

A simpler practical polygon triangulation in $O(n + k \log k)$ time

Pavel Shpigin^{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 extrema (local maxima, equivalently local minima) with respect to the sweep direction. The key idea is a chain-based reformulation of monotone decomposition: the sweep maintains active monotone chains rather than individual edges and advances chain pointers lazily, so the work over all regular vertices is $O(n)$ amortized while the sweep performs $O(k \log k)$ balanced-tree work over the $2k$ extrema. Experiments against implementations of Garey et al., Hertel–Mehlhorn, and Seidel show that our method is competitive across polygon families, with large speedups on nearly convex inputs ($k = O(1)$). We also include the standard linear-time fan triangulation fast path for convex polygons.

Keywords: simple polygon, triangulation, monotone decomposition, instance-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 extrema (local maxima, equivalently local minima) 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 $O(k \log k)$.

Related work. The triangulation problem has a rich history spanning four decades. The earliest approaches include the ear-clipping method of ElGindy et al. [6], which runs in $O(n^2)$ time. Garey et al. [8] achieved $O(n \log n)$ time through monotone decomposition, which remains the standard practical approach; see de Berg et al. [5] for a textbook treatment.

Hertel and Mehlhorn [9] described an $O(n \log n)$ algorithm that first triangulates, then merges triangles into a convex partition with at most 4 times the minimum number of convex pieces; this convex-partition step is sometimes used as a triangulation subroutine. Subsequent work focused on reducing the $O(n \log n)$ bound. Tarjan and Van Wyk [14] achieved $O(n \log \log n)$ using sophisticated data structures, later simplified by Kirkpatrick et al. [11]. Randomized approaches proved fruitful: Clarkson et al. [4] gave an $O(n \log^* n)$ expected-time Las Vegas algorithm, which Seidel [13] simplified significantly using trapezoidal decomposition. Amato et al. [1] provided further simplifications of randomized linear-time triangulation.

*Corresponding author

Email address: pavelandrewshpigin@knu.ua (Pavel Shpigin)

The breakthrough came with Chazelle’s [3] deterministic $O(n)$ algorithm, resolving a long-standing open problem. However, the algorithm’s complexity—Involving hierarchical polygon decomposition and intricate merge operations—has limited practical adoption. Fournier and Montuno [7] studied triangulation of monotone polygons. Keil [10] surveys polygon decomposition more broadly.

Beyond n -based worst-case analysis, instance/output complexity measures have also been considered in triangulation. Chazelle and Incerti [2] study triangulation and shape complexity, and Toussaint [15, 16] gives triangulation algorithms whose running time depends on an *output* measure of the produced triangulation. Separately, the number of monotone chains/extrema depends on the sweep direction; Shin and Kim [12] study choosing a direction that optimizes a monotone-chain decomposition.

Our contribution is a deterministic sweep-based algorithm with running time $O(n + k \log k)$ parameterized by the number of local extrema k , achieved by maintaining chains in the sweep status structure and handling regular vertices lazily. To the best of our knowledge, this explicit extrema-parameterization of monotone decomposition (together with an implementation-oriented equivalence proof to the textbook sweep) has not been previously stated for polygon triangulation.

Organization. Section 2 defines vertex types, monotone 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, \dots, 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 $\angle_{v_{i-1}v_iv_{i+1}}$ 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 textbook 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 v_a, v_{a+1}, \dots, v_b such that the y -coordinates are strictly monotonic (either strictly increasing or strictly decreasing) along the sequence. 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. If there are k local maxima, there are also k local minima (since the boundary is a closed curve), and hence $2k$ chains.

Remark 3 (Event-complexity parameter k). Our running time and analysis are stated in terms of k , the number of local maxima (equivalently, local minima) with respect to the sweep direction. This parameter directly controls the number of sweep events in the chain-based formulation.

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). For each local minimum v , we store a pointer to the unique *left-boundary chain* terminating at v —the chain for which, when traversed from upper to lower endpoint, the polygon interior lies to the right.

