $I_{\mathfrak{P}}$ is on the *left*. We define the angle by which one turns from the direction of $P_{j-1}P_j$ to that of P_jP_{j+1} to be θ_j in the range $-\pi < \theta_j < \pi$. If $\theta_j > 0$, we call the vertex P_j *convex* (corresponding to a turn to the *left*); if $\theta_j < 0$, we call P_j *re-entrant* (corresponding to a turn to the *right*); and if $\theta_j = 0$, we say that P_j is *redundant* or *collinear*.

We use the name *triad* for a triangle whose vertices are vertices of the given polygon $\mathfrak{P}$, and write

$$\Delta_j = P_{j-1}P_jP_{j+1} \qquad \text{and} \qquad \delta(i, j, k) = P_iP_jP_k. \tag{2}$$

Lemma 7: The vertex P_k lies inside the convex triad Δ_j if and only if

$$\left.\begin{array}{l}\gamma[j-1,\ j,\ k] > 0,\\ \gamma[j,\ j+1,\ k] > 0,\\ \text{and}\quad \gamma[j+1,\ j-1,\ k] > 0.\end{array}\right\} \qquad (7)$$

Lemma 8: No simple, closed polygon has an empty interior.

Theorem 1: Every simple, closed polygon $\mathfrak{P}$ has at least two convex triads Δ_r and Δ_s each containing no other vertex of $\mathfrak{P}$.

Lemma 9: If the convex triad $\Delta_j = P_{j-1}P_jP_{j+1}$ does contain certain vertices, then the vertices P_{h^-} and P_{h^+} among them, respectively having the least values of $\gamma[j-1,\ j,\ h^-]$ and $\gamma[j,\ j+1,\ h^+]$ are re-entrant, and the corresponding triads $P_{j-1}P_jP_{h^-}$ and $P_jP_{j+1}P_{h^+}$ are empty convex triads.

Lemma 13: The bound in Lemma 5 is tight: there are polygons of any number of vertices $n \geqslant 3$ with only three convex vertices.

(Additional Lemmas 10, 11, and 12, and Theorems 2, 3, and 4, are of a more technical nature and refer to various triangulation algorithms presented in [1].)

Of the algorithms presented in [1], Algorithm 0 is preparatory and identifies the convex and re-entrant vertices of the given polynomial into respective lists named $\mathcal{A}$ and $\mathcal{B}$. (Use is made of Lemmas 1 and 4.) This is refined in Algorithm 0* and in the actual program presented on pages 47 - 61 (Section 7) of [1]. Algorithm 1, for successive vertices in list $\mathcal{A}$, tests all vertices in list $\mathcal{B}$ for inclusion in the corresponding convex triad with apex in list $\mathcal{A}$. (Use is made of Lemmas 5, 6, and 7, and Theorem 1.) When an empty convex triad is encountered, it is added to the triangulation list $\mathcal{C}$ and lists $\mathcal{A}$ and $\mathcal{B}$ are adjusted accordingly. This is iterated until the triangulation is complete. Algorithm 1* is a modification of this algorithm preparatory to Algorithm 3, setting up lists of included vertices $\boldsymbol{u}_i$ and corresponding inverse lists $\boldsymbol{t}_k$ (see also the actual program presented in [1]). Algorithm 2 uses Lemma 9 to permit positive action at every convex triad; leading to the splitting of the polygon into two or three disjoint simple closed polygons to be triangulated, whenever a non-empty triad is encountered. Algorithm 3 is a variant of Algorithm 1 in which information previously obtained is retained (via Algorithm 1*), modified only when necessary, and re-used, rather than re-calculated at each iteration. This is refined in the program presented in [1].

In what follows, results are called Propositions, to distinguish them from the Lemmas of the earlier paper [1].

2. Basic Propositions

PROPOSITION 1. *A convex polygon has only convex or collinear vertices.*

PROPOSITION 2. *A polygon with only convex or collinear vertices is convex.*

PROPOSITION 3. *Given a convex polygon* $\mathfrak{K}$ *and a general polygon* $\mathfrak{P}$ *entirely in or on* $\mathfrak{K}$, *if a vertex* P_j *of* $\mathfrak{P}$ *lies on* $\mathfrak{K}$, *then* P_j *is either a convex or collinear vertex of* $\mathfrak{P}$.

COROLLARY 3.1. *If the vertices of a simple closed polygon* $\mathfrak{P}$ *have the coordinates* (5), *then the vertices satisfying* (6) *are all convex or collinear vertices of* $\mathfrak{P}$.

COROLLARY 3.2. *Under the conditions of Proposition 3, the first and last vertices of* $\mathfrak{P}$ *(in the order in which they appear in* $\mathfrak{P}$*) in any side of* $\mathfrak{K}$ *are convex vertices of* $\mathfrak{P}$.

COROLLARY 3.3. *Under the conditions of Corollary 3.1, the first and last vertices of* $\mathfrak{P}$ *(in the order in which they appear in* $\mathfrak{P}$*) satisfying any one of the conditions* (6) *are convex vertices of* $\mathfrak{P}$.

COROLLARY 3.4. *Under the conditions of Corollary 3.1, any of the vertices of* $\mathfrak{P}$ *satisfying*

$$x_i = \boldsymbol{E}_j x_j \quad \text{and} \quad y_i = \boldsymbol{F}\,\{y_k \mid x_k = x_i\}\,, \tag{8}$$

or

$$y_i = \boldsymbol{E}_j y_j \quad \text{and} \quad x_i = \boldsymbol{F}\,\{x_k \mid y_k = y_i\}\,, \tag{9}$$

where each of $\boldsymbol{E}$ *and* $\boldsymbol{F}$ *represents either "min" or "max", is a convex vertex of* $\mathfrak{P}$.

Proof. As in [1], proofs will be enclosed in double brackets ⟦...⟧. Propositions 1, 2, and 3, respectively present amendments to Lemmas 2, 3, and 4, of [1]. The proofs of these results stand essentially unchanged, with the observation that, when a vertex P_j is *not* re-entrant ($\Gamma_j \nless 0$), then it is *either* convex *or collinear* ($\Gamma_j > 0$ *or* $\Gamma_j = 0$); the latter alternative was previously omitted.

Corollary 3.1 is similarly a restatement of the corollary to Lemma 4. We are left with Corollaries 3.2, 3.3, and 3.4, which provide the needed sharpening of the preceding results.

⟦Let AB be a side of the convex polygon $\mathfrak{K}$ containing at least three vertices of the included simple closed polygon $\mathfrak{P}$, with the customary interior-on-the-left traversal of $\mathfrak{K}$ going from A to B. Let P and Q be two vertices of $\mathfrak{P}$ lying in AB. Denote the polygonal arc of $\mathfrak{P}$ traversed from P to Q in the interior-on-the-left direction by ⟨PQ⟩, and the remainder of $\mathfrak{P}$, traversed from Q to P by ⟨QP⟩. If all vertices of $\mathfrak{P}$ in ⟨PQ⟩ lie on AB (i.e., ⟨PQ⟩ is simply a line-segment PQ lying on AB), then their order in AB is the same as their order in $\mathfrak{P}$. If not, the segment PQ of AB consists of sub-segments shared with ⟨PQ⟩ (in which vertices are ordered as in $\mathfrak{P}$) and sub-segments not so shared, in which the corresponding pieces of ⟨PQ⟩ lie in the interior of $\mathfrak{K}$. Let XY be one of the latter intervals (with X and Y vertices of $\mathfrak{P}$, of course). Then suppose that R is a vertex of $\mathfrak{P}$ in ⟨QP⟩; if R lay in XY, the edges of $\mathfrak{P}$ through it would have to lie on the interior side of $\mathfrak{K}$ and parts of them would have to be inside the simple closed polygon formed by ⟨XY⟩ and the segment YX; but this is impossible, since $\mathfrak{P}$ can neither cross $\mathfrak{K}$ nor itself. Thus we see that no vertex of $\mathfrak{P}$ not in ⟨PQ⟩ can lie in AB between P and Q; or, in other words, the order of vertices of $\mathfrak{P}$ in AB is the same as their order in $\mathfrak{P}$. From this we conclude that, of the vertices of $\mathfrak{P}$ lying in any side AB of $\mathfrak{K}$, the nearest to A and to B are respectively the first and the last, in the order in which they are traversed in $\mathfrak{P}$. The reasoning of the proof of Lemma 4 now shows that these particular vertices are *strictly convex.* (They cannot be collinear and be first or last in AB.) This proves Corollary 3.2.⟧

⟦Corollary 3.3 takes for $\mathfrak{K}$ the rectangle with vertices $(E_j x_j,\ F_j y_j,\ 0)$, with each of E and F denoting either "min" or "max", as in (8) and (9). It follows from Corollary 3.2 that the first and last vertices (as ordered in $\mathfrak{P}$) lying in any side of this rectangle (whose equation is one of the equations (6)) are strictly convex. The conditions (8) and (9) identify those of the vertices of $\mathfrak{P}$ lying in a side of the rectangle, nearest to the end-points of this side. As has been proved above, these are precisely the first and last of these in the order in which they are traversed in $\mathfrak{P}$. This proves Corollary 3.4.⟧

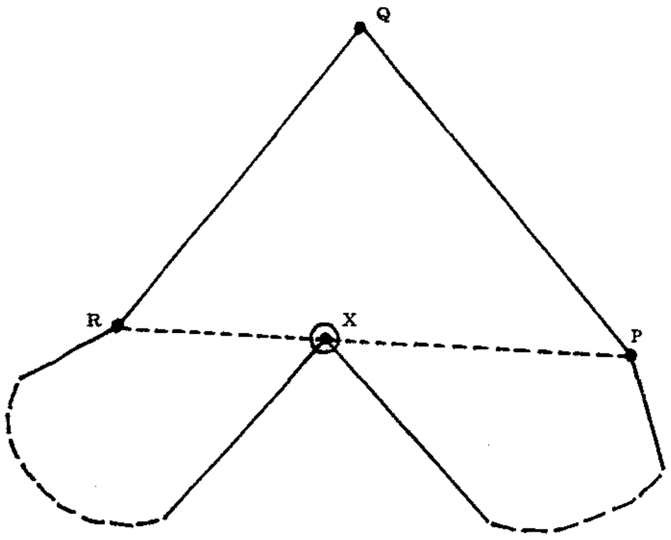
Lemmas 1 (with corollary), 5, 6, 8, and 13, and Theorem 1, are taken over unchanged. While Lemma 7 is correct, it is preferable to use the following result (which is an immediate corollary).

PROPOSITION 4. *The vertex* P_k *lies inside or on the convex triad* Δ_j *if and only if*

$$\left.\begin{aligned} \gamma[j-1,\ j,\ k] &\geqslant 0, \\ \gamma[j,\ j+1,\ k] &\geqslant 0, \\ \gamma[j+1,\ j-1,\ k] &\geqslant 0. \end{aligned}\right\} \qquad (10)$$

This prevents the problem encountered when removal of a triad leaves two contiguous simple closed polygons (or a closed polygon which crosses itself). Figure (i) illustrates this: the vertex X lies on the line PR, satisfying (10) but not (7). Using the negation of (7), we would try to remove the triad PQR, leaving polygons PX ∪ ⟨XP⟩ and XR ∪ ⟨RX⟩.

Figure (i).



The above results suffice to establish the validity of Algorithms 0, 0*, 1, 1*, and 3. In [1], we used Lemma 9 to justify Algorithm 2. Here, a difficulty has been discovered. The concept of an "empty" triad turns out to have been left very slightly vague; and we must now sharpen this. Following Lemma 7 and Theorem 1 of [1], we define:

DEFINITION 1. *A triad* $\delta(i, j, k) = P_iP_jP_k$ *is said to be "empty" iff there is no vertex of* $\mathfrak{P}$ *other than* P_i, P_j, *or* P_k *within or on it.*

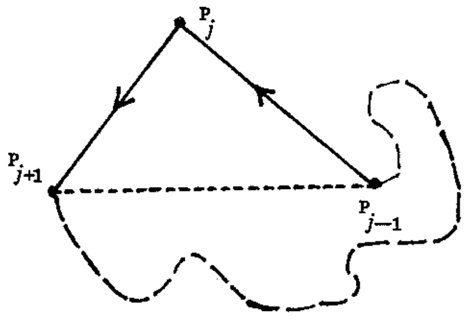
Note that we have moved from (7) to (10), as indicated in Proposition 4 above. With either this definition or the earlier concept based on (7), we see that Lemma 9 stands as stated. However, emptiness turns out not to ensure that the triad may be removed! We define:

DEFINITION 2. *A triad* $\delta(i, j, k) = P_iP_jP_k$ *is said to be "removable" if and only if* (a) *it is empty* (in the sense of Definition 1), (b) *no side of* $\mathfrak{P}$ *intersects its interior, and* (c) *its interior is part of the interior of* $\mathfrak{P}$.

PROPOSITION 5. *If* $\mathfrak{P}$ *is a simple closed polygon with* $n \geqslant 4$ *vertices, and* Δ_j *is a triad, removable* (in the sense of Definition 2) *from* $\mathfrak{P}$*; then the polygon* $\mathfrak{P}'$ *obtained from* $\mathfrak{P}$ *by removing the vertex* P_j *and directly connecting* P_{j-1} *to* P_{j+1} *is also simple and closed, with one less vertex than* $\mathfrak{P}$.

