Planar lower envelope of monotone polygonal chains

Daniel L. Lu
Carnegie Mellon University, Robotics Institute, 5000 Forbes Ave, Pittsburgh, PA 15213

Abstract
A simple linear search algorithm running in $O(n + mk)$ time is proposed for constructing the lower envelope of $k$ vertices from $m$ monotone polygonal chains in 2D with $n$ vertices in total. This can be applied to output-sensitive construction of lower envelopes for $n$ arbitrary line segments in optimal $O(n \log k)$ time, where $k$ is the output size. Compared to existing output-sensitive algorithms for lower envelopes, this is simpler to implement, does not require complex data structures, and is a constant factor faster.

Keywords: computational geometry; upper envelope; lower envelope; output-sensitive algorithms

1. Introduction
Finding the lower envelope of line segments in 2D is useful in visibility problems in robotics, facility location, architecture, video games, and computer graphics. The lower envelope of $n$ segments has a complexity bounded by the third order Davenport–Schinzel sequence, which is in $O(\alpha(n))$ where $\alpha(n)$ is the slow-growing inverse Ackermann function. A worst-case optimal divide-and-conquer algorithm running in $O(n \log n)$ time was proposed by Hershberger [2] as a refinement of earlier $O(n \alpha(n) \log n)$ time methods.

Since a visibility polygon can be considered a lower envelope in polar coordinates centered on the query point, a lower envelope algorithm can be applied to finding visibility polygons. Some visibility polygon problems are posed with assumptions which allow faster or simpler algorithms to be used. For a visibility polygon of $n$ segments allowed to intersect at only their endpoints, $O(n \log n)$ time algorithms simpler than Hershberger’s algorithm are known, based on either divide-and-conquer or angular plane sweep [5, 6]. For point visibility in a simple polygon, a linear time algorithm suffices [7, 8]. For point visibility in a complex polygon with $h$ holes and $n$ vertices, a worst-case optimal $O(n + h \log h)$ time algorithm using Chazelle’s linear time triangulation is known [9]. Some algorithms using preprocessing have been proposed, involving various tradeoffs between preprocessing time, query time, and storage space [10, 11, 12].

Optimal output-sensitive $O(n \log k)$ time algorithms for lower envelopes, where $k$ is the output size, were proposed by Chan [1] and Nielsen and Yvinec [3]. Both use Hershberger’s algorithm as a subroutine. Parallelized versions of Nielsen and Yvinec’s method have been proposed, including a deterministic $O(\log n (\log k + \log \log n))$ time algorithm and faster randomized variants, each doing a total of $O(n \log k)$ work [4]. However, these all require complex data structures such as lazy interval trees. This paper presents a simple extension of the Jarvis march, similar to Welzl’s observation in Idea 5 of Chan’s paper [1], to monotone polygonal chains. Combined with any $O(n \log n)$ time algorithm, this allows a simpler implementation of Chan’s algorithm running in deterministic $O(n \log k)$ time without preprocessing or parallelization.

Although output-sensitive algorithms for computing lower envelopes have been proposed, no known implementation exists; however, efficient implementations of $O(n \log n)$ time visibility polygon algorithms are available in the Computational Geometry Algorithms Library (CGAL) [13] as well as elsewhere [14]. These can be easily adapted using my algorithm to obtain an optimal $O(n \log k)$ running time. An implementation of a divide-and-conquer lower envelope algorithm running in $O(n \alpha(n) \log n)$ time is also available in the CGAL [15]. Using this instead of an $O(n \log n)$ time subroutine yields an overall $O(n \alpha(k) \log k)$ time complexity.

Email address: dllu@cmu.edu (Daniel L. Lu)
2. Lower envelope of monotone polygonal chains

Suppose there are \( m \) polygonal chains \( V_0, \ldots, V_{m-1} \) monotone with respect to the vertical axis, with a total of \( n \) vertices, where the \( j \)th vertex of the \( i \)th chain is denoted as \( V_i(j) = (V_i(j)_x, V_i(j)_y) \) such that \( V_i(j)_x \leq V_i(j+1)_x \). For simplicity, we assume no vertex lies on a segment from a different chain (without this assumption, we would need to compare slopes in addition to intersections). A lower envelope is calculated by following the active chain, denoted \( V_{\text{active}} \), until some other chain intersects it. To efficiently find such intersections, we maintain a pointer \( p_i \) on the chain \( V_i \), which only advances and never retreats.

