Algorithmica, vol. 8 (1992), pp. 345-364.

Constructing Strongly Convex Hulls Using Exact or Rounded Arithmetic

Zhenyu Li and Victor Milenkovic
Harvard University

Abstract

One useful generalization of the convex hull of a set S of n points is the *ϵ-strongly convex δ-hull.* It is defined to be a convex polygon $\mathcal{P}$ with vertices taken from S such that no point in S lies farther than δ outside $\mathcal{P}$ and such that even if the vertices of $\mathcal{P}$ are perturbed by as much as ϵ, $\mathcal{P}$ remains convex. It was an open question[1] as to whether an ϵ-strongly convex $\mathrm{O}(\epsilon)$-hull existed for all positive ϵ. We give here an $\mathrm{O}(n \log n)$ algorithm for constructing it (which thus proves its existence). This algorithm uses exact rational arithmetic. We also show how to construct an ϵ-strongly convex $\mathrm{O}(\epsilon + \mu)$-hull in $\mathrm{O}(n \log n)$ time using rounded arithmetic with rounding unit μ. This is the first rounded arithmetic convex hull algorithm which guarantees a convex output and which has error independent of n.

1 Introduction

Recently, some work in the area of computational geometry has focused on the numerical issues that arise when geometric algorithms are implemented using rounded floating point arithmetic. Unlike other types of numerical algorithms, most geometric algorithms require that all arithmetic be performed over the field of reals or rationals, and they behave erratically when implemented with a non-associative and non-distributive number system such as rounded arithmetic. The term *robust* has arisen to describe algorithms whose correctness is not spoiled by round-off error. The assumption is we do not "cheat": each real variable in the algorithm is replaced by a floating point variable, and each real addition, subtraction, multiplication and division is replaced by a single corresponding floating point operation. Clearly, robust algorithms are of great practical interest.

Robust algorithms have been devised for constructing arrangements of lines [6, 7] or algebraic curves [5], convex hulls and triangulations of planar point-sets [1], and Voronoi diagrams [8]. As one would expect, these algorithms generate objects which are near but not equal to objects constructed using exact arithmetic. As one does not expect, the robustly generated objects are not actually line or algebraic curve arrangements, convex hulls, triangulations, or Voronoi diagrams. For example, Fortune's convex hull algorithm constructs a polygon which is only $\mathrm{O}(\mu)$-*weakly convex*, meaning that each vertex may have to be perturbed by as much as $c\mu$ in order to eliminate all reflex angles, where μ is the *rounding unit* (defined in Section 3.1) and c is a small constant depending on the implementation. Similarly, for the other robust algorithms, the constructed object must be perturbed by some amount in order to make it into the type of object it is purported to be.

[1]The notion of an ϵ-strongly convex δ-hull and the issues of its existence and construction were introduced by Guibas *et al.* [3], although the notation used here is slightly different.

Paradoxically, if the constructed objects become inputs to other robust algorithms, we may desire that the objects have *more* definition than exactly constructed objects, not less. For example, intersecting two polygons or performing a Minkowsky sum on two polygons is easier if the polygons are convex. For reasons given in Section 5, if these operations are also to be robust, we might wish that all the angles in the polygons be less than $\pi - \alpha$ radians for some positive angle α determined by the operation, a condition not even satisfied by the exact convex hull. As another example, performing a Euclidean transformation or some other operation in floating point perturbs the relative positions of the vertices. We might desire that the input polygon be *ϵ-strongly convex*, meaning that it remains convex even if its vertices are arbitrarily perturbed by as much as some positive distance ϵ, an upper bound on the error introduced by the transformation being performed. The values α or ϵ must be input parameters to any algorithm that constructs a strongly convex polygon. Of course, the strongly convex polygon should be a good approximation to the actual convex hull. In particular, the maximum distance of a point on the actual convex hull to the strongly convex polygon should not be more than a small multiple of ϵ or $\alpha\Delta$, where Δ is the diameter of the polygon.[2]

Naive methods exist for extracting the vertices of an ϵ-strongly convex polygon from a set of n points in the plane, but these lead to a polygon with $n\epsilon$ error (distance from the exact convex hull). Even the existence, for every set of points, of a polygon with error $\mathrm{O}(\epsilon)$ has been an open problem up to now. We settle this question in the affirmative here by giving an $\mathrm{O}(n \log n)$ algorithm for constructing an ϵ-strongly convex polygon with error only a small constant times ϵ. This algorithm is robust, and when it is implemented in floating point arithmetic, the additional error is only a small constant times the rounding unit μ.

In order to compare the theoretical complexity of the robust and the exact algorithms, we assume that the coordinates of the input points are B-bit integers (the robust algorithm converts these integers to floating point before performing additions and multiplications) Both the robust and the exact version perform $\Omega(n \log n)$ comparisons of B-bit integers in order to sort the x-coordinates. This step costs exactly the same for both algorithms. After sorting, both algorithms must perform a linear number of additions and multiplications. In order to avoid rounding in all cases, the exact algorithm must use $(2B+1)$-bit integer arithmetic. The robust algorithm requires only $(B+6)$-bit (mantissa) floating point arithmetic to generate B bits of accuracy in the output. As Section 3 discusses, the approximate algorithm requires a few more arithmetic operations than an exact algorithm. Break-even occurs when $(2B+1)$-bit arithmetic is twice as expensive as $(B+6)$-bit arithmetic. On a typical computer today, if the inputs are integers, the exact algorithm can use double precision floating point operations, and there will be not much difference in cost between the robust and the exact algorithm. However, if the inputs are double precision and if double precision accuracy is desired, then the arithmetic cost of the exact algorithm may be at least ten times greater than the robust algorithm. The improved efficiency of the robust algorithm occurs only after sorting, which dominates the cost. However, since sorting is a necessary step, this is the best improvement possible for the construction of convex hulls.