⟦The triad $\Delta_j = P_{j-1}P_jP_{j+1}$, and, since $n \geqslant 4$, there is at least one more vertex of $\mathfrak{P}$. Since Δ_j is empty and no side of $\mathfrak{P}$ intersects the interior of Δ_j, all sides of $\mathfrak{P}$ except for $P_{j-1}P_j$ and P_jP_{j+1} lie entirely outside the triad. If the interior of Δ_j is part of the interior of $\mathfrak{P}$, P_j must be a convex vertex and so Δ_j must be a convex triad. The proposition now follows (see Figure (ii)).⟧

Figure (ii).



PROPOSITION 6. *If* Δ_j *is a convex triad of a simple closed polygon* $\mathfrak{P}$, *and if it is empty* (in the sense of Definition 1)*; then it is removable* (in the sense of Definition 2) *from* $\mathfrak{P}$.

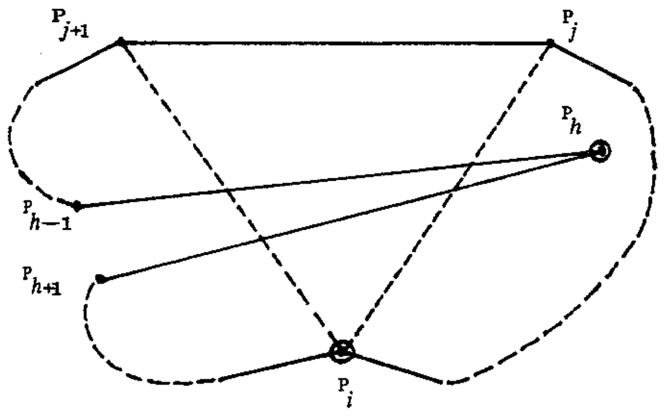
⟦Since the triad is convex, its interior is part of the interior of $\mathfrak{P}$. (The two interiors are disjoint if the triad is re-entrant.) Since it is empty, no vertex of $\mathfrak{P}$ other than P_{j-1}, P_j, and P_{j+1} lies in or on Δ_j. All that remains is to prove condition (b) of Definition 2, that no side of $\mathfrak{P}$ intersects the interior of Δ_j. Both ends of such a side must be outside the triad; and, since the polygon is simple, the side cannot intersect $P_{j-1}P_j$ or P_jP_{j+1}. But any line which intersects the interior of a triangle must cross two of its sides; so a segment which is part of such a line and which has its ends outside the triangle either lies entirely outside the triangle or crosses two sides of the triangle. This completes the proof of the proposition.⟧

It is this result which justifies all the algorithms except Algorithm 2 and permits the vagueness about distinguishing emptiness and removability.

PROPOSITION 7. *It is possible for a triad* $\delta(i, j, k) = P_iP_jP_k$ *of a simple closed polygon* $\mathfrak{P}$ *to be empty* (in the sense of Definition 1), *and yet for it not to be removable from* $\mathfrak{P}$.

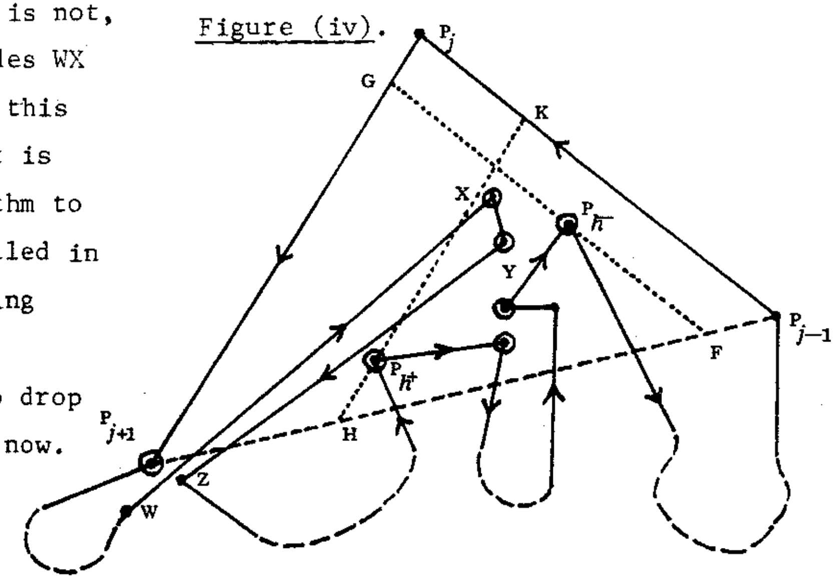
⟦An illustration (counter-example to the extension of Proposition 6 to non-consecutive triads) is given in Figure (iii). The triad is $P_iP_jP_{j+1}$ and, though it is empty (i.e., contains no other vertices of $\mathfrak{P}$), the sides $P_{h-1}P_h$ and P_hP_{h+1} cross its interior, in contradiction of Definition 2.⟧

Figure (iii).



As a further example of this proposition, consider Figure (iv), which is a variation on the theme illustrated in Figures 12 and 13 of [1], exemplifying Lemma 9 there. The line FG is parallel to the side $P_{j-1}P_j$ of the convex triad Δ_j, intersecting $P_{j-1}P_{j+1}$ in F and P_jP_{j+1} in G, and passes through P_{h^-}; $P_{j-1}P_jP_{h^-}$ is then necessarily empty (as indeed is the whole of the quadrilateral $FGP_{j-1}P_j$). The line HK, similarly, is parallel to P_jP_{j+1}, intersecting $P_{j-1}P_{j+1}$ in H and $P_{j-1}P_j$ in K, and passes through P_{h^+}; $P_jP_{j+1}P_{h^+}$ is then empty (as is HKP_jP_{j+1}). We see that $P_{j-1}P_jP_{h^-}$ is, in fact, removable; but that $P_jP_{j+1}P_{h^+}$ is not, since it is crossed by the sides WX and YZ of $\mathfrak{P}$. It follows from this that Algorithm 2 can fail. It is possible to refine the algorithm to deal with the difficulty revealed in Proposition 7; but the resulting procedure becomes excessively laborious; so it seems best to drop the algorithm altogether, for now.

Figure (iv).



PROPOSITION 8. *The interior of a triad is part of the interior of the polygon only if no side of the polygon intersects the interior of the triad.*

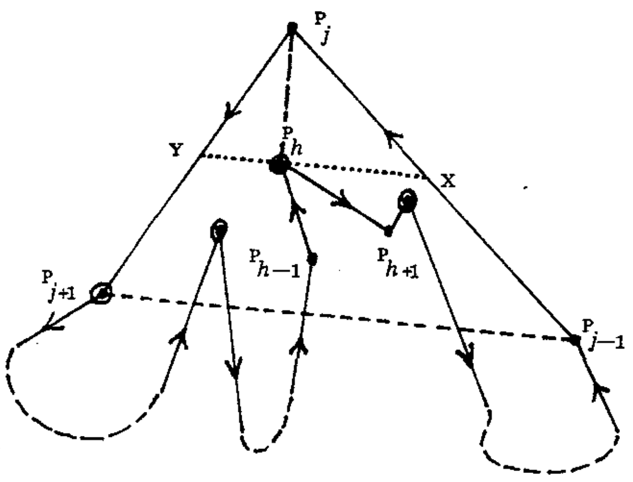
⟦If a side of $\mathfrak{P}$ intersects the interior of the triad, then it separates the interior $I_{\mathfrak{P}}$ and the exterior $E_{\mathfrak{P}}$ of $\mathfrak{P}$; whence part of the interior of the triad is in $E_{\mathfrak{P}}$.⟧ Thus, Condition (c) of Definition 2 implies Condition (b) [and the old form of Condition (a)]. The new form of Condition (a) [see (10), Proposition 4, and Definition 1] requires further that the case shown in Figure (i) not occur (i.e., no extraneous vertex of $\mathfrak{P}$ lie *on* the triad.

The following proposition was communicated privately (without proof) to the author by Dr T. H. Brylawski. The proof given here is the present author's.

PROPOSITION 9 [*BRYLAWSKI*]. *Consider any convex triad* Δ_j *of a simple closed polygon* $\mathfrak{P}$. *Either* (a) *the triad is empty; in which case it is removable, leaving a polygon of one less vertex; or* (b) *there is a vertex* P_h *having the greatest value of* $\gamma[j - 1, h, j + 1]$ *among all vertices in the triad; in which case no side of* $\mathfrak{P}$ *crosses the segment* P_hP_j, *and this segment creates two contiguous simple closed polygons, each with less vertices than* $\mathfrak{P}$. *In both cases, iteration of the process leads to a full, economical triangulation of the polygon* $\mathfrak{P}$.

⟦If Δ_j is an empty convex triad; then, by Propositions 5 and 6, it is removable from $\mathfrak{P}$, leaving a simple closed polygon $\mathfrak{P}'$ of one less vertex, so long as $\mathfrak{P}$ has at least four vertices; and when there are only three vertices left, the removal of the triad completes the process of triangulation. If Δ_j is *not* empty, then, by Definition 1, it must contain at least one vertex other than P_{j-1}, P_j, and P_{j+1}, and therefore there is a vertex P_h having the property stated in (b). It then follows (see Figure (v) and the observation below) that there is a line XY, parallel to $P_{j-1}P_{j+1}$, intersecting $P_{j-1}P_j$ at X and P_jP_{j+1} at Y, and passing through P_h, such that the triangle XP_jY is empty (further, P_h either lies in the interior of the triad or in the segment $P_{j-1}P_{j+1}$, since it cannot lie in the sides $P_{j-1}P_j$ or P_jP_{j+1}). An argument exactly similar to that used to prove Proposition 6 shows that no side of $\mathfrak{P}$ can intersect the interior of the triangle XP_jY; so that no side can cross

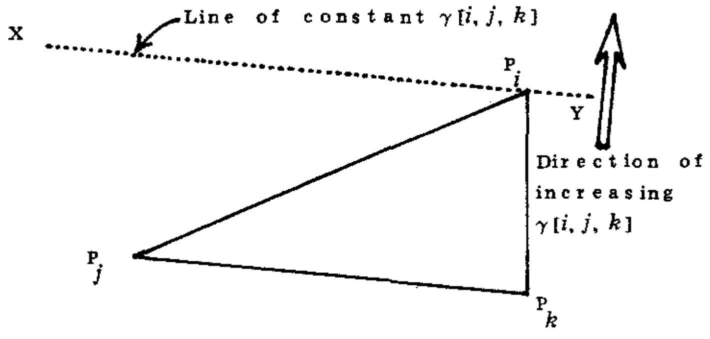
Figure (v).



the segment P_hP_j. Now, P_h and P_j divide 𝔓 into the non-intersecting polygonal arcs $\langle P_hP_j\rangle$ and $\langle P_jP_h\rangle$; with $\langle P_hP_j\rangle$ containing at least P_{j-1} and $\langle P_jP_h\rangle$ containing at least P_{j+1}. Thus, the addition of P_hP_j to 𝔓 yields two contiguous simple closed polygons, $P_jP_h \cup \langle P_hP_j\rangle$ and $P_hP_j \cup \langle P_jP_h\rangle$, each having less vertices than 𝔓. Iteration of the process yields polygons with strictly decreasing numbers of vertices; so that the entire algorithm must terminate. The resulting triangulation is economical, since every triad removed has vertices of 𝔓 as its vertices.⟧

We should point out that the discriminant $\gamma[i, j, k]$ defined in (3) is equal to twice the area of the triangle $P_iP_jP_k$; that is, the product of the length of P_jP_k and the vertical height of P_i above P_jP_k. Thus (see Figure (vi)) $\gamma[i, j, k]$ is constant, for fixed P_j and P_k, for all positions of P_i in a line XY parallel to P_jP_k. As XY moves upwards above P_jP_k, the value of $\gamma[i, j, k]$ increases (below P_jP_k, γ is negative). This fact justifies the statements in Lemma 9, Proposition 9, and the proofs thereof. [For example, in Figure (v), if P_h has maximal $\gamma[j - 1, h, j + 1]$, then there can be no vertex above the line XY, in the triad $P_{j-1}P_jP_{j+1}$.]

Figure (vi).


X
Line of constant $\gamma[i, j, k]$
P_i
Y
Direction of increasing $\gamma[i, j, k]$
P_j
P_k

PROPOSITION 10. *In Case* (b) *of Proposition 9, any maximal vertex* P_h *as defined there must be re-entrant or collinear; and at least one such* P_h *has to be re-entrant, unless the triad* Δ_j *coincides with the polygon* 𝔓.

⟦We argue similarly to the proofs of Lemma 4, Proposition 3, and Corollary 3.2. Sides $P_{j-1}P_j$ and P_jP_{j+1} cannot intersect the interior of the triangle

XP_jY; so the angle $\theta_h \leqslant 0$ (see page 1); whence P_h is either re-entrant or collinear, as stated. Now P_h must be strictly between X and Y (even if X = P_{j-1} and Y = P_{j+1}); so either the triad (or triangle) $P_{j-1}P_jP_{j+1}$ coincides with the polygon $\mathfrak{P}$ (i.e., all other vertices of $\mathfrak{P}$ are in the segment $P_{j-1}P_{j+1}$), or there is at least one vertex of $\mathfrak{P}$ (other than P_{j-1} and P_{j+1}) exterior to the triangle XP_jY. In the latter case, there must be a vertex P_h lying in XY whose predecessor or successor in $\mathfrak{P}$ lies off XY; and this vertex is re-entrant.⟧

Propositions 9 and 10 provide us with the foundation of a new algorithm, which will not fail, and which turns out to be quite efficient and simple to program.