On each iteration, a segment is added to the lower envelope currently being constructed. Consider the active segment \( S^* = (V_{\text{active}}(p^*-1), V_{\text{active}}(p^*)) \). Suppose it has been shown that the left endpoint of \( S^* \) is not occluded. Then, for each \( i = 0, \ldots, m-1 \), the pointer \( p_i \) is incremented until either \( V_i(p_i)_x \geq V_{\text{active}}(p^*)_x \) or the segment \( S_i = (V_i(p_i-1), V_i(p_i)) \) intersects the segment \( S^* \) or \( (V_{\text{active}}(p^*-1), V_{\text{active}}(p^*-1)_x, -\infty) \) — otherwise, it would imply \( S_i \) is occluded by a combination of \( S^* \) and the then-active segments of all previous iterations. While these pointers are being incremented, the first intersection with \( S^* \) is found. The segment from \( V_{\text{active}}(p^*-1) \) to the intersection point is added to the lower envelope, and then the next \( S^* \) is set to the segment from the intersection point to the end of the intersecting segment. This is repeated until the entire envelope has been found. A pseudocode is shown in the Appendix.

The total number of increments of pointers \( p_i \) is \( O(n) \) since they never retreat; the comparisons on each increment are done in constant time. Each time a new vertex is added to the output, the algorithm loops over all \( m \) polygonal chains to check for intersection with the currently examined edge and to test if the associated pointer needs to be incremented. Since the output has \( k \) vertices, this takes \( O(mk) \) time. The total time complexity is thus \( O(n + mk) \). The algorithm adapts easily to polar coordinates to find the intersection of star-shaped polygons (visibility polygons) or the lower envelope of arbitrary piecewise functions composed of pieces which intersect at most once.

3. Output-sensitive construction of lower envelopes

Here, a straightforward application of Chan’s algorithm \([1]\) is described.

```plaintext
1: for \( t = 0, 1, 2, \ldots \) do
2: \( \kappa \leftarrow 2^t \)
3: Arbitrarily partition segments into \( \lceil \frac{n}{\kappa} \rceil \) subsets each of size at most \( \kappa \)
4: Run any \( O(n \log n) \) time algorithm on each group, yielding \( \lceil \frac{n}{\kappa} \rceil \) monotone polygonal chains
5: Find the lower envelope of these monotone polygonal chains, and abort if the output size exceeds \( \kappa \)
6: end for
```

The time complexity is analyzed here per iteration:

- Running a \( O(n \log n) \) time algorithm (such as Hershberger’s algorithm \([2]\), or, for non-intersecting segments, one of the algorithms implemented in \([13]\)) on each group takes \( O(\kappa \log \kappa) \) time each, for a total of \( O(n \log \kappa) \). The number of vertices in each of the \( \lceil \frac{n}{\kappa} \rceil \) chains is in \( O(\kappa \alpha(\kappa)) \).
- Finding the lower envelope takes \( O(n \alpha(\kappa)) \) time since the total number of vertices in the \( \lceil \frac{n}{\kappa} \rceil \) chains is \( O(n \alpha(\kappa)) \) and the algorithm immediately aborts when the output exceeds size \( \kappa \).

In particular, the second term of time complexity \( O(n \alpha(\kappa)) \) is nearly a log factor improvement upon Chan’s ray-shooting method \([1]\). Ultimately each iteration runs in \( O(n \log \kappa) = O(n \kappa^t) \), dominated by the first term. The total time complexity of the \( \lceil \log \log k \rceil \) iterations is as desired:

\[
O \left( \sum_{t=1}^{\lceil \log \log k \rceil} n 2^t \right) = O \left( n 2^\lceil \log \log k \rceil + 1 \right) = O(n \log k).
\]

It is easy to see that, if the \( O(n \log n) \) time algorithm used in step 3 was replaced with one running in \( O(n \alpha(n) \log n) \) time such as \([13]\), then the overall time complexity increases to \( O(n \alpha(k) \log k) \).
References

[1] Chan, Timothy M. “Optimal output-sensitive convex hull algorithms in two and three dimensions.” *Discrete & Computational Geometry* 16.4 (1996): 361-368.

[2] Hershberger, John. “Finding the lower envelope of \( n \) line segments in \( O(n \log n) \) time.” *Information Processing Letters* 33.4 (1989): 169-174.

[3] Nielsen, Franck, and Mariette Yvinec. “An output-sensitive convex hull algorithm for planar objects.” *International Journal of Computational Geometry & Applications* 8.01 (1998): 39-65.