As the previous paragraph describes, our robust convex hull algorithm has only a modest improvement in running time over the exact algorithm. There are a number of reasons why even this modest improvement is important. First, our robust algorithm achieves the full amount of improvement possible for the convex hull problem. Second, since we are trying to develop a theory of robust geometry, we must solve the simple problems before we can understand the difficult ones. For difficult constructions in higher dimensions or with greater algebraic degree, the cost differential between a rounded and exact solution is much greater. For the convex hull problem, we develop new techniques, trimming and spine construction (Sections 2 and 4.1), which may be more generally applicable.

1.1 Definitions and Lemmas

The following are the definitions used in this paper.

Definition 1 *A polygon $\mathcal{P}$ is a δ-*hull *of a set of points S if all vertices of $\mathcal{P}$ belong to S and if no point of S is farther than δ from the polygonal region bounded by $\mathcal{P}$.*

[2]Absolute error is scale-dependent, and so it is necessary to multiply α by Δ.

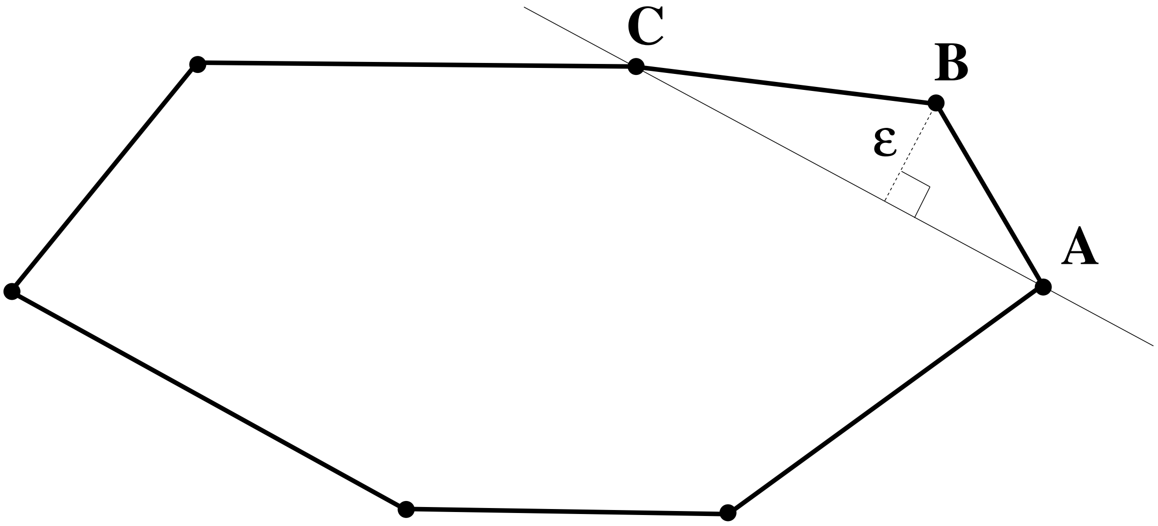

Figure 1: ϵ-Strongly Convex Vertex

A polygon $\mathcal{P}$ is ϵ-weakly convex *($\epsilon \geq 0$) if there exists some way of perturbing each vertex of $\mathcal{P}$ no farther than ϵ so that $\mathcal{P}$ becomes convex.*

A polygon $\mathcal{P}$ is ϵ-strongly convex *($\epsilon \geq 0$) if $\mathcal{P}$ is convex and remains convex even after each vertex of $\mathcal{P}$ is perturbed as far as ϵ.*

The following definition and lemma reduce strong convexity to a local property. The signed distance $\delta(\mathbf{B}, \mathbf{AC})$ from $\mathbf{B}$ to line $\mathbf{AC}$ is positive when trangle $\mathbf{ABC}$ has clockwise orientation.

Definition 2 *Let $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ be consecutive vertices, in counter-clockwise order, of a simple polygon $\mathcal{P}$. We say that vertex $\mathbf{B}$ is* ϵ-strongly convex *if the distance $\delta(\mathbf{B}, \mathbf{CA})$ from $\mathbf{B}$ to line $\mathbf{CA}$ is greater than ϵ (see Figure 1).*

No matter where we move $\mathbf{B}$ inside a disk of radius ϵ, it cannot cross line $\mathbf{AC}$, and angle $\mathbf{ABC}$ remains convex.

Lemma 1 *If each vertex of a polygon $\mathcal{P}$ is 2ϵ-strongly convex, then $\mathcal{P}$ is ϵ-strongly convex.*

In other words, if we can move any vertex 2ϵ, then we can move all of them ϵ. The converse of this lemma is false, but it is tight in the sense that 2ϵ is the smallest value that will make this lemma hold.

Proof. Our proof is by contradiction. Assume all vertices are 2ϵ-strongly convex but $\mathcal{P}$ is not ϵ-strongly convex. Then there must be some ϵ-perturbation of the vertices that makes some vertex $\mathbf{B}$ non-convex. Let $\mathbf{A}$ and $\mathbf{C}$ be the neighboring vertices. It is easy to show that all three vertices of triangle $\mathbf{ABC}$ are 2ϵ-strongly convex *with respect to that triangle*. There must be some slightly smaller perturbation transforming triangle $\mathbf{ABC}$ into a degenerate triangle $\mathbf{A}'\mathbf{B}'\mathbf{C}'$ where, without loss of generality, $\mathbf{B}'$ lies on segment $\mathbf{A}'\mathbf{C}'$. The perturbed vertices $\mathbf{A}'$ and $\mathbf{C}'$ lie within ϵ of line $\mathbf{AC}$. Therefore $\mathbf{B}'$ lies within ϵ of line $\mathbf{AC}$. But $\mathbf{B}$ lies within ϵ of $\mathbf{B}'$, and thus $\mathbf{B}$ lies within 2ϵ of line $\mathbf{AC}$. This contradicts the fact that $\mathbf{B}$ is 2ϵ-strongly convex. □