3. The Programs

Four new programs have been written and tested, using the improved results of the present study. Program A was a refined version of the program listed in [1] (on pages 47 - 61). Applying the new program to the same four polygons used as examples in [1], we find that the arithmetic-operation (a.o.) counts are considerably improved:

POLYGON	n	A.O. COUNT		IMPROVEMENT
		OLD	NEW	(OLD - NEW)/OLD
1	15	4,257	2,511	41.01%
2	20	6,579	4,311	34.47%
3	27	11,367	7,839	31.04%
4	48	36,306	29,088	19.88%

Program B was essentially the same as Program A, but modified so as to keep track of elapsed execution time (excluding input and output operations) and to work with a family of "double square spiral" polygons of $8i$ vertices, for i = 1, 2, 3, ..., 12. Output was somewhat terser than from Program A.

Program C was similar to Program B, except that the algorithm, instead of being essentially Algorithm 3 of [1], was the new algorithm based on Propositions 9 and 10 above.

Because elapsed time is perturbed by the interstitial accounting and monitoring functions of the Unix operating system, each instance of the last two programs was run three times and the least elapsed time recorded. Even so, the results were a little erratic and truncated to whole seconds.

Program A was written to input coordinates of the vertices of a polygon and triangulate it. Programs B and C input the parameter i only, generated the corresponding member of the family of double square spirals (with $8i$ vertices), and triangulated it. Program D bore a similar relation to Program A as Program C did to Program B (new algorithm; arbitrary polygon input).

The results for Programs B and C, for the family of double square spirals, are shown below:

POLYGON PARAMETER	n	A. O. COUNT		RATIO OF COUNTS	ELAPSED TIME*	
		PROGRAM B	PROGRAM C		PROG. B	PROG. C
1	8	513	207	2.48	0*	0*
2	16	3,393	972	3.49	1*	0*
3	24	8,829	2,169	4.07	4	1*
4	32	16,821	3,798	4.43	6	1*
5	40	27,369	5,859	4.67	11	2
6	48	40,473	8,352	4.85	16	3
7	56	56,133	11,277	4.98	22	5*
8	64	74,349	14,634	5.08	30	5*
9	72	95,121	18,423	5.16	38	7
10	80	118,449	22,644	5.23	48	9
11	88	144,333	27,297	5.29	60	11
12	96	172,773	32,382	5.34	72	13

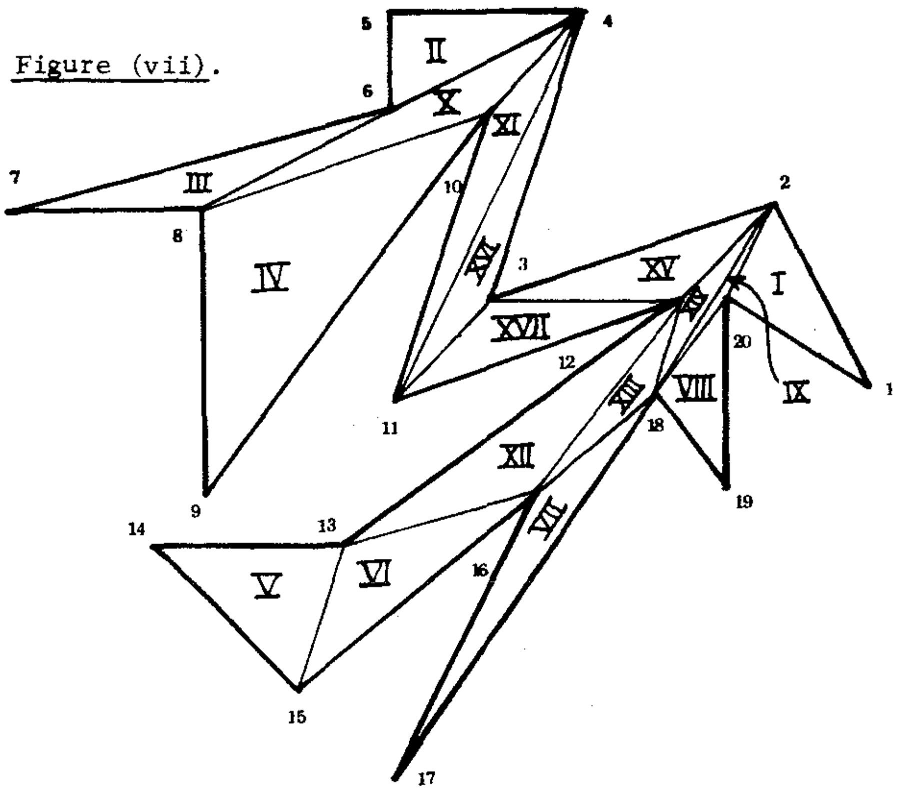
*Note: elapsed times are given to the nearest second only.

The results for Programs A and D are compared below:

POLYGON PARAMETER	n	A.O. COUNT		RATIO OF COUNTS
		PROGRAM A	PROGRAM D	
1	15	2,511	693	3.62
2	20	4,311	1,008	4.28
3	27	7,839	2,493	3.14
4	48	29,088	4,023	7.23

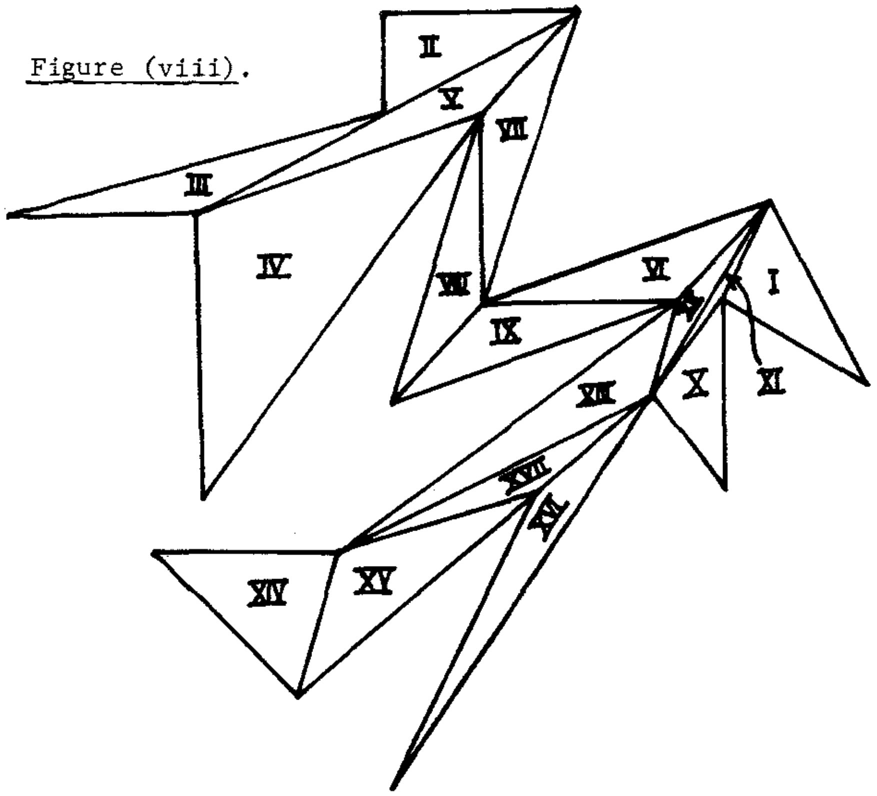
The four example-polygons used in [1] are fully described there. In Figures (vii) and (viii), we show the triangulations of the second of these (Figure 30 of [1], a 20-gon), as performed by Programs A and D, respectively.

Figure (vii).



We note that the order of triangulation is not the same in the two figures, though we begin with the same convex vertex P_1 in each case; and further, the final triangulations differ in a few particulars (the quadrilaterals $P_3P_4P_{10}P_{11}$ and $P_{12}P_{13}$ $P_{16}P_{18}$ are split into pairs of triangles in different ways.) This is hardly surprising, but we did often find identical final triangulations, in practice.

Figure (viii).



Figures (ix) and (x) show the triangulations obtained with Programs B and C, respectively, for the double square spirals with $i = 2$ (i.e., 16 vertices). This is an example of the situation mentioned above, in which the triangulations coincide; though the order in which the triads are removed differs.

Such experimental results would, in themselves, convince most

Figure (ix).

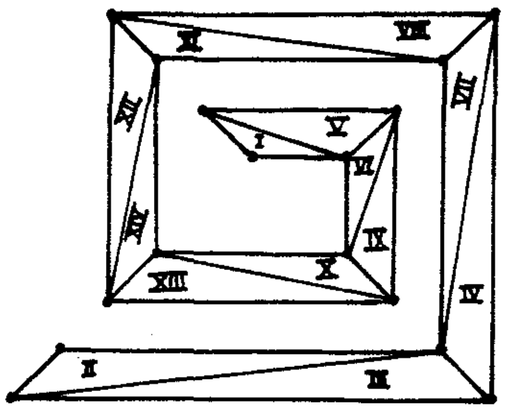
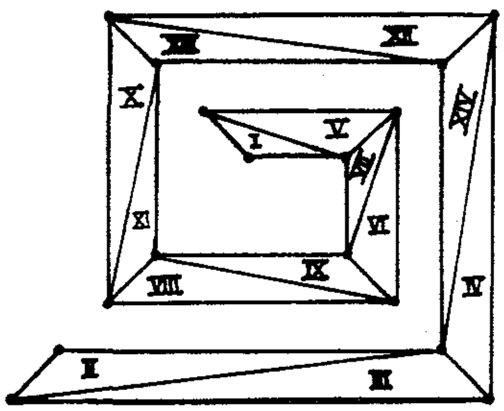


Figure (x).



users to choose the new algorithm, based on Propositions 9 and 10, over the old one, based on Algorithm 3 of [1]. We shall see later that the theoretical worst-case bound on the a.o. count of the new algorithm is considerably better than that for the old one.

In the listings below, output functions (which are dependent on the purpose and context of the program, and will therefore vary) are omitted. All the programs were written in standard "C" with preprocessor calls. They were run under Unix System V (version 4.2 by Callan Data Systems) on a C.D.S. Unistar 100/200 workstation with a Motorola 68000 c.p.u.

4. Program A

The program opens with preliminary definitions.

```
#include <stdio.h>

#define MAX 100                     MAX = greatest allowed number of vertices

int g,                              g = gamma (discriminant) call count
        n,                          n = number of vertices in polygon
            p,                      p = number of convex vertices (in A-list)
                q,                  q = number of re-entrant vertices (B-list)
                    r,              r = removed-triad index
                        C[3][MAX];  Removed triads are stored in C

float P[2][MAX],                    P[0] = x, P[1] = y, for polygon vertices
                    discr[MAX];     discr[i] is discriminant for vertex P[i+1]
```

gamma(h, i, j) increments the gamma-count g and returns the value of the discriminant $\gamma(h, i, j)$, as defined in (3).

```
#define gamma(h, i, j) (g++, P[0][h] * (P[1][i] - P[1][j]) \
                             - P[1][h] * (P[0][i] - P[0][j]) \
                             + P[0][i] * P[1][j] - P[1][i] * P[0][j])
```

Each u-list has its identifying pointer in the "up" component of a corresponding D-cell (see below); this points to the first cell of the u-list (a dummy cell of the form {ul, 0, 0}, with "ul" a u-list-pointer). Each u-cell thereafter takes the form {ul, uSt, udex}, where "ul" points to the next u-cell, "uSt" points to the predecessor of the cell in the t-list S[udex] (see below) which itself points back to the predecessor of the current u-cell, and the index "udex" identifies a vertex P(udex+1) of the polygon, contained inside the convex triad to which the D-cell (whose "up" component points to the current u-list) refers. The list is therefore absent if "up" = 0, and effectively empty if up → ul = 0.

```
struct u_cell { struct u_cell *ul;
                struct t_cell *uSt;
                int udex;
              } ;
```

The t-lists have identifying pointers S[k], pointing to header-cells head-t = {tf, ts}, with "tf" pointing to the first and "ts" to the last t-cell. The first t-cell is a dummy cell of the form {tl, 0}, where "tl" is a t-list-pointer. Every t-cell thereafter takes the form {tl, tu}, where "tl" points to the next t-cell and "tu" points to a u-cell, which is the predecessor of a u-cell whose index "udex" is k, the index of the t-list S[k] (see above).

```
struct t_cell { struct t_cell *tl;
                struct u_cell *tu;
              } ;

struct head_t { struct t_cell *tf, *ts; } *S[MAX];
```

The D-list initially has an identifying pointer D, pointing to the first D-cell. Each D-cell = {pp, np, f, b, up, index}, where "pp" is a D-list-pointer, "np" is a reverse-sense D-list-pointer, "f" and "b" are other pointers to D-cells (see below), "up" is the identifying pointer to a u-list (see above), and "index" is the index of the vertex P(index+1) of the polygon, to which the D-cell refers.

The D-list incorporates two other lists, the A-list and the B-list. None of these three lists have header-cells. The identifying pointer of the A-list (which points directly to the first D-cell in the A-list) is A, and that of the B-list (which points to the first D-cell in the B-list) is B; the pointers X, Y, and Z respectively point to the last D-cells of the A-, B-, and D-lists. The D-cells in the A-list are those referring to convex vertices; the D-cells in the B-list are those referring to re-entrant vertices. The "f" and "b" pointers are respectively forward and backward list-pointers for D-cells of like kind (i.e., both in the A-list or both in the B-list).

