A Simple, Practical, and Fast Polygon Triangulation in $O(n + k \log k)$ Time

Pavel Shpagin[a,*], Vasyl Tereschenko[a]

[a]*Faculty of Computer Science and Cybernetics, Taras Shevchenko National University of Kyiv, Kyiv, Ukraine*

Abstract

We present a practical algorithm for triangulating a simple polygon with n vertices in $O(n + k \log k)$ time, where k is the number of local maxima with respect to the sweep direction. The key idea is a chain-based reformulation of the monotone-decomposition sweep: the sweep maintains active monotone chains rather than individual edges and advances chain pointers lazily, so all regular vertices cost $O(n)$ total while balanced-tree work is confined to $O(k \log k)$ over the $2k$ extrema. We evaluate against state-of-the-art practical triangulation pipelines with publicly available implementations, including Garey et al., Seidel, and Hertel–Mehlhorn, and find that our method is faster across all tested families, with large speedups on nearly convex inputs.

Keywords: simple polygon, triangulation, monotone decomposition, input-sensitive algorithm, plane sweep

1. Introduction

Triangulating simple polygons is a fundamental problem in computational geometry. Chazelle [3] proved that $O(n)$ time is achievable, but the algorithm's complexity has limited practical adoption. The plane sweep of Garey et al. [8], running in $O(n \log n)$ time via monotone decomposition, remains standard; see de Berg et al. [5]. Seidel [13] gave a simplified randomized $O(n \log^* n)$ expected-time algorithm.

We present an algorithm running in $O(n + k \log k)$ time, where k is the number of local maxima with respect to the sweep direction. This interpolates between $O(n)$ for convex polygons ($k = 1$) and $O(n \log n)$ for worst-case polygons ($k = \Theta(n)$), and can improve upon classical methods when k is substantially smaller than n. We show in Section 5 that $k \leq r + 1$, where r is the number of reflex vertices, so the bound also implies $O(n + r \log r)$.

The key observation is that the sweep needs to process only local extrema, and regular vertices can be handled implicitly by advancing chain pointers lazily. This yields $O(k)$ sweep events and $O(n)$ total pointer advancement, with balanced-tree operations totaling $O(k \log k)$.

Beyond the improved bound, the method is deliberately simple: it follows the classical monotone-decomposition sweep with the same event order and tie-breaking, but represents the status structure by monotone chains and advances along chains lazily. In Section 6, our implementation is faster than the standard Garey et al. pipeline across all tested polygon families and sizes, suggesting a new practical baseline for polygon triangulation. For clarity, we assume general position in the analysis; the implementation resolves equal y-coordinates by a deterministic lexicographic tie-break.

Related work. The ear-clipping method is among the oldest practical triangulation algorithms; for smaller inputs, its $O(n^2)$ complexity [6] is still attractive due to its simplicity. The classical plane sweep of Garey et al. [8] decomposes a polygon into y-monotone pieces and triangulates each in linear time, achieving $O(n \log n)$ overall; see de Berg et al. [5]. Triangulation of monotone polygons, the linear-time subroutine underlying this pipeline, goes back at least to Fournier and Montuno [7].

Several works relate running time to geometric properties of the input. Hertel and Mehlhorn [9] gave a sweep whose cost depends on the number of start vertices s and runs in $O(n + s \log s)$; for simple polygons, $s \leq r + 1$.

*Corresponding author
Email address: pavelandrewshpagin@knu.ua (Pavel Shpagin)

Chazelle and Incerpi [2] study triangulation under shape-complexity measures. Toussaint [15, 16] gives output-sensitive triangulation algorithms. Our parameter k counts local maxima in the sweep direction; Section 5 shows $k \leq r + 1$ and it can be strictly smaller than r when many reflex vertices are regular.

Worst-case bounds in n have also been improved. Tarjan and Van Wyk [14] achieved $O(n \log \log n)$ using sophisticated data structures, later simplified by Kirkpatrick et al. [11]. Randomized incremental approaches proved fruitful: Clarkson et al. [4] gave an $O(n \log^* n)$ expected-time Las Vegas algorithm, which Seidel [13] simplified significantly via trapezoidal decomposition. The breakthrough came with Chazelle's [3] deterministic $O(n)$ algorithm, but its complexity has limited practical adoption; Amato et al. [1] provided further simplifications to the randomized approach. Keil [10] surveys polygon decomposition more broadly.

Since both monotone decomposition and the maxima count k depend on the sweep direction, the choice of direction matters; Shin and Kim [12] study selecting directions that optimize monotone-chain decompositions.