Definition 4 (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. This phase runs in $O(n)$ time and produces $O(k)$ chains, where k is the number of local maxima (equivalently local minima).

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:      $u \leftarrow C.curr.upper$                                  $\triangleright$  vertex just passed
5:     if  $u$  is regular and  $C.pending \neq \text{NULL}$  and  $\text{InteriorRight}(u)$  then
6:        $D \leftarrow D \cup \{(u, C.pending)\}; C.pending \leftarrow \text{NULL}$ 
7:     end if
8:   end while
9: end procedure

```

Here $\text{InteriorRight}(u)$ is the standard regular-vertex side test from the textbook sweep (i.e., the subtype of regular vertices handled like an end vertex); it can be decided from the local boundary configuration at u . 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:     Insert left-boundary chain originating at  $v$  into  $T$ 
8:     Initialize  $C.curr$  to top edge,  $C.pending \leftarrow \text{NULL}$ 
9:   case End
10:     $R \leftarrow$  left-boundary chain terminating at  $v$ 
11:    if  $R.pending \neq \text{NULL}$  then
12:       $D \leftarrow D \cup \{(v, R.pending)\}$                                  $\triangleright$  Connect pending merge downward
13:    end if
14:    Remove  $R$  from  $T$ 
15:   case Split
16:      $L \leftarrow$  predecessor of  $v$  in  $T$                                  $\triangleright$  Chain immediately left of  $v$ 
17:     if  $L.pending \neq \text{NULL}$  then
18:        $D \leftarrow D \cup \{(v, L.pending)\}; L.pending \leftarrow \text{NULL}$ 
19:     else
20:        $D \leftarrow D \cup \{(v, L.curr.upper)\}$                                  $\triangleright$  Slab entry
21:     end if
22:     Insert left-boundary chain originating at  $v$  into  $T$ 
23:   case Merge
24:      $R \leftarrow$  left-boundary chain terminating at  $v$ 
25:      $L \leftarrow$  predecessor of  $R$  in  $T$                                  $\triangleright$  Chain immediately left
26:     if  $R.pending \neq \text{NULL}$  then
27:        $D \leftarrow D \cup \{(v, R.pending)\}$ 
28:     end if
29:     if  $L.pending \neq \text{NULL}$  then
30:        $D \leftarrow D \cup \{(v, L.pending)\}$ 
31:     end if
32:      $L.pending \leftarrow v$ 
33:     Remove  $R$  from  $T$ 
34: end for
35: 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 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) or equivalent adjacency structure from the original boundary edges plus the $|D| \leq r$ diagonals. Each face is extracted by traversing half-edges, yielding the vertex sequence of each monotone subpolygon.

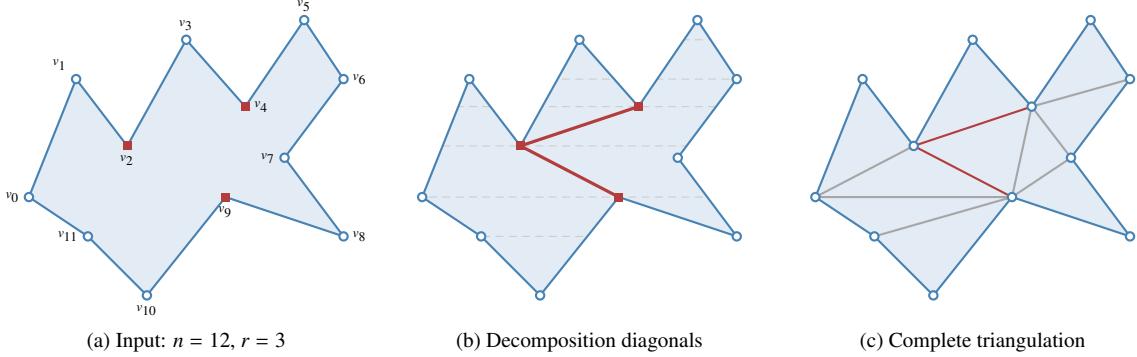


Figure 1: Triangulation of a 12-vertex polygon with 3 reflex vertices (squares). (a) Input polygon P ; circles denote convex vertices, squares denote reflex vertices v_2 (merge), v_4 (merge), v_9 (split). (b) Monotone decomposition: dashed lines indicate sweep heights at local extrema; bold diagonals (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.

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 $r = 3$ reflex 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

The geometric idea of Algorithm 1 is to implement the classical monotone-decomposition sweep (e.g., de Berg et al. [5]) while exploiting the fact that regular vertices occur only along monotone chains. Instead of processing every regular vertex as an explicit event, we maintain, for each active chain, a pointer to its current edge at the sweep height and update it lazily. Correctness is therefore best stated as an equivalence to the textbook sweep, rather than via ad-hoc visibility regions.