1.2 Results

Section 2 shows how to construct an ϵ-strongly convex 12ϵ-hull of a set of points S using exact arithmetic, where ϵ is any non-negative value. Section 3 defines a rounding unit μ which depends on the diameter Δ

of the set of input points S and on the precision used to express their coordinates. This section also shows how to compute angles and distances accurately using rounded arithmetic. Section 4 shows how to modify the algorithm of Section 2 to construct a ϵ-strongly convex $(12\epsilon + 54\mu)$-hull using rounded floating point arithmetic. If we require only that all the angles be less than $\pi - \epsilon/\Delta$ radians, then the same algorithm can generate a $(4\epsilon + 36\mu)$-hull that has this property. Both algorithms run in time $O(n)$, after sorting, where n is the number of points in S. Recall that exact methods require $(2B + 1)$-bit arithmetic to achieve B bits of accuracy. Since $\log_2 54 < 6$, the robust algorithm requires only $(B + 6)$-bit arithmetic as claimed.

As stated above, it was not known until now whether an ϵ-strongly convex $c\epsilon$-hull exists for every ϵ, where c is some constant independent of ϵ and n. The first algorithm gives a proof of existence and a method of construction. The naive approach, simply removing every vertex that is not 2ϵ-strongly convex (see Lemma 1), generates a $2n\epsilon$-hull.

If one sets ϵ to zero, the algorithm of Section 4 becomes the first robust geometric algorithm to construct a convex $O(\mu)$-hull faster than the best exact arithmetic algorithm for the same problem. Fortune's algorithm [1] is slightly faster and has a little less error (6μ), but it only generates a 6μ-weakly convex polygon. Guibas, Salesin, and Stolfi [4] have developed a rounded arithmetic algorithm for constructing strongly convex hulls, but it has running time $O(n^2 \log n)$.

2 Constructing A Strongly Convex Approximate Hull

We will first consider the problem of constructing an ϵ-strongly convex approximation to the convex hull of a set S of n points. We will do this with exact arithmetic and then switch to rounded arithmetic in Section 4.

The first stage is to construct the convex hull $\mathcal{H}$ and throw away all the other points. This stage takes $O(n \log n)$. The rest of the algorithm removes points from $\mathcal{H}$ until it becomes ϵ-strongly convex. This step takes $O(|\mathcal{H}|)$ which is $O(n)$.

We know from Lemma 1 that we can make the polygon ϵ-strongly convex by making each vertex 2ϵ-strongly convex. Thus if $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are consecutive vertices and $\delta(\mathbf{B}, \mathbf{AC}) \geq 2\epsilon$, then we want to mark $\mathbf{B}$ as STRONG and leave it alone (do not remove it). We are free to remove other points because $\mathcal{H}$ is convex, and removing other points can only make $\mathbf{B}$ stronger. Thus, the first thing we do is write a predicate IsStrong($\mathbf{V}$) which checks if a vertex $\mathbf{V}$ is 2ϵ-strong and use it to mark all the appropriate vertices STRONG.

If IsStrong($\mathbf{B}$) is false, then $\mathbf{ABC}$ is nearly a straight angle. (How straight depends on the lengths $|\mathbf{AB}|$ and $|\mathbf{BC}|$.) We want to alter the polygon to make $\mathbf{B}$ 2ϵ-strong. We do this by a process called *trimming*. Essentially what we do is locate points $\mathbf{A}'$ and $\mathbf{C}'$ (not necessarily vertices) on the boundary of $\mathcal{H}$ such that

$$\delta(\mathbf{A}', \mathbf{AB}) \geq 2\epsilon \qquad \text{and} \qquad \delta(\mathbf{C}', \mathbf{BC}) \geq 2\epsilon,$$

and then we replace the section of $\mathcal{H}$ from $\mathbf{A}'$ to $\mathbf{C}'$ by segments $\mathbf{A}'\mathbf{B}$ and $\mathbf{BC}'$ (see Figure 2). This process guarantees that $\mathbf{B}$ becomes 2ϵ-strong, and it only trims off a region of thickness 2ϵ. Thus all points on $\mathcal{H}$ lie within 2ϵ of the trimmed polygon.

There are a number of difficulties with this naive definition of trimming.

- $\mathbf{A}'$ or $\mathbf{C}'$ may not be vertices of the original polygon.
- The vertices of the polygon may be too near or too far from line $\mathbf{AB}$ to act even as approximations to $\mathbf{A}'$.
- Subsequent trimmings may undo the work of the first.
- If we perform multiple trimmings, we may cut away a region more than $c\epsilon$ thick.

We will deal with the first two difficulties by allowing trimming to fail and then fixing things later. We deal with the last two difficulties by first performing a set of non-overlapping *forward* (counter-clockwise)