When the construction of the A-, B-, and D-lists is completed, the list-pointers of the last cells (i.e., Z → pp, X → f, and Y → f) are made to point to the first cells in their respective lists, and the backward-list-pointers of the first cells (i.e., D → np, A → b, and B → b) are made to point to the last cells in their respective lists; making these lists circular. Thereafter, only the pointers A and B are maintained; since D, X, Y, and Z are no longer needed.

```
struct D_cell { struct D_cell *pp, *np, *f, *b;
                struct u_cell *up;
                int index;
              } *A, *B, *D, *X, *Y, *Z;
```

NEW-u, NEW-t, NEW-Ht, and NEW-D respectively allocate, using the function malloc(), and return pointers to free memory space for new u-cells, t-cells, t-header-cells, and D-cells, allowing us to construct the needed lists, cell-by-cell.

```
char *malloc();

#define NEW_u   (struct u_cell *) malloc(sizeof(struct u_cell))

#define NEW_t   (struct t_cell *) malloc(sizeof(struct t_cell))

#define NEW_Ht   (struct head_t *) malloc(sizeof(struct head_t))

#define NEW_D   (struct D_cell *) malloc(sizeof(struct D_cell))
```

ins-u(u, k) appends, to the end of the t-list S[k], a new t-cell {0, u}, and inserts, after the u-cell pointed to by the pointer u, a new u-cell {ul, S[k] → ts, k} (S[k] → ts, pointing to the old last t-cell is then updated to the new cell).

```
ins_u(u, k)

  struct u_cell *u;                 Pointer to predecessor u-cell
  int k;                            Index of included vertex P(k+1)

  { struct u_cell *v;
    struct t_cell *t;

    t = S[k] -> ts -> tl = NEW_t;   t points to new t-cell; as does last t-cell
    t -> tl = 0;                    New last t-cell points nowhere
    t -> tu = u;                    Last t-cell points to given u-cell u
    v = NEW_u;                      v points to new u-cell
    v -> udex = k;                  Index of new u-cell is k (given)
    v -> uSt = S[k] -> ts;          New u-cell points back to old last t-cell
    v -> ul = u -> ul;              New u-cell points to successor cell of u
    u -> ul = v;                    Predecessor cell points to new u-cell
    S[k] -> ts = t;                 S[k] → ts points to new last t-cell
  }
```

del-u(d) deletes, from all t-lists, cells pointing back to u-cells in the u-list pointed to by d → up, using the "uSt" pointers; then voids d → up.

```
del_u(d)

  struct D_cell *d;                 Pointer to D-cell whose u-list is removed

  { struct u_cell *u;
```

```
        if ((u = d -> up) != 0)                       Do nothing if u-list is absent
          { while ((u = u -> ul) != 0)                For every cell in u-list:
              u -> uSt -> tl = u -> uSt -> tl -> tl;    remove corresponding t-cell
                                                        from its t-list
            d -> up = 0;                      Finally, annul the u-list pointer
          }
      }
```

del-S(i) deletes all cells referring to the vertex P(i+1) from all u-lists, using the listing of their predecessors in S[i]; then voids S[i].

```
    del_S(i)

      int i;

      { struct t_cell *t;

        if (S[i] != 0)                                Do nothing if t-list is absent
          { t = S[i] -> tf;                           t points to first cell of t-list
            while ((t = t -> tl) != 0)                For every cell in t-list:
              t -> tu -> ul = t -> tu -> ul -> ul;      remove corresponding u-cell
                                                        from its u-list
            S[i] = 0;                         Finally, annul the t-list pointer
          }
      }
```

fill-D(j, x) appends a D-cell {0, np, 0, b, 0, j}, with null forward pointers, to the D-list, unless x = 0, and adds it to the A-list, if x > 0, or to the B-list, if x < 0. In application, x = discr[j] is the discriminant associated with the vertex P(j+1), and this is zero for redundant (collinear) vertices, positive for convex vertices, and negative for re-entrant vertices.

```
    fill_D(j, x)

      int j;
      float x;

      { struct D_cell *d;

        if (x != 0)                   Omit redundant (collinear) vertices
          { d = NEW_D;                d points to new D-cell
            d -> pp = d -> f = 0;     New cell points nowhere (forward)
            d -> up = 0;              New cell has no u-list
            d -> index = j;           Index of new cell is j (given)
            if (Z != 0)               If this is not the first D-cell:
              { Z -> pp = d;            old last cell points to new one
                d -> np = Z;            new cell points back to old last one
              }
            else                      If this is the first D-cell:
              { d -> np = 0;            new cell points back nowhere
                D = d;                  D points to first D-cell
              }
            Z = d;                    Z points to (new) last D-cell
          }
        if (x > 0)                          If vertex is convex:
          { if (X != 0)                       if this is not the first A-cell:
              { X -> f = d;                      old last cell points to new one
                d -> b = X;                      new cell points back to old last one
              }
            else                              if this is the first A-cell:
              { d -> b = 0;                      new cell points back nowhere
                A = d;                           A points to first A-cell
              }
            X = d;                            X points to (new) last A-cell
          }

        if (x < 0)                          If vertex is re-entrant:
          { if (Y != 0)                       if this is not the first B-cell:
              { Y -> f = d;                      old last cell points to new one
                d -> b = Y;                      new cell points back to old last one
              }
            else                              if this is the first B-cell:
              { d -> b = 0;                      new cell points back nowhere
                B = d;                           B points to first B-cell
              }
            Y = d;                            Y points to (new) last B-cell
          }
      }
```

adjust(c, d, e, x) makes the necessary adjustments when a vertex which was re-entrant becomes convex or collinear. Pointers c and e point to consecutive convex vertices with the vertex pointed to by d, originally re-entrant, between them. The new value of the discriminant of d is x.

```
    adjust(c, d, e, x)

      float(x);
      struct D_cell *c, *d, *e;

      { q--;                                Decrement the re-entrant vertex count q

        if (d -> f != d)                    If d was not the last re-entrant vertex:
          { d -> f -> b = d -> b;              eliminate d from B-list, in both
            d -> b -> f = d -> f;                                     directions
          }

        if (x == 0)                         If d is now redundant (collinear):
          { d -> pp -> np = d -> np;           eliminate d from D-list, in both
            d -> np -> pp = d -> pp;                                  directions
            if (d -> np == c) find_u(c);       if c is predecessor of d:
                                                  construct new u-list for c
            if (d -> pp == e) find_u(e);       if e is successor of d:
                                                  construct new u-list for e
            del_S(d -> index);                 delete references to d in all u-lists
          }
```

```
       else                              If d is now convex:
         { p++;                             increment convex vertex count p
           d -> b = c;                      insert d into A-list,
           c -> f = d;                         between c and e,
           d -> f = e;                         in both directions
           e -> b = d;
           find_u(d);                       construct new u-list for d
         }
     }
```

find-u(d) constructs a u-list for the vertex whose corresponding D-cell is pointed to by the pointer d. d must point to a convex vertex.

```
   struct D_cell *find_u(d)

     struct D_cell *d;

     { int h, i, j, k, mt;
       struct u_cell *u;
       struct D_cell *c;

       del_u(d);                         Delete any previous u-list for d

       u = d -> up = NEW_u;              u and d → up point to new u-cell
       u -> ul = 0;                      First u-cell is dummy,
       u -> uSt = 0;                        with uSt null
       u -> udex = 0;                       and udex = 0

       h = d -> np -> index;             h = index of predecessor of d
       i = d -> index;                   i = index of d
       j = d -> pp -> index;             j = index of successor of d

       mt = 0;                           mt = "empty triad" flag; initially zero,
                                                                meaning "empty"
       c = B;                            Starting with B,
       if (c != 0)                          if B-list not empty
         do                              For every B-cell:  c points to cell
           { k = c -> index;                k = index of c

             if (k != h && k != i && k != j)      if k is not h, i, or j,
               if (gamma(i, j, k) >= 0)           and if k
                 if (gamma(j, h, k) >= 0)            is inside or on
                   if (gamma(h, i, k) >= 0)             triad γ(h, i, j):
                     { mt = 1;                       mt = 1, meaning "non-empty"
                       ins_u(u, k);                  insert cell referring to k,
                       u = u -> ul;                     after u in u-list, and
                     }                                  advance u to new cell
             c = c -> f;                    go to next B-cell
           }
         while (c != B) ;                   until cycle back to B

       if (mt == 1)                      If the triad is not empty:
         { c = A;                           starting with A, for every A-cell:
           do
```

```
163 :               { k = c -> index;                              Do the same for
164 :                 if (k != h && k != i && k != j)              the A-list as
165 :                   if (gamma(i, j, k) >= 0)                   was done before
166 :                     if (gamma(j, h, k) >= 0)                 for the B-list,
167 :                       if (gamma(h, i, k) >= 0)               seeking included
168 :                         { ins_u(u, k);                       vertices and
169 :                           u = u -> ul;                       adding them to
170 :                         }                                    the u-list
171 :                 c = c -> f;
172 :               }
173 :             while (c != A) ;
174 :           }

175 :         if (mt == 0) return(d);          If the triad is empty, return pointer d
176 :         else         return(Z);          If not, return the pointer Z (initially 0)
177 :       }                                     [Above used only in line 224 to find
                                                    the first empty convex triad.]
```

THE MAIN PROGRAM. After reading-in the polygon (number n of vertices first; then the coordinate-pairs of the vertices, in cyclic order, interior-on-the-left or reverse sense), the program constructs the D-, A-, B-, and all u- and t-lists; and then executes the algorithm, finding and removing to the C-array successive empty convex triads.

```
178 :     main()
179 :       { int h, i, j, k;
180 :         float x, y;
181 :         struct D_cell *c, *d, *find_u();
182 :         struct t_cell *t;

183 :         do scanf("%d ", &n); while (n < 3);        Input n = number of vertices
184 :         for (i = 0; i < n; i++)                       and coordinates (x, y)
185 :           { scanf("%f %f ", &x, &y);                     of each vertex into
186 :             P[0][i] = x;                                     P[0][...] and
187 :             P[1][i] = y;                                     P[1][...]
188 :           }

189 :         g = p = q = r = 0;                  Initialize all counts to zero

190 :         h = 0; x = P[0][0];                 Find vertex P(h+1) with greatest
191 :         for (i = 0; i < n; i++)                x-coordinate x = P[0][h], and if
192 :           { if (P[0][i] > x)                   several such, that with greatest
193 :               { h = i;                         y-coordinate P[1][h]
194 :                 x = P[0][i];
195 :               }
196 :             if (P[0][i] == x && P[1][i] > P[1][h]) h = i;

197 :             if (i == 0)             discr[i] = gamma(n - 1, 0, 1);
198 :             else if (i == n - 1)    discr[i] = gamma(n - 2, n - 1, 0);
199 :             else                    discr[i] = gamma(i - 1, i, i + 1);

200 :             C[0][i] = C[1][i] = C[2][i] = 0;   Initialize (empty) C-array
```

```
          S[i] = NEW_Ht;                        S[i] points to new t-header cell
          t = S[i] -> tf = S[i] -> ts = NEW_t;  t, S[i] → tf, and S[i] → ts point
          t -> tl = 0;                             to new t-cell, initiated with
          t -> tu = 0;                             null pointers forward
        }

      x = ((discr[h] > 0) ? 1 : (-1));   Extreme vertex P(h+1) must be convex;
      for (i = 0; i < n; i++)               so its discriminant should be positive.
        { discr[i] = x * discr[i];          Adjust all discriminant values to be
          if (discr[i] > 0) p++;            of appropriate signs (for interior-
          if (discr[i] < 0) q++;            on-the-left sense of traversal)
        }

      A = X = B = Y = D = Z = 0;       Initialize all list-pointers to be null

      if (x > 0) for (i = 0;     i < n;   i++) fill_D(i, discr[i]);  Construct
      else       for (i = n - 1; i > - 1; i--) fill_D(i, discr[i]);  the D-, A-,
                                                                      and B-lists
      Z -> pp = D;                     Connect beginnings and ends
      D -> np = Z;                        of D-, A-, and B-lists,
      X -> f = A;                         so as to make each list
      A -> b = X;                         circular, in both
      Y -> f = B;                         directions
      B -> b = Y;

      d = X;                           Beginning with the last A-cell and cycling
      Z = 0;                              backwards through the A-list:
      do
        { Z = find_u(d);                  construct the u-list for each convex
          d = d -> b;                        vertex
        }
      while (d != X) ;                    (end-up with the Z pointing to the
                                              first empty convex triad)
      while (p > 2)                    So long as three convex vertices remain:
        { X = Z -> np;                    X points to predecessor of Z in D-list
          Y = Z -> pp;                    Y points to successor of Z in D-list
          X -> pp = Y;                    remove Z from D-list, in both
          Y -> np = X;                                           directions

          C[0][r] = h = X -> index;       put the triad δ(h, i, j) into the
          C[1][r] = i = Z -> index;          C-array
          C[2][r] = j = Y -> index;
          r++;                            increment the removed-triad count

                                          decrement the convex vertex count
          p--;                            delete all u-cells referring to P(i+1)
          del_S(i);                       delete all t-cells pointing back to
          del_u(Z);                          the u-list of Z (i.e., of P(i+1)).

          c = Z -> f -> b = Z -> b;       delete Z from the A-list, and let c and
          d = Z -> b -> f = Z -> f;          be previous and next convex vertices
          if (A == Z) A = Z -> f;         if A = Z, move A forward
          discr[i] = 0;
```

```
244 :           if (discr[h] < 0)             if X was re-entrant:
245 :             { x = discr[h] = gamma(X -> np -> index, h, j);  x = new discriminant
246 :               if (x >= 0)                 if X changes to collinear or convex:
247 :                 { if (X -> f == X) B = 0;   if X was last re-entrant, annul B
248 :                   if (B == X) B = B -> b;   if B = X, move B back
249 :                   adjust(c, X, d, x);
250 :                   if (x > 0) c = X;         if X is now convex, move c to X
251 :                 }
252 :             }
253 :           else find_u(X);               if X was convex, recompute its u-list

254 :           if (discr[j] < 0)             if Y was re-entrant:
255 :             { x = discr[j] = gamma(Y -> pp -> index, h, j);  x = new discriminant
256 :               if (x >= 0)                 if Y changes to collinear or convex:
257 :                 { if (Y -> f == Y) B = 0;   if Y was last re-entrant, annul B
258 :                   if (B == Y) B = B -> b;   if B = Y, move B back
259 :                   adjust(c, Y, d, x);
260 :                   if (x > 0) d = Y;         if Y is now convex, move d to Y
261 :                 }
262 :             }
263 :           else find_u(Y);               if Y was convex, recompute its u-list

264 :           h = 1;                    h = "empty triad" flag; initially 1, meaning
265 :           while (h == 1)              non-empty.  Begin with d = next convex vertex
266 :             { if (d -> up -> ul == 0)   if u-list of d is empty:
267 :                 { h = 0;                  h = 0 (meaning empty triad found)
268 :                   Z = d;                  Z points to first empty triad found,
269 :                 }                           after that just removed
270 :               else d = d -> f;            if u-list of d is not empty,
271 :             }                             advance d to next convex vertex
272 :         }
273 :     }
```

5. Program B

Program B was a modification of Program A to generate members of the family of "double square spiral" polygons, on input of only the parameter i (see Figures (ix) and (x)). In the listing below, only the changes from Program A are noted (all line numbers refer to Program A).