Textbook baseline (edge-based sweep). The classical algorithm maintains a balanced BST of active edges intersecting the sweep line, ordered by their intersection x -coordinate. Each active edge e stores a vertex $\text{helper}(e)$, defined as the most recently processed vertex above the sweep line that can “see” e from the interior. As in the textbook algorithm [5], diagonals are inserted whenever the relevant helper is a *merge* vertex (this can occur at split, merge, end, and certain regular vertices). The resulting diagonal set is known to be valid and to yield a y -monotone decomposition; see [5].

Chain-based representation. Fix a sweep height y between two consecutive event levels (no vertex has y -coordinate exactly y). For a chain C active at height y , let $e(C, y)$ denote the unique edge of C whose y -span contains y (after ADVANCE). Our BST T stores active left-boundary chains ordered by the x -coordinate of $e(C, y)$ at height y ; this is well-defined under general position.

Tie-breaking and degeneracies. Throughout, we assume general position (no two vertices share the same y-coordinate) so that event levels are distinct and each non-extremal vertex is strictly above one neighbor and strictly below the other. If equal y-coordinates are allowed, a standard symbolic perturbation (or lexicographic order by (y, x) together with corresponding non-strict inequalities in Definition 1) can be used; this does not affect the algorithmic structure and is omitted for clarity. We also evaluate $e(C, y)$ at a height y strictly between event levels, so no sweep query is performed exactly at a vertex height.

Definition 5 (Implicit helper state). For an active chain C at sweep height y , define its *implicit helper* vertex

$$ih(C, y) := \text{the upper endpoint of } e(C, y).$$

Intuitively, $ih(C, y)$ is the last vertex of C that the sweep has passed strictly above height y .

Lemma 6 (Advance simulates regular-vertex processing). *Fix any chain C and consider two sweep heights $y_1 > y_2$ with no extrema between them. Let $e_1 = e(C, y_1)$ and $e_2 = e(C, y_2)$ after performing ADVANCE as needed. Then $ih(C, y_2)$ equals the last vertex of C encountered when walking down C from its top endpoint until reaching height y_2 ; moreover, along this descent, every regular vertex of C is encountered exactly once over the entire sweep. If ADVANCE emits a diagonal at a regular vertex (as in Section 3.2), that diagonal coincides with the diagonal inserted by the textbook sweep at that regular vertex (to a pending merge helper).*

Proof. By definition, ADVANCE walks the chain pointer $C.curr$ monotonically downward along C , stopping once the current edge spans the query height. Therefore the upper endpoint of the resulting current edge is exactly the last chain vertex above the height, i.e., $ih(C, y_2)$. Since $C.curr$ only moves forward (down the chain), each vertex of C can become an upper endpoint of $C.curr$ at most once, implying the claimed “encountered once” property.

The only diagonals emitted inside ADVANCE are of the form $(u, C.pending)$ at a regular vertex u with $\text{InteriorRight}(u)$. By Definition 7, $C.pending$ represents the textbook merge-helper state of the corresponding active edge on this chain. The textbook sweep inserts exactly this diagonal at u in the regular-vertex case when that helper is a merge vertex [5]. \square

Definition 7 (Pending as “merge-helper”). In the textbook sweep, the only helper values that trigger a diagonal later are merge vertices. We encode this by storing, for each active chain C , a field $C.pending$ which is either null or a merge vertex. We interpret

$$C.pending \neq \text{NULL} \iff \text{helper}(e(C, y)) \text{ is a merge vertex, and equals } C.pending.$$

If $C.pending = \text{NULL}$, the role of $\text{helper}(e(C, y))$ is played by the implicit helper $ih(C, y)$ from Definition 5.

Lemma 8 (Equivalence of diagonals). *Let D_{chain} be the set of diagonals produced by Algorithm 1 together with any diagonals emitted inside ADVANCE. Let D_{textbook} be the set of diagonals produced by the textbook edge-based sweep on the same polygon and event order. Then $D_{\text{chain}} = D_{\text{textbook}}$.*