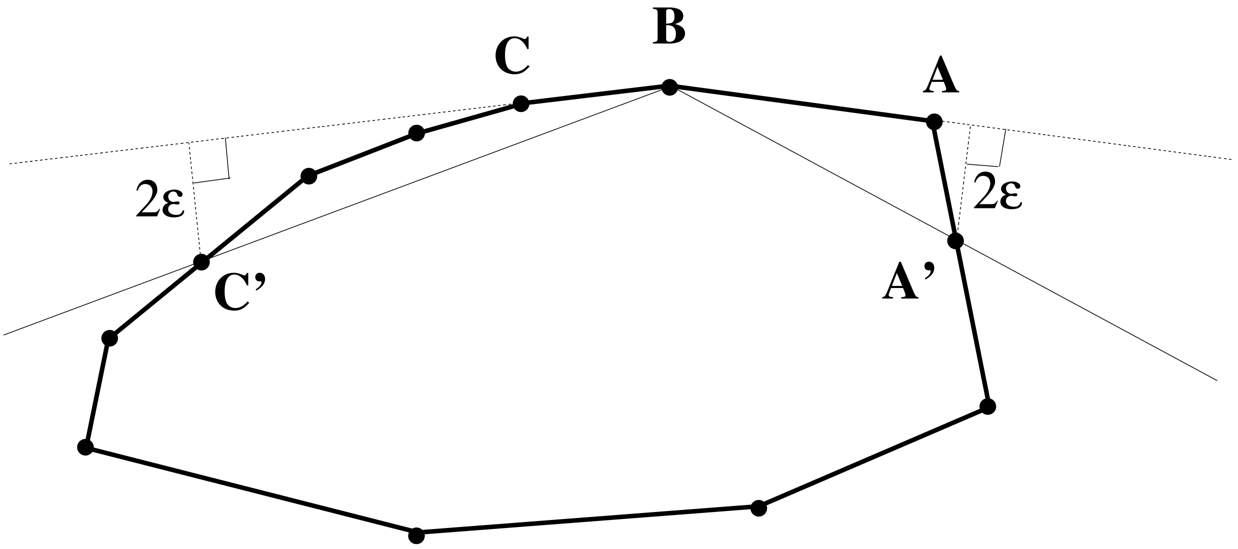

Figure 2: Trimming a Polygon

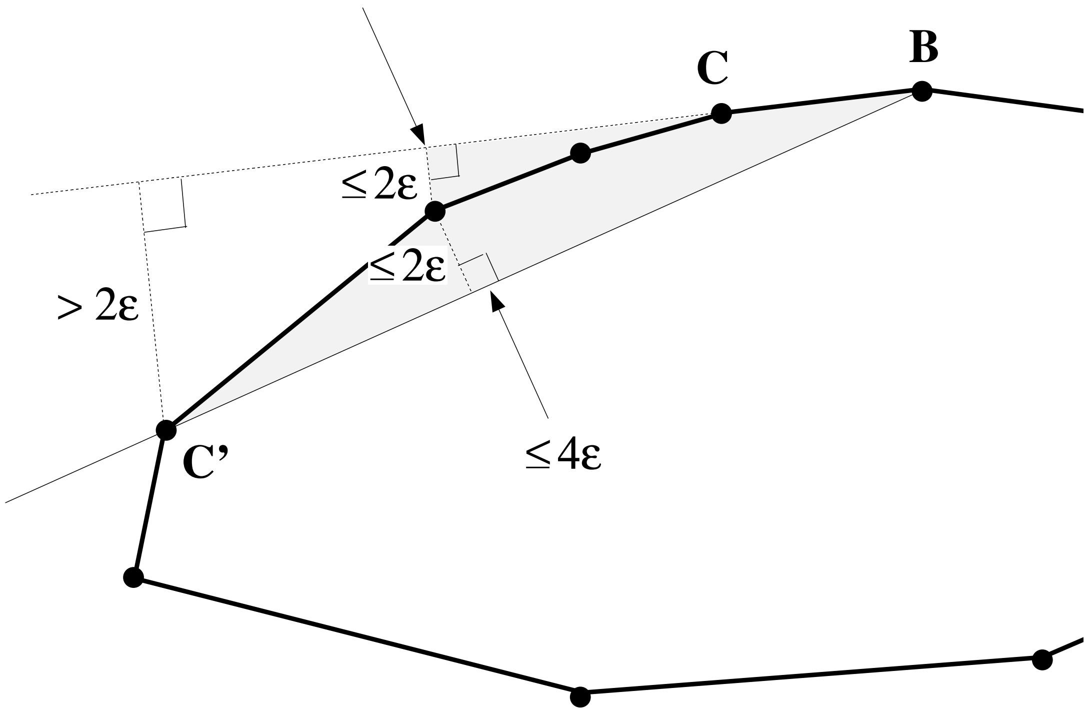

Figure 3: Forward Trimming a Vertex

trimmings and then performing a set of non-overlapping *backward* (clockwise) trimmings. Thus each region is trimmed at most twice.

The following is the actual algorithm we use. We walk around the polygon in the counter-clockwise direction applying the function ForwardTrim($\mathbf{B}$) (given below) to each vertex. This function removes some vertices and marks the ones that remain. We then walk around in the opposite direction applying the function BackwardTrim($\mathbf{B}$) to each vertex $\mathbf{B}$ that is not marked FAILED or STRONG by the first stage. Finally, we remove all the vertices marked FAILED. At this point, all the vertices will be marked STRONG, meaning that they are 2ϵ-strongly convex.

We define the function ForwardTrim($\mathbf{B}$) to remove some of the successors (in the counter-clockwise direction) of vertex $\mathbf{B}$. Let $\mathbf{C}$ be the original successor of $\mathbf{B}$ in $\mathcal{H}$, and let $\mathbf{C}'$ be the current successor of $\mathbf{B}$ after we have removed a number of vertices (see Figure 3). The function ForwardTrim($\mathbf{B}$) removes each successor of $\mathbf{B}$ from the polygon until one of the following occurs:

- IsStrong($\mathbf{B}$) is true;
- IsStrong($\mathbf{C}'$) is true;
- $\mathbf{C}'$ is marked in some way (we have closed the loop);
- $\delta(\mathbf{C}', \mathbf{BC}) > 2\epsilon$.

These conditions are checked in the given order, with the following actions taken: in the first case, $\mathbf{B}$ is marked STRONG; in the middle two, $\mathbf{B}$ is marked FAILED; in the last, $\mathbf{B}$ is marked FTRIMMED.