Our contribution is a deterministic chain-based reformulation of monotone decomposition that makes an extrema parameter k explicit in both the analysis and the implementation: the sweep performs balanced-tree work only at the $2k$ extremal events, while all regular vertices are handled via amortized pointer advancement along chains. For correctness, we prove equivalence to the classical helper-based monotone-decomposition sweep (see, e.g., de Berg et al. [5]): under the same event order and tie-breaking, our chain-based sweep produces the same decomposition diagonals.

Organization. Section 2 defines vertex types, monotone chains, left-boundary chains, and the event-complexity parameter k. Section 3 presents the algorithm. Section 4 establishes correctness. Section 5 analyzes complexity and relates k to the reflex count r. Section 6 provides experimental evaluation. Section 7 discusses extensions.

2. Preliminaries

Let P be a simple polygon with vertices $v_0, v_1, \ldots, v_{n-1}$ listed in counterclockwise order along the boundary ∂P. We write $v_i = (x_i, y_i)$ for the coordinates of each vertex and adopt the convention that indices are taken modulo n, so $v_{-1} = v_{n-1}$ and $v_n = v_0$. The *interior angle* at vertex v_i is the angle $\angle v_{i-1} v_i v_{i+1}$ measured inside P. A vertex is *convex* if its interior angle is at most π and *reflex* if its interior angle strictly exceeds π. We denote by r the number of reflex vertices.

Definition 1 (Vertex classification)**.** Assuming general position (no two vertices share the same y-coordinate), each vertex v_i is classified according to the relative y-coordinates of its neighbors:

- **Start vertex:** $y_{i-1} < y_i$ and $y_{i+1} < y_i$, with interior angle $< \pi$.
- **Split vertex:** $y_{i-1} < y_i$ and $y_{i+1} < y_i$, with interior angle $> \pi$.
- **End vertex:** $y_{i-1} > y_i$ and $y_{i+1} > y_i$, with interior angle $< \pi$.
- **Merge vertex:** $y_{i-1} > y_i$ and $y_{i+1} > y_i$, with interior angle $> \pi$.
- **Regular vertex:** exactly one neighbor has y-coordinate greater than y_i.

Start and split vertices are *local maxima*; end and merge vertices are *local minima*. Split and merge vertices are precisely the reflex vertices among local extrema. Regular vertices partition into two subtypes based on whether the polygon interior lies to their left or right as one traverses the boundary; this distinction is needed in ADVANCE to mirror the classical sweep's regular-vertex behavior, but it does not affect the asymptotic bounds.

Definition 2 (Monotone chain)**.** A *y-monotone chain* is a maximal contiguous sequence of boundary vertices whose y-coordinates are strictly monotonic (strictly increasing or strictly decreasing) along the chain. Each chain connects a local maximum to a local minimum.

The boundary ∂P decomposes uniquely into monotone chains, with consecutive chains sharing their endpoint extrema. Let k denote the number of local maxima (equivalently, local minima) with respect to the sweep direction; then there are $2k$ chains, and k controls the number of sweep events in our formulation.

Definition 3 (Left-boundary chain)**.** A monotone chain is a *left-boundary chain* if, when traversed from its upper endpoint to its lower endpoint, the polygon interior lies to the right of the traversal direction. Equivalently, for a counterclockwise-oriented polygon, a chain is left-boundary if its traversal direction (downward) opposes the boundary orientation.

At each local maximum, exactly one of the two originating chains is left-boundary; at each local minimum, exactly one of the two terminating chains is left-boundary.

3. Algorithm

The algorithm consists of three phases: (1) chain construction and vertex classification, (2) monotone decomposition via chain-based plane sweep, and (3) triangulation of monotone pieces. We describe each phase in detail.

3.1. Phase 1: Chain Construction

A single traversal of ∂P classifies each vertex according to Definition 1 and partitions the boundary into monotone chains. For each vertex v_i, we compare y_i to y_{i-1} and y_{i+1} and compute the cross product $(v_{i+1} - v_i) \times (v_i - v_{i-1})$ to determine convexity. Simultaneously, we record each chain as an array of vertex indices from its upper endpoint (local maximum) to its lower endpoint (local minimum), and identify the incident left-boundary chain at each extremum (Definition 3). This phase runs in $O(n)$ time and produces $O(k)$ chains.

3.2. Phase 2: Monotone Decomposition

A polygon is *y-monotone* if every horizontal line intersects it in a connected set (either empty, a point, or a segment). Split vertices violate monotonicity by creating local maxima where the boundary diverges downward; merge vertices violate it by creating local minima where boundary paths converge from above. The decomposition phase inserts diagonals to eliminate all split and merge vertices, partitioning P into y-monotone subpolygons.