B 1– 21: *Lines **1–21** are unchanged. Lines **22–25** are replaced by*

```
B 22:    struct D_cell { struct D_cell *pp, *np, *f, *b;
B 23:                    struct u_cell *up;
B 24:                    int index;
B 25:                  } *A, *B, *Z;
```

B 26– 64: *removing pointers D, X, and Y from global to main() declaration. Lines **26–64***
B 65–135: *and **107–177** are unchanged. The fill-D(j, x) function is omitted (it is incorporated in main(): see below), leaving out lines **65–106**. The declarations in lines **178–182** become (note line **181**)*

```
B136:      main()
B137:        { int h, i, j, k;
B138:          float x;
B139:          struct D_cell *c, *d, *D, *X, *Y, *find_u();
B140:          struct t_cell *t;
```

Lines 183–188 (polygon input) are replaced by

```
B141:          scanf("%d %d", &i, &prfl);
B142:          n = 8 * i;
B143:          for (h = 0; h < i; h++)
B144:            { P[0][    4 * h    ] = - 2 * h    ; P[1][    4 * h    ] =   2 * h    ;
B145:              P[0][n - 4 * h - 1] = - 2 * h - 1; P[1][n - 4 * h - 1] =   2 * h + 1;
B146:              P[0][    4 * h + 1] =   2 * h + 2; P[1][    4 * h + 1] =   2 * h    ;
B147:              P[0][n - 4 * h - 2] =   2 * h + 3; P[1][n - 4 * h - 2] =   2 * h + 1;
B148:              P[0][    4 * h + 2] =   2 * h + 2; P[1][    4 * h + 2] = - 2 * h - 2;
B149:              P[0][n - 4 * h - 3] =   2 * h + 3; P[1][n - 4 * h - 3] = - 2 * h - 3;
B150:              P[0][    4 * h + 3] = - 2 * h - 2; P[1][    4 * h + 3] = - 2 * h - 2;
B151:              P[0][n - 4 * h - 4] = - 2 * h - 3; P[1][n - 4 * h - 4] = - 2 * h - 3;
B152:            }
```

B153: *Line 189 initializes all counts to zero as in Program A. Since all the spiral polygons are traversed in the accepted interior-on-the-left sense, the discriminant calculation and the generation of the D-, A-, and B-lists are slightly simplified. Lines 190–199 become*

```
B154:          for (i = 0; i < n; i++)
B155:            { if (i == 0)            discr[i] = gamma(n - 1, 0, 1);
B156:              else if (i == n - 1) discr[i] = gamma(n - 2, n - 1, 0);
B157:              else                   discr[i] = gamma(i - 1, i, i + 1);
```

B158–162: *lines 200–204 are unchanged, and lines 205–211 become*

```
B163:              if (discr[i] > 0) p++;
B164:              if (discr[i] < 0) q++;
B165:            }
```

B166: *Line 212 initializes all list-pointers to be null as in Program A. Then lines 213 and 214 are replaced by the inclusion of the equivalent of fill-D(j, x), as follows*

```
B167:          for (i = 0; i < n; i++)
B168:            { if (discr[i] != 0)
```

B169–182: *then lines 70–83, then*

```
B183:              if (discr[i] > 0)
```

B184–193: *then lines 85–94, then*

```
B194:              if (discr[i] < 0)
```

B195–204: *then lines 96–105, then*

```
B205:            }
```

B206–264: *The remainder of the program, namely, lines 215–273, are then unchanged; since the algorithm is the same.*

6. Program C

This program again, like Program B, generates members of the family of double square spirals (see Figures (ix) and (x)), on input of the parameter i = 1, 2, 3, ..., 12; but it uses the new algorithm, based on Propositions 9 and 10, instead of the old one. In the listing below, comparison with the previous programs is stressed.

```
C  1 :   #include <stdio.h>
C  2 :   #define MAX 100
```

*Like lines **A 1, 2** and **B 1, 2***

```
C  3 :   int g,
C  4 :          n,
C  5 :             r,
C  6 :                          C[3][MAX];
```

*Unlike lines **A 3–8** and **B 3–8** : vertex counts p and q are removed from global to main()*

```
C  7 :   float P[2][MAX],
C  8 :                         discr[MAX];
```

*Like lines **A 9–13** and **B 9–13***

```
C  9 :   #define gamma(h, i, j) (g++, P[0][h] * (P[1][i] - P[1][j]) \
C 10 :                                - P[1][h] * (P[0][i] - P[0][j]) \
C 11 :                                + P[0][i] * P[1][j] - P[1][i] * P[0][j])
```

*We omit all reference to u- and t-lists; so lines **A 14–21** and **B 14–21** are left out of the program.*

```
C 12 :   struct D_cell { struct D_cell *pp, *np, *f, *b;
C 13 :                   int index;
C 14 :                 } *A, *B;
```

*Compare lines **A 22–25** and **B 22–25** : only A and B are global, and no u-list (up)*

```
C 15 :   char *malloc();
```

*Like lines **A 26** and **B 26***

*Omit NEW-u, NEW-t, NEW-Ht (lines **A 27–29** and **B 27–29**)*

```
C 16 :   #define NEW_D   (struct D_cell *) malloc(sizeof(struct D_cell))
```

*Like lines **A 30** and **B 30***

*Omit ins-u(), del-u(), del-S() (lines **A 31–64** and **B 31–64**) and fill-D(), which is incorporated in main() later (lines A 65–106).*

The adjust() function below is essentially the same as those in Programs A and B, except for the absence of reference to u- and t-lists, and no updating of the convex and re-entrant vertex counts p and q.

```
C 17 :    adjust(c, d, e, x)

C 18 :      float x;
C 19 :      struct D_cell  *c, *d, *e;

C 20 :      { if (d -> f != d)
C 21 :          { d -> f -> b = d -> b;
C 22 :            d -> b -> f = d -> f;
C 23 :          }
C 24 :        if (x == 0)                     Compare lines A 107–130 and B 65–88 :
C 25 :          { d -> pp -> np = d -> np;       same, except for omission of
C 26 :            d -> np -> pp = d -> pp;       find-u() and del-S() calls:  no
C 27 :          }                                u- and t-lists; also, no p and
C 28 :        else                               q count updates
C 29 :          { d -> b = c;
C 30 :            c -> f = d;
C 31 :            d -> f = e;
C 32 :            e -> b = d;
C 33 :          }
C 34 :      }
```

*Omit function find-u() (lines **A 131–177** and **B 89–135**).*

```
C 35 :    main()
C 36 :      { int h, i, j, k, p, q;                   Compare lines A 178–182
C 37 :        struct D_cell *AA, *BB, *D, *DD, *d;      and B 136–140

C 38 :        scanf("%d", &i);
C 39 :        n = 8 * i;                          Like lines B 141–152
C 40 :        for (h = 0; h < i; h++)
C 41 :          { P[0][    4 * h    ] = - 2 * h    ; P[1][    4 * h    ] =   2 * h    ;
C 42 :            P[0][n - 4 * h - 1] = - 2 * h - 1; P[1][n - 4 * h - 1] =   2 * h + 1;
C 43 :            P[0][    4 * h + 1] =   2 * h + 2; P[1][    4 * h + 1] =   2 * h    ;
C 44 :            P[0][n - 4 * h - 2] =   2 * h + 3; P[1][n - 4 * h - 2] =   2 * h + 1;
C 45 :            P[0][    4 * h + 2] =   2 * h + 2; P[1][    4 * h + 2] = - 2 * h - 2;
C 46 :            P[0][n - 4 * h - 3] =   2 * h + 3; P[1][n - 4 * h - 3] = - 2 * h - 3;
C 47 :            P[0][    4 * h + 3] = - 2 * h - 2; P[1][    4 * h + 3] = - 2 * h - 2;
C 48 :            P[0][n - 4 * h - 4] = - 2 * h - 3; P[1][n - 4 * h - 4] = - 2 * h - 3;
C 49 :          }

C 50 :        g = p = q = r = 0;                  Like lines A 189  and B 153

C 51 :        for (i = 0; i < n; i++)             Like lines B 154–157
C 52 :          { if (i == 0)              discr[i] = gamma(n - 1, 0, 1);
C 53 :            else if (i == n - 1) discr[i] = gamma(n - 2, n - 1, 0);      Compare
C 54 :            else                   discr[i] = gamma(i - 1, i, i + 1);      lines
C 55 :            if (discr[i] > 0) p++;               Like lines B 158            A 190
C 56 :            if (discr[i] < 0) q++;               and B 163–165               -211
C 57 :            C[0][i] = C[1][i] = C[2][i] = 0;
C 58 :          }

C 59 :        A = AA = B = BB = D = DD = 0;      Compare lines A 212  and B 166
C 60 :          for (i = 0; i < n; i++)
C 61 :            { if (discr[i] != 0)
C 62 :                { d = NEW_D;
C 63 :                  d -> pp = d -> f = 0;
C 64 :                  d -> index = i;
C 65 :                  if (DD != 0)
C 66 :                    { DD -> pp = d;
C 67 :                      d -> np = DD;
C 68 :                    }
C 69 :                  else
C 70 :                    { d -> np = 0;
C 71 :                      D = d;
C 72 :                    }
C 73 :                  DD = d;
C 74 :                }
C 75 :              if (discr[i] > 0)
C 76 :                { if (AA != 0)
C 77 :                    { AA -> f = d;
C 78 :                      d -> b = AA;
C 79 :                    }
C 80 :                  else
C 81 :                    { d -> b = 0;
C 82 :                      A = d;
C 83 :                    }
C 84 :                  AA = d;
C 85 :                }
C 86 :              if (discr[i] < 0)
C 87 :                { if (BB != 0)
C 88 :                    { BB -> f = d;
C 89 :                      d -> b = BB;
C 90 :                    }
C 91 :                  else
C 92 :                    { d -> b = 0;
C 93 :                      B = d;
C 94 :                    }
C 95 :                  BB = d;
C 96 :                }
C 97 :            }
C 98 :          DD -> pp = D;
C 99 :          D -> np = DD;
C100 :          AA -> f = A;
C101 :          A -> b = AA;
C102 :          BB -> f = B;
C103 :          B -> b = BB;
C104 :          split(A, B);
C105 :        }
```

Compare lines ***B 167–205*** *and the fill-D() function in lines* ***A 65–106 :*** *here, AA, BB, and DD play the parts of X, Y, and Z there (also, lines* ***A 72*** *and* ***B 174*** *are not needed, since the D-cells have no up u-list pointer*

Compare lines ***A 215–220*** *and* ***B 206–211***

CALL THE RECURSIVE SPLIT() ROUTINE

We now come to the crucial SPLIT() routine, which recursively calls itself until the triangulation is completed.