Lemma 2 *After the forward trimming stage, each vertex of $\mathcal{H}$ lies within 4ϵ of the trimmed polygon.*

Proof. Figure 3 illustrates the case in which we halt the trimming at $\mathbf{C}'$ because of the fourth termination condition. No matter what reason we halt the trimming, we know that the predecessor of $\mathbf{C}'$ lies within 2ϵ of lines $\mathbf{BC}$ and $\mathbf{BC}'$; otherwise, we would have halted one step earlier. The portion of $\mathcal{H}$ from $\mathbf{B}$ to $\mathbf{C}'$ lies inside the shaded region which lies within 4ϵ of segment $\mathbf{BC}'$. As we walk around the polygon, we never remove $\mathbf{BC}'$. Thus we conclude that every point of the original polygon lies within 4ϵ of some segment of the trimmed polygon. □

The function BackwardTrim($\mathbf{B}$) is defined analogously to ForwardTrim. It can remove successors (in the clockwise direction) of $\mathbf{B}$ marked FAILED or FTRIMMED but not STRONG (because of the second termination condition). Naturally, the third termination condition refers to vertices marked in this stage, not the first stage. The main difference is that if the fourth termination condition halts the trimming, then $\mathbf{B}$ is marked STRONG instead of FTRIMMED. We know from our discussion at the beginning of the section that $\mathbf{B}$ will in fact be 2ϵ-strongly convex in this case.

Lemma 2 implies that the first trimmed polygon lies within 4ϵ of the second trimmed polygon, and therefore, all points in $\mathcal{H}$ lie within 8ϵ of the second trimmed polygon.

Recall that the final stage of the algorithm removes all the FAILED vertices at this point, leaving only STRONG vertices. The following lemma gives the error bound for this stage.

Lemma 3 *After forward trimming and backward trimming, there is no string of more than three consecutive FAILED vertices. This implies that removing FAILED vertices adds at most another 4ϵ to the error.*

Proof. After the forward trimming stage, each FAILED vertex is followed by a STRONG one (second termination condition) unless it is the last vertex marked (third termination condition), in which case it may be followed by a FAILED vertex followed by a STRONG one. Thus there may be at most one pair of consecutive FAILED vertices. Backward trimming can add at most one more FAILED vertex clockwise of each vertex marked FAILED by ForwardTrim (because, as before, each FAILED vertex must be followed by a STRONG vertex). At worst it creates one string of three FAILED vertices from the string of two created by the forward trimming.

If **A**, **B**, and **C** are consecutive vertices, and **B** is FAILED, then **B** is not STRONG, and therefore **B** lies within 2ϵ of **AC**. Removing **B** only moves the polygon at most 2ϵ farther from the original polygon. If we remove two or three consecutive FAILED vertices, the errors add, but not more than 4ϵ. For two vertices this is obvious. For three, we first remove the first and third (no overlap) and then remove the second. □

Each stage of the algorithm moves the trimmed polygon at most 4ϵ farther from the original convex hull $\mathcal{H}$. The maximal distance from $\mathcal{H}$ to the final ϵ-strong polygon is 12ϵ. Since all points in S lie inside $\mathcal{H}$, they also satisfy this error bound. This proves the following theorem.

Theorem 4 *Given the convex hull (of size h) of a set S of points, an ϵ-strongly convex 12ϵ-hull can be constructed in time* $\mathrm{O}(h)$.

3 Arithmetic Models and Calculations

Computing convex hulls, strong or otherwise, requires two geometric computations: the angle formed by two line segments, and the distance from a point to a line segment. The cost of calculating these quantities and the accuracy with which we can compute them depends on the input representation and the arithmetic model. We consider three types of representation for the vertex coordinates of the input: fixed point, single precision floating point, and double precision floating point. We consider arithmetic operations on only the latter two types because fixed point operations can be carried out using floating point arithmetic. The size and specific properties of these representations depends on the computer, and so we present a general abstract model of computer arithmetic.

The basic conclusion of this paper is that robust convex hull algorithms can construct convex hulls more efficiently than an exact algorithm can construct them, *unless* the input representation is fixed point and the arithmetic model is double precision floating point. Furthermore, for exact algorithms to have the advantage even in these most favorable circumstances, double precision addition and multiplication cannot cost more than three times as much as the single precision versions.

3.1 Arithmetic Model

Fixed point representation is essentially integer representation with the implicit understanding that each number is multiplied by some fixed power of two. *Floating point* representations contain an explicit exponent field as well as mantissa. The value represented by a floating point number with mantissa m and exponent e is $2^e m$.

The fixed or integer representation has B_I bits; the single precision floating point mantissa has B_S bits; and the double has a B_D-bit mantissa. For a typical computers such as a Vax or Sun or PC, B_I is 16 or 32, B_S is 21 and B_D is 53. We make the reasonable assumption that $B_S < 2B_I$ and $B_D > 2B_S$. In words, single precision is not twice as big as an integer, but double precision is at least twice as big as single precision.

Mathematically, the consequence of single precision having B_S bits is that every single precision operation has a *relative error bound* of 2^{-B_S}. For example,

$$|(a +_{B_S} b) - (a + b)| \leq 2^{-B_S}|a + b|,$$

where $a +_{B_S} b$ refers to the sum of a and b computed using single precision arithmetic. The analogous bound holds for the other three operations. This broad assumption holds for all floating point representations. Particular floating point representations may have more specific properties [6], but we do not exploit them here.

For a given set of points in two dimensions, the *diameter* Δ of the set is defined as usual: the maximum distance between a pair of points in the set. For geometric calculations on a set of points, *absolute error* is measured as a multiple of the *rounding unit* $\mu = 2^{-B_S}\Delta$. This multiple is independent of precision and scale.

3.2 Calculating Angles

An essential calculation in constructing convex hulls is to determine if two line segments **AB** and **BC** form a convex or concave angle. The robust convex hull algorithm requires the slightly more general operation of computing the angle formed by separate line segments **AB** and **CD**. Without loss of generality, assume that the two segments have positive slope and "point to the right": $\mathbf{A}_x < \mathbf{B}_x$, $\mathbf{A}_y < \mathbf{B}_y$, $\mathbf{C}_x < \mathbf{D}_x$, and $\mathbf{C}_y < \mathbf{D}_y$. Define $\theta(\mathbf{AB})$ to be the angle that line **AB** makes with the x-axis. Define $\theta(\mathbf{AB},\mathbf{CD}) = \theta_{\mathbf{CD}} - \theta_{\mathbf{AB}}$ to be the angle that line **CD** makes with line **AB**. The following exact arithmetic function determines if $\theta(\mathbf{AB},\mathbf{CD})$ is greater than or equal to zero.