Sweep-line status structure. Unlike the classical algorithm, which maintains individual edges in a balanced search tree T, we maintain *active left-boundary chains*. A chain C is *active* at sweep height y if y lies strictly between the y-coordinates of C's upper and lower endpoints. The tree T stores active left-boundary chains ordered by their x-coordinate at the current sweep height.

Each chain C in T maintains:

1. **Edge pointer** *C.curr*: initialized to the topmost edge of C, this pointer tracks our position within the chain. The upper vertex of *C.curr* serves as the *slab entry*—the default diagonal target when no pending merge exists.
2. **Pending merge** *C.pending*: either null or a merge vertex awaiting connection to a lower vertex. When a merge vertex v is processed and C is immediately to v's left, we set *C.pending* $\leftarrow v$.

Lazy edge pointer advancement. The comparison function for T, when comparing chains at sweep height y, first advances each chain's edge pointer to ensure the current edge spans y:

```
1: procedure ADVANCE($C, y$)
2:     while $C.curr.lower.y > y$ do
3:         $C.curr \leftarrow$ next edge down $C$
4:         $u \leftarrow C.curr.upper$                                ▹ vertex just passed
5:         if $u$ is regular and $C.pending \neq$ NULL and InteriorRight($u$) then
6:             $D \leftarrow D \cup \{(u, C.pending)\}$; $C.pending \leftarrow$ NULL
7:         end if
8:     end while
9: end procedure
```

Here InteriorRight(u) is the standard regular-vertex side test from the classical sweep (i.e., the subtype of regular vertices handled like an end vertex); it can be decided from the local boundary configuration at u. In Algorithm 1, before inspecting a chain's fields at event vertex v we call ADVANCE on that chain at height $y(v)$; balanced-tree comparisons may also invoke ADVANCE as needed. After advancement, the x-coordinate of C at height y is computed by linear interpolation along *C.curr*. Since each vertex is visited by at most one chain's pointer exactly once, the total cost of all pointer advancements (and any diagonals emitted inside ADVANCE) is $O(n)$, amortized over all tree operations.

Event processing. The sweep processes local extrema in decreasing y-order. Let E denote the sorted list of extrema; $|E| = 2k$, where k is the number of local maxima (equivalently local minima).

Algorithm 1 Chain-Based Monotone Decomposition

Require: Polygon P with classified vertices and identified left-boundary chains

Ensure: Diagonal set D partitioning P into y-monotone subpolygons

```
 1: $E \leftarrow$ local extrema sorted by decreasing $y$-coordinate
 2: $T \leftarrow$ empty balanced BST of active left-boundary chains
 3: $D \leftarrow \emptyset$
 4: for each extremum $v$ in $E$ do
 5:     switch type of $v$
 6:     case Start
 7:     $C \leftarrow$ left-boundary chain originating at $v$
 8:     Insert $C$ into $T$
 9:     Initialize $C.curr$ to top edge, $C.pending \leftarrow$ NULL
10:     case End
11:     $R \leftarrow$ left-boundary chain terminating at $v$
12:     ADVANCE($R, y(v)$)                    ▹ updates $R.curr$ and may emit regular-vertex diagonals
13:     if $R.pending \neq$ NULL then
14:         $D \leftarrow D \cup \{(v, R.pending)\}$                    ▹ Connect pending merge downward
15:     end if
16:     Remove $R$ from $T$
17:     case Split
18:     $L \leftarrow$ predecessor of $v$ in $T$                    ▹ Chain immediately left of $v$
19:     ADVANCE($L, y(v)$)                    ▹ updates $L.curr$ and may clear $L.pending$
20:     if $L.pending \neq$ NULL then
21:         $D \leftarrow D \cup \{(v, L.pending)\}$; $L.pending \leftarrow$ NULL
22:     else
23:         $D \leftarrow D \cup \{(v, L.curr.upper)\}$                    ▹ Slab entry
24:     end if
25:     $C \leftarrow$ left-boundary chain originating at $v$
26:     Insert $C$ into $T$
27:     Initialize $C.curr$ to top edge, $C.pending \leftarrow$ NULL
28:     case Merge
29:     $R \leftarrow$ left-boundary chain terminating at $v$
30:     ADVANCE($R, y(v)$)                    ▹ updates $R.curr$ and may emit regular-vertex diagonals
31:     $L \leftarrow$ predecessor of $R$ in $T$                    ▹ Chain immediately left
32:     ADVANCE($L, y(v)$)                    ▹ updates $L.curr$ and may clear $L.pending$
33:     if $R.pending \neq$ NULL then
34:         $D \leftarrow D \cup \{(v, R.pending)\}$
35:     end if
36:     if $L.pending \neq$ NULL then
37:         $D \leftarrow D \cup \{(v, L.pending)\}$
38:     end if
39:     $L.pending \leftarrow v$
40:     Remove $R$ from $T$
41: end for
42: return $D$
```