[4] Gupta, Neelima, and Sumit Chopra. “Output-sensitive algorithms for optimally constructing the upper envelope of straight line segments in parallel.” *Journal of Parallel and Distributed Computing* 67.7 (2007): 772-782.

[5] Asano, Tetsuo. “An efficient algorithm for finding the visibility polygon for a polygonal region with holes.” *IEICE Transactions (1976-1990)* 68.9 (1985):557-559.

[6] Suri, Subhash, and Joseph O’Rourke. “Worst-case optimal algorithms for constructing visibility polygons with holes.” *Proceedings of the second annual symposion on Computational geometry*. ACM, 1986.

[7] El Gindy, Hossam, and David Avis. “A linear algorithm for computing the visibility polygon from a point.” *Journal of Algorithms* 2.2 (1981): 186-197.

[8] Joe, Barry, and R. B. Simpson. “ Corrections to Lee’s visibility polygon algorithm.” *BIT Numerical Mathematics* 27.4 (1987):458-473.

[9] Hefferman, Paul J., and Joseph SB Mitchell. “An optimal algorithm for computing visibility in the plane.” *SIAM Journal on Computing* 24.1 (1995): 184-201.

[10] Bose, Prosenjit, Anna Lubiw, and J. Ian Munro. “Efficient visibility queries in simple polygons.” *Computational Geometry* 23.3 (2002): 313-335.

[11] Aronov, Boris, et al. “Visibility queries and maintenance in simple polygons.” *Discrete & Computational Geometry* 27.4 (2002): 461-483.

[12] Zarei, Alireza, and Mohammad Ghodsi. “Query point visibility computation in polygons with holes.” *Computational Geometry* 39.2 (2008): 78-90.

[13] Bungiu, Francisc, Michael Hemmer, John Hershberger, Kan Huang, and Alexander Kröller. “Efficient Computation of Visibility Polygons.” *European Workshop on Computational Geometry* (2014).

[14] Knoll, Byron. “visibility-polygon-js: JavaScript Visibility Polygon Library”. [https://code.google.com/p/visibility-polygon-js/](https://code.google.com/p/visibility-polygon-js/) (2014). Accessed 2015-03-13.

[15] Wein, Ron. 2D envelopes. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.5.2 edition, 2015.
Appendix: Pseudocode

Here is a pseudocode description of the linear search algorithm for finding the lower envelope of monotone polygonal chains $V_0, \ldots, V_{m-1}$. An animated explanation can be found at http://www.dllu.net/present_visibility/envelope.html with a C++11 implementation at http://www.dllu.net/present_visibility/implementation.cpp

1: $p_0, \ldots, p_{m-1} \leftarrow 1$
2: $p^* \leftarrow 1$
3: active $\leftarrow$ the index of the chain with the smallest $y$-coordinate of the first vertex
4: $V \leftarrow \{V_{\text{active}}(0)\}$
5: repeat
6: if $p^* < k$ then
7: \hspace{1em} $p_{\text{active}} \leftarrow p^* + 1$
8: \hspace{1em} end if
9: \hspace{1em} best $\leftarrow (\infty, \infty)$, next $\leftarrow -1$
10: for $i = 0, \ldots, m-1$ do
11: \hspace{1em} while $V_i(p_i)x < V_{\text{active}}(p^*)x$ and $(V_i(p_i-1), V_i(p_i))$ does not intersect either $(V_{\text{active}}(p^*-1), V_{\text{active}}(p^*))$
12: \hspace{2em} or $(V_{\text{active}}(p^*-1), (V_{\text{active}}(p^*)_x, -\infty))$ do
13: \hspace{3em} $p_i \leftarrow p_i + 1$
14: \hspace{1em} end while
15: if $(V_i(p_i-1), V_i(p_i))$ intersects $(V_{\text{active}}(p^*-1), V_{\text{active}}(p^*))$ at a point $I$ such that $I_x < \text{best}_x$ then
16: \hspace{1em} best $\leftarrow I$, next $\leftarrow i$
17: \hspace{1em} end if
18: end for
19: if next $= -1$ then
20: \hspace{1em} Append $V_{\text{active}}(p^*)$ to $V$
21: \hspace{1em} break
22: else if next $\neq \text{active}$ then
23: \hspace{1em} $p_{\text{active}} \leftarrow p^*$
24: \hspace{1em} $V_{\text{next}}(p_{\text{next}}-1) \leftarrow \text{best}$
25: \hspace{1em} end if
26: Append best to $V$
27: active $\leftarrow$ next, $p^* \leftarrow p_{\text{active}}$
28: until no pointers can be incremented
29: return $V$