PositiveAngleSign ($\mathbf{AB}$, $\mathbf{CD}$)
  if $[(\mathbf{B}_x - \mathbf{A}_x)(\mathbf{D}_y - \mathbf{C}_y) - (\mathbf{B}_y - \mathbf{A}_y)(\mathbf{D}_x - \mathbf{C}_x) \geq 0]$
  then return *true*
  else return *false*

Actually calculating the angle $\theta(\mathbf{AB},\mathbf{CD})$ requires taking a square root and arcsine. However, we can avoid these by computing the *relative change in slope*,

$$\begin{aligned}\tau(\mathbf{AB},\mathbf{CD}) &= \frac{\tan\theta(\mathbf{CD}) - \tan\theta(\mathbf{AB})}{|\tan\theta(\mathbf{CD})| + |\tan\theta(\mathbf{AB})|}, \\ &= \frac{\dfrac{\mathbf{D}_y - \mathbf{C}_y}{\mathbf{D}_x - \mathbf{C}_x} - \dfrac{\mathbf{B}_y - \mathbf{A}_y}{\mathbf{B}_x - \mathbf{A}_x}}{\left|\dfrac{\mathbf{D}_y - \mathbf{C}_y}{\mathbf{D}_x - \mathbf{C}_x}\right| + \left|\dfrac{\mathbf{B}_y - \mathbf{A}_y}{\mathbf{B}_x - \mathbf{A}_x}\right|}, \\ &= \frac{(\mathbf{B}_x - \mathbf{A}_x)(\mathbf{D}_y - \mathbf{C}_y) - (\mathbf{B}_y - \mathbf{A}_y)(\mathbf{D}_x - \mathbf{C}_x)}{|(\mathbf{B}_x - \mathbf{A}_x)(\mathbf{D}_y - \mathbf{C}_y)| + |(\mathbf{B}_y - \mathbf{A}_y)(\mathbf{D}_x - \mathbf{C}_x)|}.\end{aligned}$$

Under the assumption that all segments have positive slope, we can leave out the absolute values. We can compute the relative change in slope using the third formula, or if we wish to avoid any divisions, the following function determines if the relative change in slope is greater than a particular value κ.

RelativeSlopeCompare ($\mathbf{AB}$, $\mathbf{CD}$, κ)
  if $[(\mathbf{B}_x - \mathbf{A}_x)(\mathbf{D}_y - \mathbf{C}_y) - (\mathbf{B}_y - \mathbf{A}_y)(\mathbf{D}_x - \mathbf{C}_x)$
    $> \kappa((\mathbf{B}_x - \mathbf{A}_x)(\mathbf{D}_y - \mathbf{C}_y) + (\mathbf{B}_y - \mathbf{A}_y)(\mathbf{D}_x - \mathbf{C}_x))]$
  then return *true*
  else return *false*

Exact convex hull algorithms use PositiveAngleSign, and the robust algorithms presented here use RelativeSlopeCompare. It might appear that RelativeSlopeCompare is more expensive to compute, but if one exploits common subexpressions, it only requires one extra addition and one extra multiplication to implement. Furthermore, as the following lemmas show, the actual cost of exact calculations may be much higher than their apparent cost.

Lemma 5 PositiveAngleSign *can be computed exactly using double precision additions (or subtractions) and multiplications. The following table gives bounds on the number of operations required for various types of input. In the case of fixed point inputs, four of the additions can be performed using integer operations.*

Input Representation	Additions	Multiplications
Fixed Point	5	2
Single Precision Floating Point	7	8
Double Precision Floating Point	71	72

Lemma 6 RelativeSlopeCompare *can be computed approximately using* 6 *single precision floating point additions and* 3 *single precision floating point multiplications. The absolute accuracy is* $(3 + 6\kappa) \cdot 2^{-B_S}$. *In particular,*

- *if* RelativeSlopeCompare$(\mathbf{AB}, \mathbf{CD}, \kappa) =$ *true then* $\tau(\mathbf{AB}, \mathbf{CD}) \geq \kappa - (3 + 6\kappa) \cdot 2^{-B_S}$;
- *if* RelativeSlopeCompare$(\mathbf{AB}, \mathbf{CD}, \kappa) =$ *false then* $\tau(\mathbf{AB}, \mathbf{CD}) < \kappa + (3 + 6\kappa) \cdot 2^{-B_S}$.

The error bound using double precision arithmetic is $(3 + 6\kappa) \cdot 2^{-B_D}$.

In practice, we only invoke RelativeSlopeCompare for small values of κ (about $32 \cdot 2^{-B_S}$). In this case, we can neglect the error term $6 \cdot 2^{-B_S}\kappa$. For this reason, in the rest of the paper, the error bound will be simply $3 \cdot 2^{-B_S}$.

As it will turn out, the robust convex hull algorithm makes two calls to RelativeSlopeCompare for every call to PositiveAngleSign made by an exact algorithm. However, by exploiting common subexpressions, the robust algorithm can be made to use only four single precision multiplications for every two double precision multiplications made by the exact algorithm. This is assuming that the exact algorithm is acting on integer inputs. In this case it is probably best to use the exact algorithm. However, if the inputs are single or double precision, the approximate algorithm has a clear advantage.

Proof. (Lemma 5) In the case of fixed point inputs, four subtractions can be carried out in fixed point (integer) arithmetic without rounding. The remaining two multiplications and one subtraction can be carried out in double precision, and no roundoff error will occur—assuming that double precision can hold the square of an integer.