The call is to split(Q, R), where Q points to the current A-list (convex vertices) and R points to the current B-list (re-entrant vertices). Taking the convex triad $\Delta_i = \delta(h, i, j) = P(h+1)P(i+1)P(j+1)$, where Q points to the D-cell corresponding to $P(i+1)$; the routine searches the B-list for any re-entrant vertices in or on the triad, and among these, for the last one with maximal $\gamma[j, h, k]$ (see Figure (v)). If the triad is empty, the triad is removed to the C-array and the routine recurs on the reduced polygon. If not, the segment $P(i+1)P(k+1)$ defined by the pointers Q and Z (Z points to the selected re-entrant vertex, which is $P(k+1)$) is used to split the polygon into two contiguous and smaller simple closed polygons, and the routine recurs to each of them in turn.

```
C106 :    split(Q, R)

C107 :      struct D_cell *Q, *R;           Q points to current A-list, R to B-list

C108 :      { int h, i, j, k;
C109 :        float x, y;
C110 :        struct D_cell *c, *d, *Qnp, *Qb, *Qf, *W, *X, *Y, *Z, *Zpp, *Zb, *Zf;

C111 :        if (Q -> f -> f == Q) return;    Stop when only two convex vertices
                                                                        remain
C112 :        h = Q -> np -> index;     h = index of predecessor of Q
C113 :        i = Q -> index;           i = index of (convex vertex) Q
C114 :        j = Q -> pp -> index;     j = index of successor of Q

C115 :        y = 0;                                           }
C116 :        Z = Q;                                           }  Compare lines A 144-159 .
C117 :        if (R != 0)                                      }
C118 :          { d = R;                                       }  Initially, Z = Q (convex);
C119 :            do                                           }  if any re-entrant vertex
C120 :              { k = d -> index;                          }  (pointer d, index k) lies
C121 :                if (k != h && k != i && k != j)          }  in or on the triad δ(h,
C122 :                  if (gamma(h, i, k) >= 0)               }  i, j), we put Z = d for
C123 :                    if (gamma(i, j, k) >= 0)             }  the last included re-
C124 :                      if ((x = gamma(j, h, k)) >= y)     }  entrant vertex with
C125 :                        { y = x;                         }  maximal γ[j, h, k]
C126 :                          Z = d;                         }  (See Propositions 9 and
C127 :                        }                                }  10 in Section 2.)
C128 :                d = d -> f;                              }
C129 :              }                                          }
C130 :            while (d != R) ;                             }
C131 :          }                                              }
```

When the triad is empty, we treat is much as before.

```
C132 :        if (Z == Q)                 If the triad is empty:  remove it and recur:
C133 :          { C[0][r] = h;          }
C134 :            C[1][r] = i;          }
C135 :            C[2][r] = j;          }
C136 :            r++;                  }
C137 :            X = Q -> np;          }   Compare lines A229-236, 240, 241, 243
C138 :            Y = Q -> pp;          }
C139 :            X -> pp = Y;          }
C140 :            Y -> np = X;          }
C141 :            c = Q -> f -> b = Q -> b;
C142 :            d = Q -> b -> f = Q -> f;
C143 :            discr[i] = 0;         }

C144 :            W = R;                 Pointer to B-list

C145 :            if (discr[h] < 0)                                  }
C146 :              { x = discr[h] = gamma(X -> np -> index, h, j);  }
C147 :                if (x >= 0)                                    }
C148 :                  { if (X -> f == X) W = 0;                    }  Like lines
C149 :                    if (W == X) W = W -> b;                    }  A244-252 ,
C150 :                    adjust(c, X, d, x);                        }  with W for B
C151 :                    if (x > 0) c = X;                          }
C152 :                  }                                            }
C153 :              }                                                }

C154 :            if (discr[j] < 0)                                  }
C155 :              { x = discr[j] = gamma(Y -> pp -> index, h, j);  }
C156 :                if (x >= 0)                                    }
C157 :                  { if (Y -> f == Y) W = 0;                    }  Like lines
C158 :                    if (W == Y) W = W -> f;                    }  A254-262 ,
C159 :                    adjust(c, Y, d, x);                        }  with W for B
C160 :                    if (x > 0) d = Y;                          }
C161 :                  }                                            }
C162 :              }                                                }

C163 :            split(d, W);           Recur to split() routine
C164 :          }
```

When the triad is not empty, the segment QZ is used to split the polygon.

We interrupt to explain the situation. Since they are so selected, we know that <u>*Q* points to</u> [we often corrupt the language and say that <u>*Q* "is"</u>] <u>a *convex* vertex</u>, and <u>*Z* is a *re-entrant* vertex</u>. By Propositions 9 and 10 [the indices are differently named: $P_{j-1}P_jP_{j+1}$ in Figure (v) becomes $P_{h+1}P_{i+1}P_{j+1}$ here (vertices are no longer consecutively numbered, because of previous removals) and P_h there becomes P_{k+1} here; and so *Q* points to P_{i+1} (or *P*(*i*+1)) and *Z* to P_{k+1} (or *P*(*k*+1))] we know that, if the triad is not empty, such a *Z* exists; and the segment *QZ* (more properly, $P_{i+1}P_{k+1}$) splits the polygon $\mathfrak{P}_0$ into two contiguous simple closed polygons $\mathfrak{P}_1$ and $\mathfrak{P}_2$, having only the segment QZ in common. Each will recur to the split() routine with its own A- and B-lists; which must be constructed appropriately. We note further that, in the two new polygons, Q will still be convex; but Z may become collinear or convex. Each of the new polygons will have at least 3 vertices, and, by Lemma 5, each will contain *at least one more convex vertex*, but not necessarily any additional re-entrant vertex, beside Q and Z. Thus, $Q \rightarrow f$ is (more properly, "points to a vertex") in $\mathfrak{P}_1$ and $Q \rightarrow b$ is in $\mathfrak{P}_2$; but $Z \rightarrow f$ and $Z \rightarrow b$ may be in either polygon, and may equal Z itself. Of course, $Q \rightarrow pp$ and $Z \rightarrow np$ will be in $\mathfrak{P}_1$, and $Q \rightarrow np$ and $Z \rightarrow pp$ will be in $\mathfrak{P}_2$.

In setting up $\mathfrak{P}_1$, we must store the pointers to be used in $\mathfrak{P}_2$. This is tabulated below. It is assumed that U denotes the first *re-entrant* vertex *after* Q (must exist in $\mathfrak{P}_1$; may be Z) and that V denotes the last *convex* vertex *before* Z (must exist in $\mathfrak{P}_1$; will be *after* Q); then $U \rightarrow b$ and $V \rightarrow f$ must be in $\mathfrak{P}_2$ (the former may be Z; the latter will be *before* Q).

POINTER	IN $\mathfrak{P}_0$	IN $\mathfrak{P}_1$	IN $\mathfrak{P}_2$
$Q \rightarrow pp$	$Q \rightarrow pp$	$Q \rightarrow pp$	Z
$Q \rightarrow np$	$Qnp = Q \rightarrow np$	Z	Qnp
$Z \rightarrow pp$	$Zpp = Z \rightarrow pp$	Q	Zpp
$Z \rightarrow np$	$Z \rightarrow np$	$Z \rightarrow np$	Q
$Q \rightarrow f$	$Q \rightarrow f$	$Q \rightarrow f$	$Qf = V \rightarrow f$
$Q \rightarrow b$	$Qb = Q \rightarrow b$	V	Qb
$\underline{Z \rightarrow f = U:}$			
$Z \rightarrow f$	$Z \rightarrow f$	U	$Zf = Z$
$Z \rightarrow b$	$Z \rightarrow b$	$Z \rightarrow b$	$Zb = Z$
$\underline{Z \rightarrow f \neq U:}$			
$Z \rightarrow f$	$Zf = Z \rightarrow f$	U	Zf
$Z \rightarrow b$	$Z \rightarrow b$	$\left\{\begin{array}{ll} \underline{U = Z:} & Z \\ \underline{U \neq Z:} & Z \rightarrow b \end{array}\right\}$	$Zb = U \rightarrow b$
$U \rightarrow b$	$U \rightarrow b$	Z	---
$V \rightarrow f$	$V \rightarrow f$	Q	---
$Qf \rightarrow b$	V	---	Q
$Zb \rightarrow f$	U	---	Z

The various situations are illustrated in the figures below.

Figure (xi). *D-lists.*

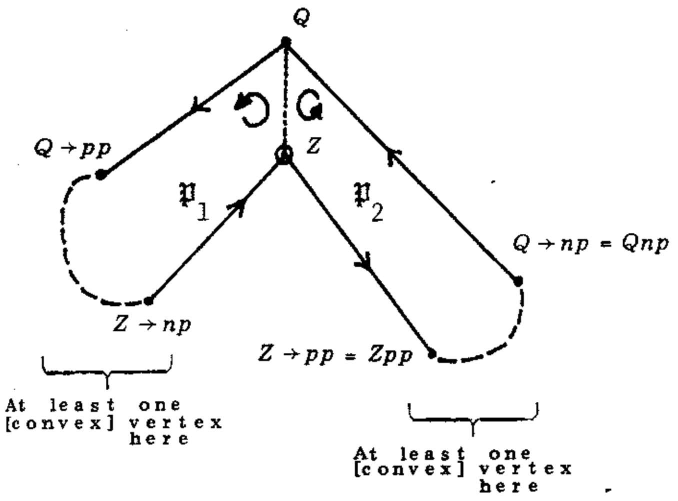


Figure (xii). *A-lists.*

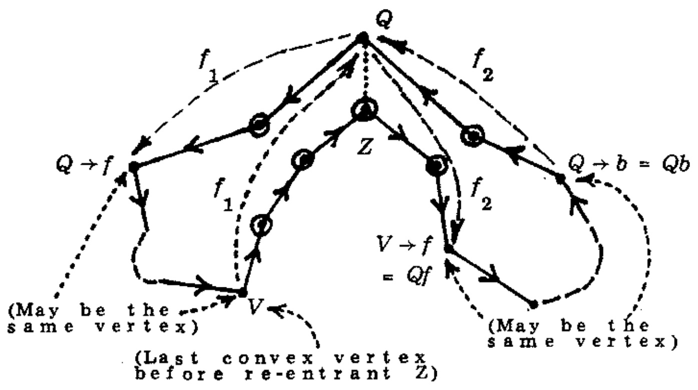


Figure (xiii). *B-lists.* *$U = Z \to f = Z$.*

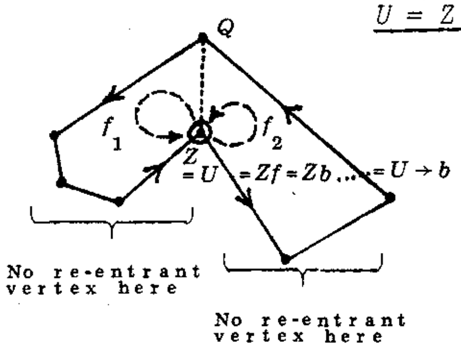

Q
f_1
f_2
$Z = U$
$= Zf = Zb \ldots = U \to b$
No re-entrant vertex here
No re-entrant vertex here

Figure (xiv). *B-lists. $U = Z \to f \neq Z$.*

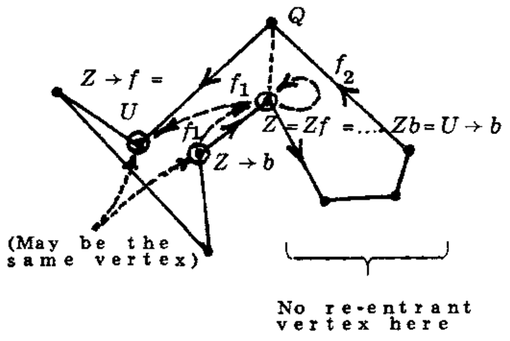


Figure (xv). *B-lists.* *$Z \to f \neq Z = U$.*

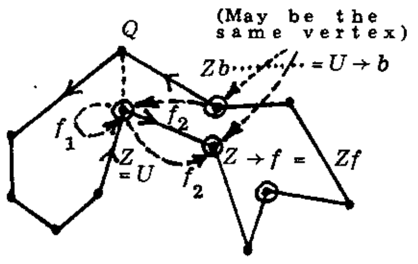
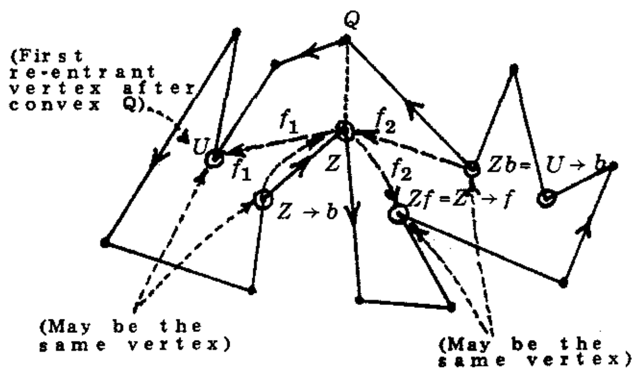


Figure (xvi). *B-lists. $Z \to f \neq Z \neq U$.*



Here, f_1 is the forward pointer relative to $\mathfrak{P}_1$, and f_2 that relative to $\mathfrak{P}_2$. The correspondence to the table is seen when it is observed that $Z \to f = Z$ only when there is only one re-entrant vertex and $U = Z$ also.

```
C165 :      else                          If the triad is not empty:  split it and recur:
C166 :        { k = Z -> index;             k = index of (re-entrant vertex) Z
C167 :          Qnp = Q -> np;              record Q → np as Qnp and
C168 :          Q -> np = Z;                replace with Z
C169 :          W = Z -> np;
C170 :          d = Z -> b;
C171 :          while (W == d)                 find last convex vertex ("V") before Z
C172 :            { W = W -> np;               and put it in W
C173 :              d = d -> b;
C174 :            }
C175 :          Qb = Q -> b;                record Q → b as Qb and
C176 :          Q -> b = W;                 replace with W (i.e. "V")
C177 :          Qf = W -> f;                record W → f as Qf and
C178 :          W -> f = Q;                 replace with Q
C179 :          Zpp = Z -> pp;              record Z → pp as Zpp and
C180 :          Z -> pp = Q;                replace with Q
C181 :          W = Q -> pp;
C182 :          d = Q -> f;
C183 :          while (W == d)                 find first re-entrant vertex ("U") after Q
C184 :            { W = W -> pp;               and put it in W
C185 :              d = d -> f;
C186 :            }
C187 :          if (Z -> f == W) Zf = Zb = Z;
C188 :          else
C189 :            { Zf = Z -> f;
C190 :              Zb = W -> b;                    (see tabulated relations)
C191 :              if (W == Z) Z -> b = Z;
C192 :            }
C193 :          Z -> f = W;
C194 :          W -> b = Z;

C195 :          x = discr[k] = gamma(Z -> np -> index, k, i);
C196 :          if (x >= 0)                    compare lines C146–152,
C197 :            { if (Z -> f == Z) W = 0;    with Z for X, Q → b for c,
C198 :              adjust(Q -> b, Z, Q, x);   and Q for d
C199 :            }

C200 :          split(Q -> f, W);     Recur to split() routine for first polygon
C201 :          Q -> np = Qnp;        }
C202 :          Q -> pp = Z;          }
C203 :          Q -> b = Qb;          }
C204 :          Q -> f = Qf;          }
C205 :          Qf -> b = Q;          }  restore all pointers for second polygon
C206 :          Z -> np = Q;          }     (see tabulated relations)
C207 :          Z -> pp = Zpp;        }
C208 :          Z -> b = Zb;          }
C209 :          Zb -> f = Z;          }
C210 :          W = Z -> f = Zf;      }  (note that W is reset)

C211 :          x = discr[k] = gamma(Z -> pp -> index, i, k);               }
C212 :          if (x >= 0)                      adjust for Z possibly       }
C213 :            { if (Z -> f == Z) W = 0;      becoming redundant or convex }
C214 :              adjust(Q, Z, Q -> f, x);       (see lines C195–199 )     }
C215 :            }                                                          }

C216 :          split(Q -> b, W);    Recur to split() routine for second polygon
C217 :        }
C218 :    }
```

7. Program D

This final program uses the new splitting algorithm; but, like Program A, applies it to an arbitrary polygon, whose vertices are input one-by-one.