The complete decomposition procedure is given in Algorithm 1. For split vertices, we connect upward to either a pending merge (if one exists on the immediately-left chain) or to the slab entry (the upper vertex of the current edge

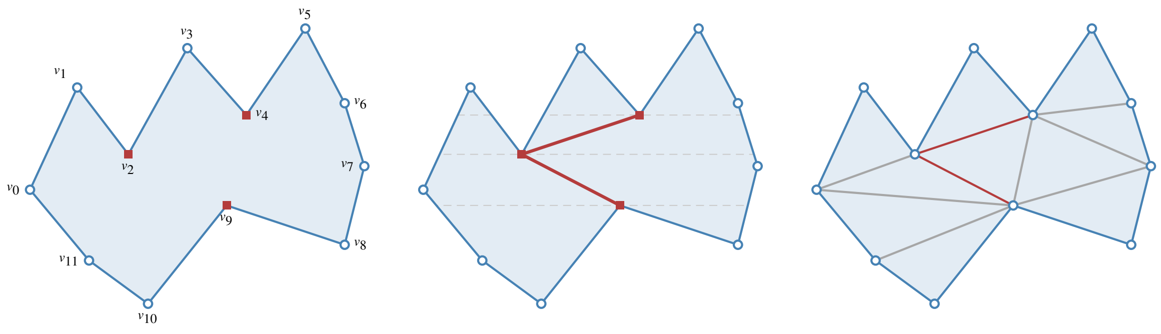
v_1
v_3
v_5
v_4
v_6
v_2
v_7
v_0
v_9
v_8
v_{11}
v_{10}
(a) Input: $n = 12$, 3 split/merge vertices
(b) Decomposition diagonals
(c) Complete triangulation

Figure 1: Triangulation of a 12-vertex polygon with 3 split/merge vertices (squares). (a) Input polygon P; circles denote convex vertices, squares denote split/merge vertices v_2 (merge), v_4 (merge), v_9 (split). (b) Monotone decomposition: dashed lines indicate sweep heights at the split/merge vertices; bold diagonals (v_2, v_4) and (v_9, v_2) partition P into three y-monotone pieces. (c) Final triangulation with $n-2 = 10$ triangles; decomposition diagonals in bold, triangulation diagonals in gray.

on that chain). For merge vertices, we first resolve any pending merges on both the terminating chain and the chain to its left, then register the current vertex as pending on the left chain.

3.3. Phase 3: Triangulation

The diagonals from Phase 2 partition P into y-monotone subpolygons. We construct a doubly-connected edge list (DCEL), i.e., a half-edge structure, from the polygon boundary edges together with the inserted diagonal set D (with $|D| \le r$). Each face is extracted by traversing half-edges, yielding the vertex sequence of each monotone subpolygon.

Each y-monotone polygon with m vertices is triangulated in $O(m)$ time using the classical stack-based algorithm: vertices are processed in y-sorted order, maintaining a stack representing the "reflex chain" of vertices not yet triangulated. When a vertex from the opposite chain is encountered, all stack vertices are triangulated; when a vertex from the same chain is encountered, we triangulate as many stack vertices as remain visible. Since the sum of face sizes equals $n + 2|D| = O(n)$, the total triangulation time is $O(n)$.

3.4. Illustrative Example

Figure 1 illustrates the algorithm on a polygon with $n = 12$ vertices and 3 split/merge vertices (v_2, v_4 are merges; v_9 is a split). The sweep processes 8 local extrema. At merge v_4, the pending mechanism registers v_4 on the left chain. At merge v_2, the pending v_4 triggers diagonal (v_2, v_4), then v_2 becomes pending. At split v_9, the pending v_2 triggers diagonal (v_9, v_2). The two diagonals partition P into three y-monotone faces, yielding 10 triangles.

4. Correctness

We prove correctness by showing that Algorithm 1 is equivalent to the classical edge-based monotone-decomposition sweep (e.g., de Berg et al. [5]), which is known to output interior, non-crossing diagonals that eliminate all split/merge vertices and hence yield a y-monotone decomposition. Throughout, we assume general position (distinct y-coordinates); standard symbolic perturbation handles degeneracies and does not affect the argument.