In the case of single precision floating point inputs, the expression can be expanded in eight terms consisting of the product of two single precision floating point numbers. Each of these products can be performed in double precision, and there exists a way of summing the eight resulting double precision numbers in double precision so that the result is guaranteed to have the same sign as the exact answer.[3]

Finally, in the case of double precision floating point inputs, each input can be expressed as the sum of three single precision floating point numbers. Expanding as before results in 72 double precision terms to be summed. □

Proof. (Lemma 6) A relative error of 2^{-B_S} in the calculation of $\mathbf{B}_x - \mathbf{A}_x$ results in an absolute error of $2^{-B_S}(\mathbf{B}_x - \mathbf{A}_x)(\mathbf{D}_y - \mathbf{C}_y)$ in the calculation of the expression,

$$(\mathbf{B}_x - \mathbf{A}_x)(\mathbf{D}_y - \mathbf{C}_y) - (\mathbf{B}_y - \mathbf{A}_y)(\mathbf{D}_x - \mathbf{C}_x).$$

Similar bounds hold for the two multiplication and three of the other subtractions. The sum of these errors is bounded by,

$$3 \cdot 2^{-B_S}((\mathbf{B}_x - \mathbf{A}_x)(\mathbf{D}_y - \mathbf{C}_y) + (\mathbf{B}_y - \mathbf{A}_y)(\mathbf{D}_x - \mathbf{C}_x)).$$

The final subtraction evaluated results in an absolute error bounded by,

$$2^{-B_S}|(\mathbf{B}_x - \mathbf{A}_x)(\mathbf{D}_y - \mathbf{C}_y) - (\mathbf{B}_y - \mathbf{A}_y)(\mathbf{D}_x - \mathbf{C}_x)|.$$

This corresponds to a relative error of up to 2^{-B_S} in the calculation of the angle. Since this error is relative, we can treat it as if it occurs to the right hand side of the inequality. Using similarly reasoning, it can be shown that the absolute error in computing the expression,

$$\kappa((\mathbf{B}_x - \mathbf{A}_x)(\mathbf{D}_y - \mathbf{C}_y) + (\mathbf{B}_y - \mathbf{A}_y)(\mathbf{D}_x - \mathbf{C}_x)),$$

is bounded by,

$$5 \cdot 2^{-B_S}\kappa((\mathbf{B}_x - \mathbf{A}_x)(\mathbf{D}_y - \mathbf{C}_y) + (\mathbf{B}_y - \mathbf{A}_y)(\mathbf{D}_x - \mathbf{C}_x)),$$

(one extra multiplication). The transfer of error from the right to the left changes the 5 to a 6. Together these error bounds imply the lemma. □

[3] Numbers with aligned least significant bits can be added without roundoff occurring. Otherwise, add numbers from smallest to largest.

3.3 A Note on Computing Angles

In the rest of the paper, we will treat $\tau(\mathbf{AB}, \mathbf{CD})$ as if it were $\theta(\mathbf{AB}, \mathbf{CD})$, the actual angle between segments $\mathbf{AB}$ and $\mathbf{CD}$. We can do this for two reasons. First, $\tau(\mathbf{AB}, \mathbf{CD})$ is always an overestimation of $\theta(\mathbf{AB}, \mathbf{CD})$. In particular,

$$\tau(\mathbf{AB}, \mathbf{CD}) \geq \sin\theta(\mathbf{AB}, \mathbf{CD}) \geq \theta(\mathbf{AB}, \mathbf{CD}),$$

for all positively sloped line segments that "point to the right." For $\mathbf{AB}$ and $\mathbf{CD}$ nearly horizontal or vertical, $\tau(\mathbf{AB}, \mathbf{CD})$ is much larger than $\theta(\mathbf{AB}, \mathbf{CD})$. This means that an absolute error of $3 \cdot 2^{-B_S}$ in the measurement of τ corresponds to a much smaller error in the measurement of θ.

The second reason that we can replace θ by τ is that for fixed $\mathbf{AB}$, $\tau(\mathbf{AB}, \mathbf{CD})$ is a monotonic function of $\theta(\mathbf{AB}, \mathbf{CD})$. This implies the following lemma which we use in Section 4.2.

Lemma 7 *If* $\tau(\mathbf{AB}, \mathbf{BC})$, $\tau(\mathbf{AB}, \mathbf{CD}) < \kappa$, *then* $\tau(\mathbf{AB}, \mathbf{BD}) < \kappa$.

Proof. This simply follows from the monotonicity of τ and the fact that,

$$\min(\theta(\mathbf{BC}), \theta(\mathbf{CD})) \leq \theta(\mathbf{BD}) \leq \max(\theta(\mathbf{BC}), \theta(\mathbf{CD})).$$

□

3.4 Geometric Distance

The signed distance from a point $\mathbf{C}$ to a line $\mathbf{AB}$ can be computed by the formula,

$$\delta(\mathbf{C}, \mathbf{AB}) = \frac{\mathbf{AB} \times \mathbf{AC}}{|\mathbf{AB}|} = \frac{(\mathbf{B}_x - \mathbf{A}_x)(\mathbf{C}_y - \mathbf{A}_y) + (\mathbf{B}_y - \mathbf{A}_y)(\mathbf{C}_x - \mathbf{A}_x)}{\sqrt{(\mathbf{B}_x - \mathbf{A}_x)^2 + (\mathbf{B}_y - \mathbf{A}_y)^2}}.$$

The distance function has positive sign when $\mathbf{C}$ lies to the left of $\mathbf{AB}$ as viewed from $\mathbf{A}$ looking at $\mathbf{B}$.

The following function determines if $\delta(\mathbf{C}, \mathbf{AB})$ is greater than a positive value κ. The version which handles negative κ is straightforward, and it involves the same number of arithmetic operations.