D1–11: *Begin with lines* ***C1–11*** *. Replace lines* ***C12–14*** *with*

```
D12:    struct D_cell { struct D_cell *pp, *np, *f, *b;
D13:                    int index;
D14:                  } *A, *B, *D, *AA, *BB, *DD;
```

D15–58: *Lines* ***C15, C16*** *follow. Then comes fill-D(), from lines* ***A65–106*** *. Then*
D59–76: *we have adjust() from lines* ***C17–34*** *. The main() function begins with*

```
D77:    main()
D78:      { int h, i, j, k, p, q;
D79:        float x, y;
```

D80–104: *Then follow lines* ***A183–200*** *and* ***A206–211*** *;* ***C59*** *(instead of* ***A212*** *) and*
D105–227: ***A213, 214*** *. After this, we have* ***C98–218*** *, completing the program.*

8. Performance Bounds: Programs A and B

Since the form of input (and output) is irrelevant to our timing estimates, it follows that Programs A and B, and Programs C and D, may be treated as one.

We begin with Programs A and B. We first deal with the auxiliary functions. ins-u() takes time $O(1)$. del-u() deletes a u-list and all references to its cells in all t-lists. del-S() deletes a t-list and all references to its vertex in all u-lists. Let us write

$$m = p + q \qquad (11)$$

for the current number of vertices in process (i.e., in the D-list). Then each u-list or t-list has length not greater than $m = O(m)$, since a list never refers to the same vertex twice. Therefore the timing-bound for del-u() and del-S() is also $O(m)$. fill-D() appends one cell in time $O(1)$. find-u() calls del-u() once and then runs through the B-list and possibly the A-list, for a timing of $O(m)$. If we count arithmetic operations (a.o.) as the number of calls to gamma() times 9 [see lines A11-13; remaining arithmetic operations are a single multiplication in line A208, plus possible subtractions implied in all the tests used, certainly not disturbing the general behavior of the algorithm], we see that each call to find-u() involves at most $3(m - 3)$ gammas $= 27(m - 3)$ a.o. Thus, adjust(), involving as it does as many as two calls to find-u() and one to del-S(), has a timing of $O(m)$. Its gamma-count is a little complicated, in practice: if one takes the line in which the number m of vertices is reduced by one, then as many as two calls to find-u() lead to a maximum of $6(m - 4)$ gammas; but, if one takes the other line, in which m does not diminish, then only one call is made to find-u(), and the maximum is $3(m - 3)$.

We now come to the main program. After input, which we will not count, but which only takes time $O(n)$, in any case; we compute the n discriminants, for a time $O(n)$ with n gammas $= 9n$ a.o. (lines A190-211). The calls to fill-D() in lines A213 and A214 take time $O(n)$ too; and then the u-lists for all convex vertices are computed in time $O(pn) = O(n^2)$, with at most $3p(n - 3)$ gammas.

The triangulation loop (lines A228-272) remains. For each of at most $n - 2$ triads; we call del-S() and del-u(), and, for both of the two flanking vertices of the triad being removed to the C-array, we perform the operations of lines A244-263. The total time is clearly $O(n^2)$. Lastly, we seek the next empty convex triad, a process which takes time $O(p)$, for a total of $O(n^2)$. Thus, the entire program takes a time $O(n^2)$ to execute. This agrees with Theorem 4 of [1].

Turning to gamma-counts, we see that the preliminaries take at most $3n^2 - 8n$ gammas $= 9(3n^2 - 8n)$ a.o. In the triangulation loop, the only calls to gamma arise in lines A244-263. The situation in which the flanking vertex was re-entrant and does not remain so is clearly somewhat more laborious than those in which it was convex (call to find-u() only) or in which it was re-entrant and remains re-entrant (call to gamma() only). Here, there is a call to gamma() and then one to adjust(), involving at least one call to find-u(). Three cases arise: (i) no additional vertices eliminated: in the worst case, we call gamma() and find-u() twice each, for a gamma-count of $6m - 22$, emerging with $m - 1$ vertices. The total gamma-count under this regime is

$$(6n - 22) + (6n - 28) + (6n - 34) + (6n - 40) + \ldots \tag{12}$$

(ii) one additional vertex is eliminated: in the worst case, we call gamma() twice and find-u() thrice, for a gamma-count of at most $9m - 40$ (when the second flanking vertex is eliminated as collinear), emerging with $m - 2$ vertices. The total gamma-count now becomes

$$(9n - 40) + (9n - 58) + (9n - 76) + \ldots \tag{13}$$

Comparing the first two terms of (12) with the first term of (13), so as to arrive at the same situation, with $n - 2$ vertices; we see that $12n - 50 \geqslant 9n - 40$, so long as $3n \geqslant 10$ (i.e., $n \geqslant 4$). Thus, case (i) is more laborious than case (ii) (when $n = 3$, the count is zero, anyway). (iii) two additional vertices are eliminated: in the worst case, we call gamma() twice and find-u() four times, for a gamma-count of at most $12m - 64$, emerging with $m - 3$ vertices. The total gamma-count under this regime is then

$$(12n - 64) + (12n - 100) + (12n - 136) + \ldots \tag{14}$$

Comparing the first three terms of (12) with the first term of (14), so as to arrive at $n - 3$ vertices; we see that $18n - 84 \geqslant 12n - 64$, so long as $6n \geqslant 20$ (i.e., $n \geqslant 4$ again). Once more, we see that case (i) is the more laborious; and so (12) is the worst-case gamma-count. The sum is

$$\sum_{m=4}^{n} (6m - 22) = \sum_{s=1}^{n-3} (6s - 4) = 3(n - 3)(n - 2) - 4(n - 3)$$
$$= (n - 3)(3n - 10). \tag{15}$$

We have thus established:

PROPOSITION 11. *The old algorithm* (embodied in Programs A and B) *(i) always yields a complete, economical triangulation, and (ii) takes less than*

$$27(2n^2 - 9n + 10) = 27(n - 2)(2n - 5) \tag{16}$$

a.o. and $O(n^2)$ other operations.

⟦We have shown that the preliminaries take at most $3n^2 - 8n$ gammas, and, by (15), the triangulation loop takes $3n^2 - 19n + 30$ gammas. The total is $6n^2 - 27n + 30$; and (16) follows.⟧

We observe that (16) is an improvement on Theorem 4 of [1], which gives the formula

$$81n(n + 1) - 360. \tag{17}$$

It is interesting to compare the bound (16) with the actual experimental a.o. counts obtained with the programs.

n	A.O. COUNT	THEOR. BOUND	RATIO
Program A			
15	2,511	8,775	3.49
20	4,311	17,010	3.95
27	7,839	33,075	4.22
48	29,088	113,022	3.89
Program B			
8	513	1,782	3.47
16	3,393	10,206	3.01
24	8,829	25,542	2.89
32	16,821	47,790	2.84
40	27,369	76,950	2.81
48	40,473	113,022	2.79
56	56,133	156,006	2.78
64	74,349	205,902	2.769
72	95,121	262,710	2.762
80	118,449	326,430	2.756
88	144,333	397,062	2.751
96	172,773	474,606	2.747

Two causes may be adduced to account for the overestimation of observed a.o. counts by the theoretical bounds. First, find-u() only examines A-vertices for inclusion in a u-list when a B-vertex has been found in the triad; and secondly, many vertices are excluded from a u-list by the first or second discriminant evaluated, whereupon the rest of the three gammas are omitted. These two features of the execution have the major effect; but other over-estimates occur also, and have a lesser inflationary effect. The observed ratios between 2.7 and 4.3 are consonant with this explanation.

9. Performance Bounds: Programs C and D

Beginning with auxiliary functions, we observe that there are now no u-lists or t-lists; so that ins-u(), del-u(), del-S(), and (notably) find-u() are absent. Thus, adjust() now takes time $O(1)$. The main program now takes a minor role, too. The calculation of discriminants and the construction of the D-, A-, and B-lists contribute time $O(n)$ with n gammas, since the u-lists need no longer be computed.

The crux of the matter is in the recursive function split(). We now suppose that an upper bound for the execution time of split() when there are m vertices in the polygon is $T(m)$, and that an upper bound for the gamma-count is $\phi(m)$.

We begin with the time estimate. The first part of the function (lines C112-131) determines whether the selected convex triad is indeed empty or not; and, in the same process, if not, finds the closest re-entrant vertex to the apex. Much as in find-u() before, we see that time $O(q) = O(m)$ is required. Two branches occur: (i) if the triad is empty, time $O(1)$ [adjust() is now less complex and laborious] suffices to remove it to the C-array and prepare the polygon (with at most $m - 1$ vertices) for recursion; (ii) if the triad contains a re-entrant vertex, suppose that the segment QZ [see the discussion in Section 6] divides the polygon into one of r and one of $m - r + 2$, with $3 \leqslant r \leqslant m - 1$. The search for the vertices U and V may take time as great as $2r - 3$ times a constant, and this is $O(m)$; the rest of the function takes only $O(1)$ time. Thus,

$$\begin{aligned} T(m) = \max \{&O(m) + T(m - 1), \\ &O(m) + T(r) + T(m - r + 2)\}. \end{aligned} \tag{18}$$

(Note that the maximum runs over all allowable values of r.) We must assume that the $O(m)$ terms can actually attain behavior proportional to m; so that (18) immediately tells us that $T(m) = \Omega(m)$. Suppose, therefore, that

$$T(m) \sim Cm^{\alpha}, \quad \text{with} \quad \alpha \geqslant 1,\ C > 0. \tag{19}$$

Then, in the first option of the maximum,

$$\begin{aligned} T(m) - T(m-1) &\sim C[m^{\alpha} - (m-1)^{\alpha}] = C[\alpha m^{\alpha-1} - \tfrac{1}{2}\alpha(\alpha-1)m^{\alpha-2} + \ldots] \\ &\sim C\alpha m^{\alpha-1} = O(m), \end{aligned} \tag{20}$$

which tells us that $\alpha \leqslant 2$ (and indeed that $\alpha = 2$ if the $O(m)$ attains the behavior proportional to m). Now examine the second option of the maximum. First, we must maximize $T(r) + T(m - r + 2) \sim C[r^{\alpha} + (m - r + 2)^{\alpha}]$. It is readily seen that the derivative of the last expression (divided by $C\alpha$) is $r^{\alpha-1} - (m - r + 2)^{\alpha-1}$, which, since $\alpha \geqslant 1$, is negative for all $r < m - r + 2$ and positive for all $r > m - r + 2$; there is thus a *minimum* at $r = (m + 2)/2$ and the maximum is attained when $r = 3$ or $m - 1$, with the value $C[3^{\alpha} + (m - 1)^{\alpha}]$. The second option thus becomes the same as the first, and we see that

$$T(m) = O(m^2). \tag{21}$$

Since the preliminaries all take time $O(n)$, it follows that the entire program takes a time $O(n^2)$.

We now turn to the a.o. count. We know from the foregoing that this will have to be $O(n^2)$ also, of course. Retracing our steps over the program, we remember that the preliminaries required n gammas. In the split() function, with m vertices, lines C112-131 require $3q \leqslant 3(m - 3)$ gammas. In the first branch of the function (empty triad removed), we get $\phi(m) \geqslant 3m - 7 + \phi(m - 1)$. In the second branch (split the polygon into two), we get $\phi(m) \geqslant 3m - 7 + \phi(r) + \phi(m - r + 2)$. Thus, much like (18), we obtain

$$\begin{aligned} \phi(m) = \max \{&3m - 7 + \phi(m - 1), \\ &3m - 7 + \phi(r) + \phi(m - r + 2)\}. \end{aligned} \tag{22}$$

Let
$$\phi(m) = am^2 + bm + c, \quad \text{with} \quad a > 0, \tag{23}$$

since $\phi(m)$ must be positive as $m \to \infty$. Then the same argument as before shows that $\phi(r) + \phi(m - r + 2)$ has a minimum when $r = (m + 2)/2$ and a maximum when $r = 3$ or $m - 1$; whence the second option in (22) becomes $3m - 7 + \phi(3) + \phi(m - 1)$, which exceeds the first option by $\phi(3)$. However, a little thought shows that $\phi(3) = 0$, since the triad is empty and there are no re-entrant vertices to test! Thus, we are left with the equation

$$\begin{aligned}\phi(m) - \phi(m-1) &= a[m^2 - (m-1)^2] + b[m - (m-1)] \\ &= a(2m-1) + b = 3m - 7; \qquad (24)\end{aligned}$$

whence

$$a = \frac{3}{2} \quad \text{and} \quad b = a - 7 = -\frac{11}{2}. \qquad (25)$$

The complete solution is obtained when we observe that, as stated above,

$$\phi(3) = 0 = 9a + 3b + c; \qquad (26)$$

whence

$$c = 3. \qquad (27)$$

The complete gamma-count is thus

$$\phi(m) = \frac{3}{2}m^2 - \frac{11}{2}m + 3; \qquad (28)$$

PROPOSITION 12. *The new algorithm* (based on Propositions 9 and 10, and embodied in Programs C and D) *(i) always yields a complete, economical triangulation, and (ii) takes less than*

$$\frac{27}{2}(n^2 - 3n + 2) = \frac{27}{2}(n-1)(n-2) \qquad (29)$$

a.o. and $O(n^2)$ other operations.