Fix a height y strictly between two consecutive event levels. For each active left-boundary chain C, let $e(C, y)$ be the unique edge of C whose y-span contains y after calling Advance, and let $ih(C, y)$ be the upper endpoint of $e(C, y)$. The BST T stores active chains ordered by the x-coordinate of $e(C, y)$ at height y, so predecessor queries in T identify the same active edge immediately left of a query point as in the classical sweep.

Definition 4 (Chain helper encoding)**.** For an active chain C at height y, we interpret the classical helper of the current active edge as follows: if $C.pending \neq$ NULL then $helper(e(C, y)) = C.pending$ (a merge vertex); otherwise $helper(e(C, y)) = ih(C, y)$.

Lemma 5 (Regular vertices via ADVANCE)**.** *Along the whole sweep,* ADVANCE *walks each chain pointer monotonically downward and encounters each regular vertex exactly once. Moreover, whenever the classical sweep would insert a diagonal at a regular vertex because the relevant helper is a merge vertex,* ADVANCE *emits the same diagonal* $(u, C.pending)$ *and clears* $C.pending$*.*

Proof. ADVANCE only moves $C.curr$ forward along C, so each chain vertex can become the upper endpoint of the current edge at most once. The only diagonals emitted inside ADVANCE are triggered at regular vertices with $\mathsf{InteriorRight}(u)$ and a non-null pending merge. By Definition 4, $C.pending$ stores exactly the classical merge-helper state on the corresponding active edge, and the classical sweep inserts exactly this diagonal in the regular-vertex case [5]. □

Lemma 6 (Equivalence of diagonals)**.** *Let* D_{chain} *be the set of diagonals produced by Algorithm 1 together with any diagonals emitted inside* ADVANCE*. Let* D_{text} *be the set of diagonals produced by the classical edge-based sweep on the same polygon and event order. Then* $D_{\text{chain}} = D_{\text{text}}$.

Proof. At any height y between event levels, the in-order order of T is the left-to-right order of intersections of the active edges $e(C, y)$ with the sweep line, so predecessor queries select the same "edge immediately left" object as in the classical sweep. By Definition 4 and Lemma 5, the helper information on that active edge (including diagonals that would be emitted at regular vertices) agrees between the two representations.

Now consider an extremum event vertex v. In the START/END cases, insertion/removal of the corresponding left-boundary chain matches insertion/removal of the corresponding active edge, and a pending merge (if any) is connected exactly when the classical helper is a merge. In the SPLIT case, both sweeps connect v to the helper of the edge immediately left of v, which by Definition 4 is either a pending merge or the implicit helper ih. In the MERGE case, both sweeps connect v to any relevant merge-helpers on the terminating and left edges and then set the helper of the left edge to v, which we encode by setting the left chain's pending to v. Therefore the same diagonal is inserted (if any) at every event, and both algorithms maintain equivalent status information throughout the sweep. □

Theorem 7 (Correctness)**.** *Algorithm 1 produces a valid set of non-crossing diagonals partitioning* P *into* y*-monotone subpolygons.*

Proof. By Lemma 6, Algorithm 1 outputs exactly the same diagonals as the classical monotone-decomposition sweep, interpreted through our chain representation. The classical sweep's diagonals are interior, non-crossing, and remove all split/merge vertices, yielding y-monotone pieces [5]. Hence the same properties hold for our output. □

5. Complexity Analysis

Remark 8 (Basic bounds on k)**.** Under the general position assumptions, no two local maxima can be adjacent along ∂P, so $1 \leq k \leq \lfloor n/2 \rfloor$. Consequently, the chain-based sweep processes exactly $2k$ extrema events.

Lemma 9 (Diagonal bound)**.** *Let* s *and* m *be the numbers of split and merge vertices of* P *(equivalently, the numbers of reflex local maxima and reflex local minima). Then the monotone-decomposition phase (Algorithm 1 together with any diagonals emitted inside* ADVANCE*) inserts at most* $s + m$ *diagonals. In particular,* $|D| \leq s + m \leq r$*.*

Proof. Each split vertex triggers exactly one diagonal insertion in Algorithm 1, contributing s diagonals.

It remains to bound diagonals created due to the `pending` mechanism. Only merge vertices are ever assigned to a field $C.pending$. Each such assignment $C.pending \leftarrow v$ occurs at the unique processing step of merge vertex v and can happen only once for that vertex. Afterwards, that stored value is consumed exactly once: it is used to form a diagonal either at a later split/merge event when the chain is immediately left of the current vertex, or at a regular vertex encountered during ADVANCE, or at the end vertex where the chain terminates; in all cases the stored value is overwritten (or the chain is removed), so it cannot generate a second diagonal. Therefore at most one diagonal is generated per merge vertex via `pending`, contributing at most m diagonals.