Proof. We compare Algorithm 1 to the textbook sweep run on the same polygon, with the same decreasing-y event order.

Step 1: Ordering agreement. Fix any height y strictly between consecutive event levels. For every active chain C , the edge $e(C, y)$ intersects the sweep line in exactly one point. By definition of our BST keys, the in-order ordering of chains in T is the left-to-right ordering of these intersection points, hence agrees with the textbook ordering of the corresponding active edges. Therefore, when Algorithm 1 takes “the predecessor of v in T ”, it identifies the same geometric object as “the active edge immediately left of v ” in the textbook sweep.

Step 2: Helper agreement on regular vertices via ADVANCE. The textbook sweep updates $\text{helper}(e)$ at regular vertices when the current active edge on a chain changes to the next edge below, and it may also add a diagonal at a regular vertex when the relevant helper is a merge vertex. Our algorithm does not process those regular vertices as events, but it does update the current edge implicitly: whenever T needs the key of a chain at some height y , it calls ADVANCE so that the stored edge equals $e(C, y)$. By Lemma 6, the vertex $ih(C, y)$ equals the last vertex encountered when walking down the chain to height y , which is exactly the vertex that the textbook sweep would have last assigned as helper on

the current edge of that chain unless that helper is a merge vertex awaiting a future diagonal. We encode precisely this exception via Definition 7. When $helper(e)$ is a merge vertex and the textbook sweep would add a diagonal at a regular vertex, our ADVANCE emits the same diagonal using $C.pending$ and clears it (see Section 3.2).

Step 3: Case analysis at extrema events. Let v be the next event vertex processed.

- **Start.** In the textbook sweep, a start vertex inserts its outgoing edge(s) into the status structure and sets their helpers to v . In our representation, the unique left-boundary chain originating at v is inserted into T and its pointer is initialized so that $ih(C, y) = v$ for heights just below v ; also $C.pending$ is null. Thus the (merge-)helper information agrees.
- **End.** In the textbook sweep, the edge terminating at an end vertex is removed, and if its helper is a merge vertex then a diagonal is added from v to that helper. In our representation, we remove the terminating left-boundary chain R ; if $R.pending \neq \text{NULL}$ we add the same diagonal before removal. Hence the diagonal action agrees.
- **Split.** In the textbook sweep, at a split vertex v one finds the active edge e immediately left of v and adds a diagonal from v to $helper(e)$; then $helper(e)$ is set to v , and the new edge leaving v is inserted with helper v . In our representation, L is the predecessor chain (Step 1), so its current edge equals $e(L, v.y)$; by Step 2, the helper used by the textbook sweep on that edge is either $L.pending$ (if it is a merge) or $ih(L, v.y) = L.curr.upper$ (otherwise). Algorithm 1 adds exactly the corresponding diagonal, clears $L.pending$ if it was used, and inserts the new left-boundary chain from v with implicit helper v . Thus the diagonal target agrees.
- **Merge.** In the textbook sweep, at a merge vertex v one may add a diagonal to the helper of the edge terminating at v if that helper is a merge, removes that edge, then finds the edge immediately left of v and may add a diagonal to its merge helper, finally setting the helper of the left edge to v . In our representation, we (i) remove the chain terminating at v after emitting a diagonal to its pending merge (if any), (ii) emit a diagonal to the pending merge of the immediately-left chain (if any), and (iii) set that chain's pending to v . Therefore the diagonal targets agree.