⟦We have shown that the preliminaries take n gammas, and the split() function (with all recursions) $\phi(n)$ gammas. The total gamma-count, by (28), is thus $\frac{3}{2}n^2 - \frac{9}{2}n + 3$, and (29) follows on multiplication by 9.⟧

We note that the ratio of the bounds (16) and (29) for the two algorithms is just $(2n-5)/\frac{1}{2}(n-1) \to 4$ as $n \to \infty$.

Comparison of the bound (29) with actual experimental results yields the following table.

n	A.O. COUNT	THEOR. BOUND	RATIO
<u>Program C</u>			
8	207	567	2.74
16	972	2,835	2.92
24	2,169	6,831	3.15
32	3,798	12,555	3.31
40	5,859	20,007	3.41
48	8,352	29,187	3.49
56	11,277	40,095	3.56
64	14,634	52,731	3.60
72	18,423	67,095	3.64
80	22,644	83,187	3.67
88	27,297	101,007	3.70
96	32,382	120,555	3.72
Program D			
15	693	2,457	3.55
20	1,008	4,617	4.58
27	2,493	8,775	3.52
48	4,023	29,187	7.26

The bulk of the work is done in lines C122-124, finding the re-entrant vertex Z (if any). Since only the B-list is scanned, we may grossly over estimate the $3q$ gammas by using $3(m - 3)$; and the same argument as before suggests that the factor of 3 (for the three gammas in the tests), since any failure in the tests will eliminate further computation for that B-vertex. The observed ratios of the theoretical upper bounds to the actual a.o. counts range between 2.7 and 4.6, except for the unusually high ratio of 7.26 for the 48-gon run with Program D. This is comparable with the ratios for Programs A and B.

A couple of additional observations may be made. (i) If we examine the earlier comparisons of Programs A and B with Programs C and D (i.e., of the old and new algorithms); we see that (a) the theoretical bounds have a ratio tending to 4 (for $n = 20$, 40, and 80, the ratios are 3.68, 3.85, and 3.92, respectively), and (b) the observed a.o. counts have ratios rising monotonically from 2.48 to 5.34 for the double square spirals (Programs B and C), and 3.62, 4.28, 3.14, and 7.23 for the miscellaneous polygons run with Programs A and D. Clearly, the 48-gon among these last is a special case, exceptionally well treated by the new algorithm. (ii) The ratios for the double square spirals under Program B (old algorithm), of the bounds to the actual a.o. counts, *decrease* monotonically from 3.47 to 2.75, as n increases; those for Program C (new algorithm) *increase* from 2.74 to 3.72, for the same values of n. The cause of this is not apparent; but one may hazard a guess that it is a peculiarity of the family of double square spirals, relative to the two algorithms, and not a significant universal property of the algorithms themselves.

10. A Further Experiment

To add weight to the experimental evidence, we carried out twelve further runs (six on each of Programs A and D), using a different family of polygons, called "ropes", with $16i$ vertices (i = 1, 2, 3, 4, 5, 6) arranged as a $2i$-fold 8-point square spiral, the skeins being connected into a zig-zag. This is illustrated below in Figures (xvii) - (xx).

Figure (xvii). *D.S.Spiral,* $i = 2$.

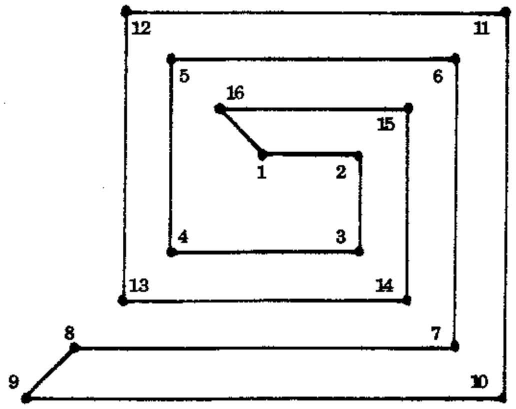
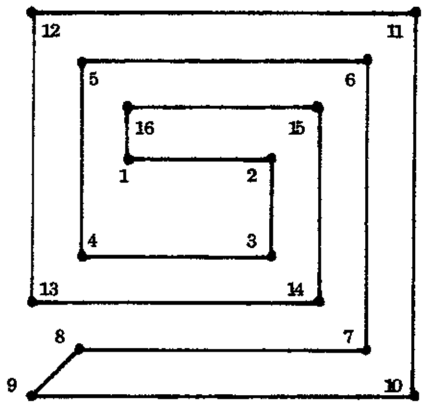
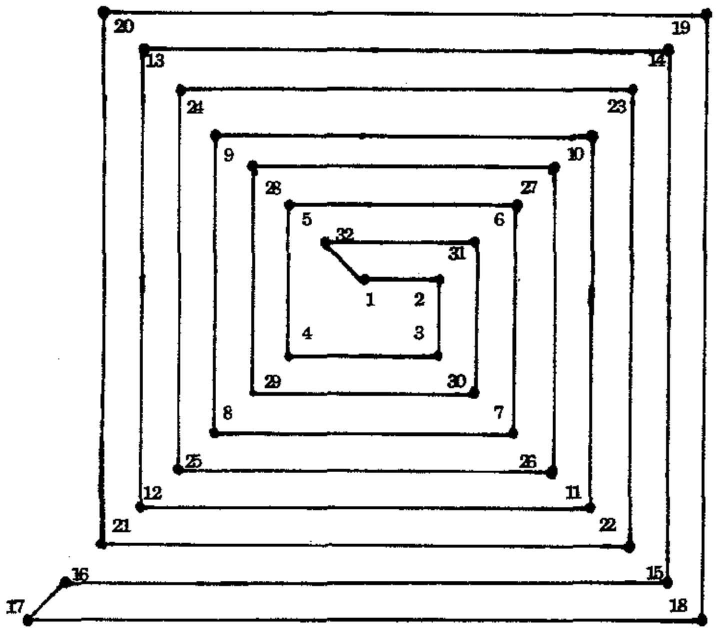


Figure (xviii). *Rope:* $i = 1$.



We note that the d.s.spiral with $i = 2$ and the rope with $i = 1$ (both having 16 vertices) are almost and essentially identical; but, while the d.s.spiral with $i = 4$ winds four times instead of twice, keeping to two skeins, the rope with $i = 2$ still winds twice, but with four skeins (both have 32 vertices). Thereafter, the i-rope has $2i$ skeins, but winds twice; the $2i$-d.s.spiral has two skeins (hence "double") but winds $2i$ times.

Figure (xix). *D.S.Spiral, i = 4.*

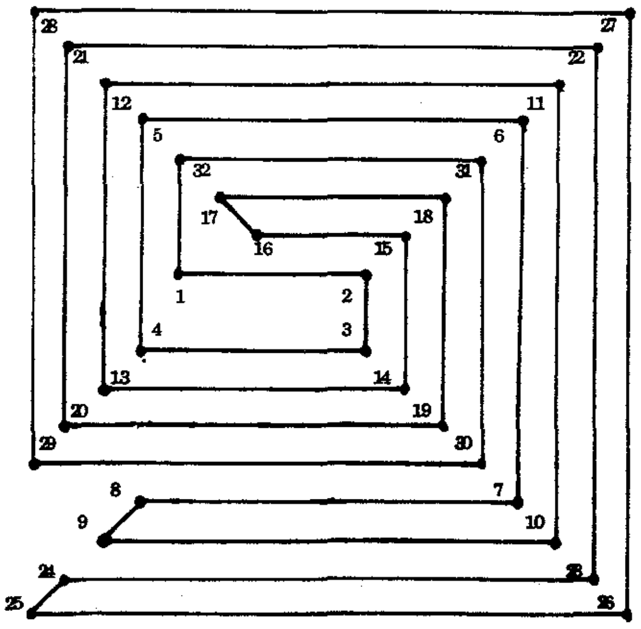


A.O. counts, theoretical upper bounds for these, and the resulting ratios are tabulated below, as before.

n	A.O. COUNT	THEOR. BOUND	RATIO
Program A			
16	3,366	10,206	3.03
32	18,252	47,790	2.618
48	43,011	113,022	2.628
64	78,048	205,902	2.638
80	123,462	326,430	2.644
96	179,172	474,606	2.649
Program D			
16	954	2,835	2.97
32	2,025	12,555	6.20
48	3,708	29,187	7.87
64	6,984	52,731	7.55
80	11,808	83,187	7.04
96	18,198	120,555	6.62

Note: (i) This time, except for the 1-rope (which is virtually identical to the 2-d.s.spiral), the "rope" polygons are extra-easy for the new algorithm to triangulate (about as easy as was the 48-gon run with Program D). (ii) This time, the ratios for the old algorithm (Program A) *increase* with n; while the ratios for the new algorithm (Program D) *decrease* with n (after n = 48). Thus our guess, that the increase or decrease is a function of both the algorithms and the polygons, rather than characteristic of the algorithms alone, is verified (or, at least, strongly indicated).

Figure (xx). *Rope: i = 2.*



11. Conclusions

After a somewhat circuitous journey, we have arrived at two working and practically useful triangulation algorithms, which we have dubbed the "old" (Algorithm 3 from [1]) and the "new", based on Propositions 9 and 10 in the present paper. The old algorithm has been proved to take time $O(n^2)$ and to require no more than (see (16))

$$54(n^2 - \frac{9}{2} n + 5) \qquad (30)$$

a.o. The new algorithm has also been proved to take $O(n^2)$ time and to require no more than (see (29))

$$\frac{27}{2}(n^2 - 3 n + 2) \qquad (31)$$

a.o. The ratio of these bounds tends to 4 as $n \to \infty$; and indeed, we see that

$$\text{"(30)"} \geqslant \alpha \times \text{"(31)"} \quad \text{for all } n \geqslant n_0 \qquad (32)$$

is equivalent to the assertion that (since we know that $n \geqslant 3$) $4n - 10 \geqslant \alpha(n - 1)$; or $(4 - \alpha)n \geqslant 10 - \alpha$. Since we know (from the asymptotic behavior as $n \to \infty$) that

$$\alpha < 4; \qquad (33)$$

we can infer that

$$n_0 = (10 - \alpha)/(4 - \alpha). \qquad (34)$$

Finally, this gives us that the upper bounds for the a.o. counts of the two algorithms satisfy the relationship (32) with (for example)

$$\left.\begin{array}{llll} \alpha = 1 \text{ and } n_0 = 3; & \alpha = 2 \text{ and } n_0 = 4; \\ \alpha = 3 \text{ and } n_0 = 7; & \alpha = 3\tfrac{1}{3} \text{ and } n_0 = 10. \end{array}\right\} \qquad (35)$$

In the similar "C" language programs listed above, we see that the old algorithm takes 264 and 273 lines (Programs B and A, respectively); while the new one takes 218 and 227 lines (Programs C and D, respectively). Thus the new algorithm is somewhat more simple to program, it would seem.

It is clear that any algorithm dealing with an n-vertex polygon in order to triangulate it must take time $\Omega(n)$. Since it is essential to the process of triangulation that the triads removed to the C-array (in which the final triangulation appears) should be *empty* (in the sense of Definition 1) and *removable* (in the sense of Definition 2), it is necessary to verify this fact for every triad removed (numbering $n - 2$, unless some vertices turn out to be redundant —— a situation which cannot be guaranteed). There appears to be no way of doing this, except by examining all of (at least) the re-entrant vertices, which (by Lemma 13 of [1]) may be as many as $n - 3$. It is not clear how this can be achieved in under $(n - 2)(n - 3)$ times some constant. If, indeed, this is impossible; then the bound of $O(n^2)$ is best-possible, and all that can be hoped-for is a reduction of the coefficients of the quadratic expressions (30) and (31). However, there remains the challenge to find an algorithm taking time $o(n^2)$, though certainly $\Omega(n)$; or to prove that no such algorithm is possible.

12. Acknowledgement

While repeating my debt to G. C. Clark and H. Fuchs, mentioned in [1], I wish to add my thanks to Dr T. H. Brylawski of The University of North Carolina for mentioning his result, given in Proposition 9, during a discussion following a presentation of the results reported in [1]. His own interest, I understand, had been in providing a proof that triangulation of a simple closed polygon in the Euclidean plane was always possible.

13. Reference

[1] J. H. HALTON. Triangulation Algorithms for Simple, Closed, Not Necessarily Convex, Polygons in the Plane. (The University of North Carolina at Chapel Hill, Computer Science Department; Technical Report TR 85-008; 1985) 84 pp.