Hence $|D| \leq s + m \leq r$ since every split and merge vertex is reflex. □

Theorem 10 (Complexity)**.** *A simple polygon with* n *vertices and* k *local maxima (equivalently* k *local minima) can be triangulated in* $O(n + k \log k)$ *time and* $O(n)$ *space.*

Proof. Phase 1 (Chain construction): A single traversal classifies all vertices and constructs all chains in $O(n)$ time using $O(n)$ space.

Phase 2 (Monotone decomposition):

- *Sorting:* The $2k$ local extrema are sorted in $O(k \log k)$ time.
- *Event processing:* There are $2k$ events. Each event involves $O(1)$ BST operations (insertions, deletions, predecessor queries), each taking $O(\log k)$ time since $|T| = O(k)$.
- *Edge pointer advancement:* The comparison function advances edge pointers lazily. Each of the n vertices is visited at most once across all advancements (each vertex belongs to exactly one chain and is passed exactly once by that chain's pointer). Total advancement cost: $O(n)$.

The decomposition phase totals $O(n + k \log k)$ time.

Phase 3 (Triangulation): Constructing the DCEL takes $O(n + |D|) = O(n)$ time by Lemma 9. Face extraction and triangulation together take $O(\sum_f |f|)$ time, where the sum is over all faces f. Since faces partition the plane inside P, and each original edge and diagonal appears in exactly two face boundaries, $\sum_f |f| = 2(n + |D|) = O(n)$.

Total time: $O(n) + O(n + k \log k) + O(n) = O(n + k \log k)$.

Space: Storing the polygon requires $O(n)$ space. The chain data structures use $O(n)$ total (chains partition the vertices). The BST T contains at most $O(k)$ chains, each with $O(1)$ auxiliary data. The DCEL for face extraction uses $O(n + |D|) = O(n)$ space. Total: $O(n)$. □

Lemma 11 (Counting extrema)**.** *Assume general position (no horizontal edges and no equal y-coordinates). Let s, t, e, m denote the numbers of start, split, end, and merge vertices of a simple polygon P, respectively. Then*

$$s = m + 1 \qquad \textit{and} \qquad e = t + 1.$$

Consequently, the number of local maxima is $k = s + t = t + m + 1$, and hence $k \le r + 1$ since split and merge vertices are reflex and $t + m \le r$.

Proof. We prove $s = m + 1$ using superlevel sets. For $y \in \mathbb{R}$, define

$$P_{\ge y} := P \cap \{(x, y') : y' \ge y\},$$

and let $c(y)$ be the number of connected components of $P_{\ge y}$. For y above all vertices, $P_{\ge y} = \emptyset$ and $c(y) = 0$; for y below all vertices, $P_{\ge y} = P$, which is connected, so $c(y) = 1$.

As y decreases, $P_{\ge y}$ grows monotonically, so $c(y)$ can change only when crossing a vertex height, and it can change only by the birth of a new component or by the merge of two components. In a sufficiently small neighborhood of a *start* vertex v (a convex local maximum), the polygon interior lies strictly below v, so for y just below $y(v)$ a new connected component of $P_{\ge y}$ appears near v; thus $c(y)$ increases by 1. In a sufficiently small neighborhood of a *merge* vertex v (a reflex local minimum), two boundary branches approach v from above and belong to two distinct components of $P_{\ge y}$ for y just above $y(v)$; when y passes below $y(v)$, the point $v \in P_{\ge y}$ connects those components, and $c(y)$ decreases by 1. For split, end, and regular vertices, no new component of $P_{\ge y}$ is born and no two components first become connected at that vertex, so $c(y)$ does not change there.

Therefore $c(y)$ increases exactly s times and decreases exactly m times as y sweeps from $+\infty$ to $-\infty$, and since the net change is 1, we obtain $s - m = 1$, i.e., $s = m + 1$.

Finally, reflecting the polygon by mapping $y \mapsto -y$ swaps local maxima and local minima while preserving convex/reflex status, mapping start to end and merge to split. Applying the already proved identity to the reflected polygon yields $e = t + 1$. □

Relationship to reflex-sensitive bounds. By Lemma 11, $k \le r + 1$, and therefore our bound also implies $O(n + r \log r)$. The advantage of phrasing the running time in terms of k is that k can be asymptotically smaller than r (e.g., when many reflex vertices are regular rather than extrema), yielding a strictly smaller $k \log k$ term.

Table 1: Running time (ms) on convex, dent, and random polygons (mean ± stdev over instances).