□

Theorem 9 (Correctness). *Algorithm 1 produces a valid set of non-crossing diagonals partitioning P into y -monotone subpolygons.*

Proof. By Lemma 8, Algorithm 1 outputs exactly the same diagonals as the textbook monotone-decomposition sweep, interpreted through our chain representation. The textbook 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 10 (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 11 (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 12 (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 adjacency structure takes $O(n + |D|) = O(n)$ time by Lemma 11. 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 adjacency structure for face extraction uses $O(n + |D|) = O(n)$ space. Total: $O(n)$. \square

Lemma 13 (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*

$$s = m + 1 \quad \text{and} \quad e = t + 1.$$

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

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

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

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

As y decreases, $P_{\geq y}$ 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 $P_{\geq y}$ 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 $P_{\geq y}$ for y just above $y(v)$; when y passes below $y(v)$, the point $v \in P_{\geq y}$ connects those components, and $c(y)$ decreases by 1. For split, end, and regular vertices, no new component of $P_{\geq y}$ 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 $e = t + 1$. \square

Relationship to reflex-sensitive bounds. By Lemma 13, $k \leq r + 1$, 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.

Constant factors in the $k \log k$ term. To compare practical performance between instance-sensitive sweeps, it is useful to separate (i) the number of balanced-tree operations from (ii) the cost of an order comparison in the tree. In our method, the balanced structure T stores only *active chains* and is queried only at local extrema and split/merge vertices. Consequently, the number of tree operations is *linear in k* : each event performs one predecessor query in T and $O(1)$ updates (insert/delete of at most a constant number of chains), and there are exactly $2k$ events. Thus, the total number of tree operations is bounded by $c_T k + O(1)$ for a small constant c_T . Each tree operation performs $O(\log k)$ key comparisons.

The key comparison between two active chains is where our constant-factor savings arise. In a classical edge-based sweep (as in Hertel–Mehlhorn [9]), the ordering predicate evaluates the x -coordinate of each active edge at the current sweep level y , which involves arithmetic on the two edge endpoints (in particular, at least one division per evaluation). In our chain representation, comparisons advance chain pointers *lazily*: when the sweep level descends, a chain pointer moves only forward along the boundary, and each boundary vertex is advanced at most once over the entire run. This turns the potentially expensive re-evaluation of edge intersections into an amortized $O(1)$ pointer update stream, keeping the constant in front of $k \log k$ dominated by the tree’s comparison count rather than repeated geometric recomputation.

6. Experimental Evaluation

We implemented the algorithm in C++ and compared against: Garey et al. [8] (monotone decomposition via the `polypartition` library), Hertel–Mehlhorn [9] (approximate convex partition via CGAL `approx_convex_partition_2`, then fan triangulation of each convex piece), and Seidel [13] (incremental randomized triangulation). All experiments were run with GCC at `-O3`. To prevent pathological slowdowns from dominating wall-clock time, each run was capped with a **10 s timeout** (missing entries are shown as “–”).