Type	n	k	**Ours**	Seidel	Garey	Hertel–Mehlhorn
Convex	500	1 ± 0	**0.00 ± 0.00**	28.30 ± 1.48	0.10 ± 0.01	0.64 ± 0.18
	1,000	1 ± 0	**0.01 ± 0.00**	91.55 ± 57.69	0.22 ± 0.13	1.80 ± 0.11
	2,000	1 ± 0	**0.03 ± 0.01**	30.37 ± 6.44	0.45 ± 0.10	5.56 ± 0.14
	5,000	1 ± 0	**0.04 ± 0.00**	30.94 ± 3.09	0.81 ± 0.02	34.27 ± 2.31
	10,000	1 ± 0	**0.09 ± 0.01**	40.71 ± 13.13	2.02 ± 0.11	138.81 ± 10.29
Dent	500	2 ± 0	**0.09 ± 0.03**	35.04 ± 4.30	0.19 ± 0.05	0.63 ± 0.13
	1,000	2 ± 0	**0.09 ± 0.01**	35.21 ± 4.47	0.22 ± 0.06	1.63 ± 0.14
	2,000	2 ± 0	**0.20 ± 0.02**	33.32 ± 2.05	0.41 ± 0.06	5.26 ± 0.69
	5,000	2 ± 0	**0.51 ± 0.05**	36.46 ± 4.48	1.02 ± 0.10	27.32 ± 3.40
	10,000	2 ± 0	**1.14 ± 0.17**	38.45 ± 1.97	2.32 ± 0.30	102.51 ± 7.97
Random	500	160 ± 2	**0.15 ± 0.02**	39.21 ± 9.84	0.21 ± 0.05	1.60 ± 0.28
	1,000	329 ± 4	**0.34 ± 0.09**	53.49 ± 30.02	0.66 ± 0.38	23.66 ± 11.18
	2,000	666 ± 9	**0.64 ± 0.16**	30.26 ± 2.67	0.91 ± 0.29	11.34 ± 0.94
	5,000	1661 ± 17	**1.45 ± 0.07**	33.20 ± 2.53	1.86 ± 0.04	63.56 ± 2.70
	10,000	3316 ± 12	**2.84 ± 0.16**	41.08 ± 3.98	4.65 ± 0.82	291.75 ± 122.72

Constant factors in the $k \log k$ term. Only local extrema generate tree events, so the sweep performs $O(k)$ balanced-tree operations and hence $O(k \log k)$ key comparisons. Ordering comparisons advance chain pointers lazily, and each boundary vertex is advanced at most once over the whole sweep, so the geometric work inside comparisons totals $O(n)$. In practice, this keeps the $k \log k$ term small compared to edge-based sweeps.

6. Experimental Evaluation

We implemented the algorithm in C++ and compared against state-of-the-art practical triangulation pipelines with publicly available implementations: Garey et al. [8] via the `polypartition` library, Seidel [13], and a Hertel–Mehlhorn [9] convex-partition baseline that triangulates each convex piece by a fan.

Environment. Benchmarks were run under WSL2 (Linux kernel 6.6.87.2) on a 13th Gen Intel(R) Core(TM) i5-1345U (6 cores / 12 threads) with 16 GiB RAM, using g++ 13.3.0 and Python 3.12.3.

Benchmark setup. For each polygon family and size n, we generate five instances (seeds $0, \ldots, 4$) and report *mean ± standard deviation* of wall-clock time (ms). The table also reports k, the number of local maxima in the sweep direction, computed per instance and aggregated in the same way. We use three families: Convex (affine images of a regular n-gon), Dent (one inward perturbation), and Random (star-shaped polygons from random angles and radii).

Correctness checks. We validate that each output forms a triangulation of the input polygon (exactly $n - 2$ triangles, and edges intersect only at shared endpoints).

Our main convex, dent, and random results are included verbatim from the benchmark harness:

For convex inputs, our implementation includes the standard linear-time fan triangulation fast path (triangulating around a fixed vertex), achieving roughly 20× speedup over Garey et al. for $n = 10{,}000$. On dent polygons (very small $k = 2$), the chain-based sweep is consistently fastest, roughly 2× faster than Garey et al. On random polygons, our method is fastest for all reported sizes in Table 1. Seidel exhibits high constant factors in our experiments; the Hertel–Mehlhorn convex partition baseline is substantially slower than the monotone-decomposition baselines in our setup.

7. Discussion

Comparison with existing algorithms. Table 2 summarizes theoretical complexity. The primary comparison is against the classical $O(n \log n)$ monotone decomposition of Garey et al. [8] and Seidel's $O(n \log^* n)$ randomized algorithm [13].

Table 2: Algorithm comparison.

Algorithm	**Complexity**	**Notes**
Garey et al. [8]	$O(n \log n)$	monotone decomposition
Hertel–Mehlhorn [9]	$O(n + s \log s)$	start-vertex sensitive in s
Seidel [13]	$O(n \log^* n)$ exp.	randomized
Chazelle [3]	$O(n)$	impractical
This paper	$O(n + k \log k)$	maxima-sensitive in k

Our algorithm achieves the maxima-sensitive bound $O(n+k \log k)$, where k is the number of local maxima with respect to the sweep direction. The sweep processes $O(k)$ events with $O(\log k)$ balanced-tree operations each, while lazy pointer advancement handles all n vertices in $O(n)$ amortized time. This interpolates between $O(n)$ on nearly monotone inputs and $O(n \log n)$ in the worst case. The implementation uses cache-friendly monotone chain representation for efficient constant factors.

Extensions. The algorithm extends naturally to polygons with holes: each hole boundary contributes its own set of chains and extrema, and the sorted extrema lists are merged. The analysis carries through with k now denoting the total number of local maxima across all boundaries (equivalently, local minima). See also Tereschenko and Tereschenko [17] for triangulating regions between polygonal boundaries.

Implementation. For k close to n, a simpler edge-based sweep may have better constants. Reducing constants in this high-event regime without sacrificing the k-sensitive structure remains an open practical challenge.

Open problems. Can the $O(k \log k)$ term be reduced to $O(k)$? Do analogous bounds hold for polygons with holes or higher-dimensional tetrahedralization?

Acknowledgments

The authors thank the computational geometry group at Taras Shevchenko National University of Kyiv for helpful discussions.

References

[1] N. M. Amato, M. T. Goodrich, and E. A. Ramos. A randomized algorithm for triangulating a simple polygon in linear time. *Discrete & Computational Geometry*, 26(2):245–265, 2001.

[2] B. Chazelle and J. Incerpi. Triangulation and shape-complexity. *ACM Transactions on Graphics*, 3(2):135–152, 1984.

[3] B. Chazelle. Triangulating a simple polygon in linear time. *Discrete & Computational Geometry*, 6(5):485–524, 1991.

[4] K. L. Clarkson, R. E. Tarjan, and C. J. Van Wyk. A fast Las Vegas algorithm for triangulating a simple polygon. *Discrete & Computational Geometry*, 4(5):423–432, 1989.

[5] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 3rd edition, 2008.

[6] H. ElGindy, H. Everett, and G. T. Toussaint. Slicing an ear using prune-and-search. *Pattern Recognition Letters*, 14(9):719–722, 1993.

[7] A. Fournier and D. Y. Montuno. Triangulating simple polygons and equivalent problems. *ACM Transactions on Graphics*, 3(2):153–174, 1984.

[8] M. R. Garey, D. S. Johnson, F. P. Preparata, and R. E. Tarjan. Triangulating a simple polygon. *Information Processing Letters*, 7(4):175–179, 1978.

[9] S. Hertel and K. Mehlhorn. Fast triangulation of simple polygons. In *Proc. 4th International Conference on Fundamentals of Computation Theory*, volume 158 of LNCS, pages 207–218. Springer, 1983.

[10] J. M. Keil. Polygon decomposition. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 491–518. Elsevier, 2000.

[11] D. G. Kirkpatrick, M. M. Klawe, and R. E. Tarjan. Polygon triangulation in $O(n \log \log n)$ time with simple data structures. *Discrete & Computational Geometry*, 7(4):329–346, 1992.

[12] H. Shin and D.-S. Kim. Optimal direction for monotone chain decomposition. In *Computational Science and Its Applications – ICCSA 2004*, pages 583–591. Springer, 2004.

[13] R. Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Computational Geometry: Theory and Applications*, 1(1):51–64, 1991.

[14] R. E. Tarjan and C. J. Van Wyk. An $O(n \log \log n)$-time algorithm for triangulating a simple polygon. *SIAM Journal on Computing*, 17(1):143–178, 1988.

[15] G. T. Toussaint. An output-complexity-sensitive polygon triangulation algorithm. In *CG International '90*, pages 443–466. Springer, 1990.

[16] G. T. Toussaint. Efficient triangulation of simple polygons. *The Visual Computer*, 7(5–6):280–295, 1991.

[17] V. Tereschenko and Y. Tereschenko. Triangulating a region between arbitrary polygons. *International Journal of Computing*, 16(3):160–165, 2017.