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. We report *mean \pm standard deviation* of the *elapsed time* (ms) printed by each executable for each (family, n) configuration (seeds 0, …, 4), with `polygons_per_config=5`. We perform one unrecorded warm-up run per executable before collecting timings to reduce cold-start bias from code paging and dynamic loader effects. To reduce systematic warm/cold bias and cache interference from extremely slow baselines, we benchmark in blocks per (family, n): we run the fast algorithms (ours, Garey) across all seeds first, then run the slower baselines (Seidel, Hertel–Mehlhorn). All polygon families are generated deterministically from the seed and n , followed by a fixed rotation to avoid accidental equal- y degeneracies. The `RANDOM` family uses the same generator as `scripts/generate_polygons.py`: vertices are sampled by sorting random angles and assigning random radii (yielding a simple star-shaped polygon). The `CONVEX` family uses seeded affine images of a regular n -gon, and `DENT` pulls one seeded vertex inward (seeded location and depth). The event parameter k shown in the tables is computed directly from each polygon as the number of local maxima, and is reported as mean \pm stdev across instances.

Correctness checks. For every benchmark run, our implementation either (i) outputs exactly $n - 2$ triangles or (ii) fails explicitly (no hidden fallbacks). In addition to this necessary sanity check, we include a diagonal-validity checker in the accompanying reproducibility package (`check_diagonal_validity.py`) which samples generated polygons and verifies that the decomposition diagonals do not properly intersect the boundary or each other. The `reproduce.sh` script runs this check before benchmarking.

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, 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; CGAL’s Hertel–Mehlhorn convex partition baseline is substantially slower than the monotone-decomposition baselines in our setup.

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

Type	n	k	Ours	Seidel	Garey	Hertel–Mehlhorn
Convex	500	1 \pm 0	0.01 \pm 0.00	54.62 \pm 4.03	0.23 \pm 0.14	1.02 \pm 0.04
	1,000	1 \pm 0	0.03 \pm 0.01	54.74 \pm 4.06	0.37 \pm 0.17	3.25 \pm 0.19
	2,000	1 \pm 0	0.03 \pm 0.01	54.79 \pm 3.56	0.62 \pm 0.03	11.42 \pm 1.64
	5,000	1 \pm 0	0.07 \pm 0.01	53.43 \pm 3.90	1.74 \pm 0.13	54.22 \pm 2.28
	10,000	1 \pm 0	0.17 \pm 0.03	51.61 \pm 2.04	4.15 \pm 1.59	228.47 \pm 15.83
Dent	500	2 \pm 0	0.10 \pm 0.02	52.71 \pm 5.54	0.23 \pm 0.16	1.18 \pm 0.44
	1,000	2 \pm 0	0.16 \pm 0.02	50.22 \pm 2.72	0.33 \pm 0.02	2.79 \pm 0.13
	2,000	2 \pm 0	0.34 \pm 0.17	44.98 \pm 3.38	0.59 \pm 0.05	8.99 \pm 0.86
	5,000	2 \pm 0	0.69 \pm 0.12	52.17 \pm 7.80	1.77 \pm 0.53	42.03 \pm 3.29
	10,000	2 \pm 0	1.90 \pm 0.70	61.40 \pm 4.54	4.95 \pm 1.54	180.82 \pm 33.46
Random	500	160 \pm 2	0.22 \pm 0.02	46.98 \pm 2.25	0.28 \pm 0.02	2.72 \pm 0.45
	1,000	329 \pm 4	0.41 \pm 0.03	45.18 \pm 10.04	0.61 \pm 0.06	7.72 \pm 2.76
	2,000	666 \pm 9	0.78 \pm 0.06	49.52 \pm 6.31	1.06 \pm 0.11	17.62 \pm 2.69
	5,000	1661 \pm 17	2.38 \pm 0.14	53.72 \pm 3.38	3.38 \pm 0.19	107.60 \pm 4.81
	10,000	3316 \pm 12	5.36 \pm 0.35	65.52 \pm 8.93	7.62 \pm 1.07	415.67 \pm 34.07

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]	$O(n \log n)$	via convex partition
Seidel [13]	$O(n \log^* n)$ exp.	randomized
Chazelle [3]	$O(n)$	impractical
This paper	$O(n + k \log k)$	instance-sensitive (in #extrema k)

Our algorithm achieves the instance-sensitive bound $O(n + k \log k)$, where k is the number of local extrema 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. Improving constants in the high-event regime without sacrificing the k -sensitive structure is a key practical direction.

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. Icerpi. 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.