DistanceCompare ($\mathbf{AB}$, $\mathbf{C}$, κ)
  if $[(\mathbf{B}_x - \mathbf{A}_x)(\mathbf{C}_y - \mathbf{A}_y) - (\mathbf{B}_y - \mathbf{A}_y)(\mathbf{C}_x - \mathbf{A}_x) \geq 0$
    and
    $((\mathbf{B}_x - \mathbf{A}_x)(\mathbf{C}_y - \mathbf{A}_y) - (\mathbf{B}_y - \mathbf{A}_y)(\mathbf{C}_x - \mathbf{A}_x))^2$
    $\geq \kappa^2((\mathbf{B}_x - \mathbf{A}_x)^2 + (\mathbf{B}_y - \mathbf{A}_y)^2)]$
  then return *true*
  else return *false*

DistanceCompare avoids division and square roots. Even for fixed point inputs, computing this function exactly costs more than using approximate arithmetic. For double precision inputs, the approximate version is considerably cheaper, as it was for comparing angles.

Lemma 8 DistanceCompare *can be computed approximately using single precision floating point operations. The absolute accuracy is* $3 \cdot 2^{-B_S}|\mathbf{AB}| + 6.5\kappa \cdot 2^{-B_S}$. *The error bound using double precision arithmetic is* $3 \cdot 2^{-B_D}|\mathbf{AB}| + 6.5\kappa \cdot 2^{-B_D}$.

Proof. (partial) The proof of this lemma is similar to that of Lemma 6. As with RelativeSlopeCompare, we will invoke this function for only very small values of κ. Hence, we will use the error bound $3 \cdot 2^{-B_D}|\mathbf{AB}|$ which is in turn bounded by 3μ since $|\mathbf{AB}| \leq \Delta$, the diameter of the set of input points. □

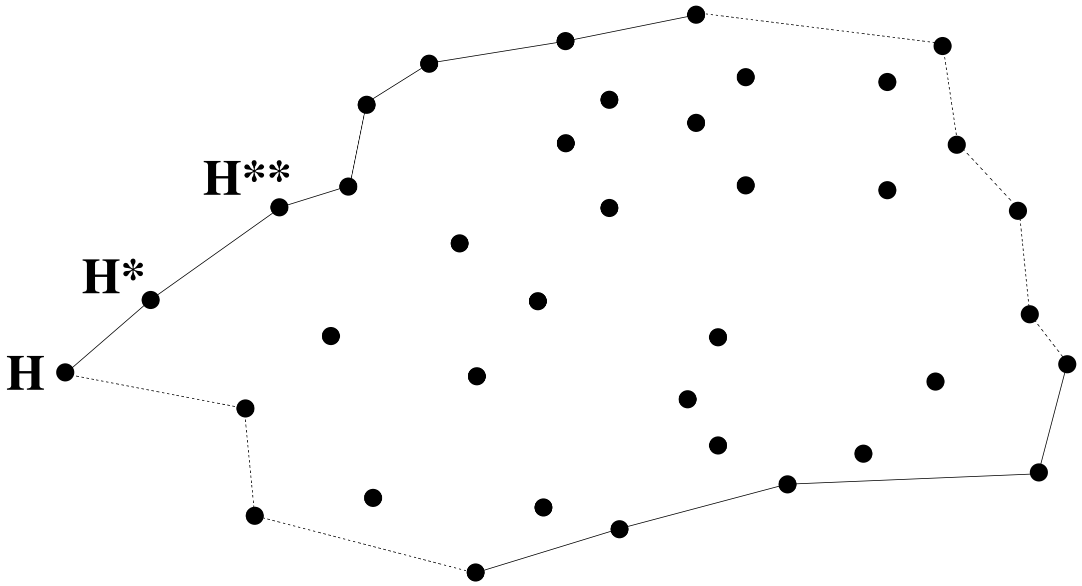

Figure 4: Monotonic Sequence Containing Section of Convex Hull

4 Robust Convex Hull Algorithm

In order to implement the algorithm of Section 2 using single (or double) precision floating point arithmetic, we apply Lemma 8. Replacing 2ϵ by $2\epsilon + 3\mu$ wherever it appears assures that the rounded version will still generate an ϵ-strongly convex polygon. However, instead of a 12ϵ-hull, the output is a $(12\epsilon + 18\mu)$-hull.[4] Unfortunately, rounded arithmetic does not allow us to construct the convex hull we need to get started. The rest of this section will show how to modify the standard Graham algorithm to generate a convex 36μ-hull of a set S of n points in $O(n \log n)$ time. Using this instead of the convex hull results in a ϵ-strongly convex $(12\epsilon + 54\mu)$-hull.

The first step is to extract four monotonic sequences of points from S which cover the vertices of the convex hull. For each sequence, the x and y coordinates are monotonic functions of index. To generate a sequence, we sort S by increasing x-coordinate and then examine the vertices in that order. Every time we encounter a vertex whose y-coordinate is less than or equal to the greatest y-coordinate we have seen so far, we throw it out. If several vertices have the same x-coordinate, we throw out all but the one with the largest y-coordinate. In Figure 4, vertex **H** is the head of the sequence and **H*** and **H**** are subsequent elements, where "*****" is the successor operator. To generate the other three sequences we run the same algorithm on the original set S, but with the sense of the comparison in the x-direction or the y-direction or both switched. Figure 4 illustrates the four sequences. The increasing sequences are solid and the decreasing sequences are dotted.

The next three sections will describe how to extract the vertices of a convex polygonal curve from each monotonic sequence. Putting the four parts together generates a convex 36μ-hull. Generating the monotonic sequences costs $O(n \log n)$ (to sort the x-coordinates). Extracting the convex curves is $O(n)$.

4.1 Extracting A Spine

It can be shown that if we naively remove reflex vertices from a monotonic sequence, we may introduce error $n\mu$ because non-reflex vertices may *appear* to be reflex as a result of roundoff error. Instead, we first create a sequence of vertices and edges we call a *spine* which is a subsequence of a monotonic sequence. Removing reflex vertices from a spine introduces only $O(\mu)$ error. This section gives the algorithm for creating a spine,

[4] Each time we think we are removing $2\epsilon + 3\mu$, we may actually be removing an extra 3μ, hence an addition error of $6 \cdot 3\mu = 18\mu$.

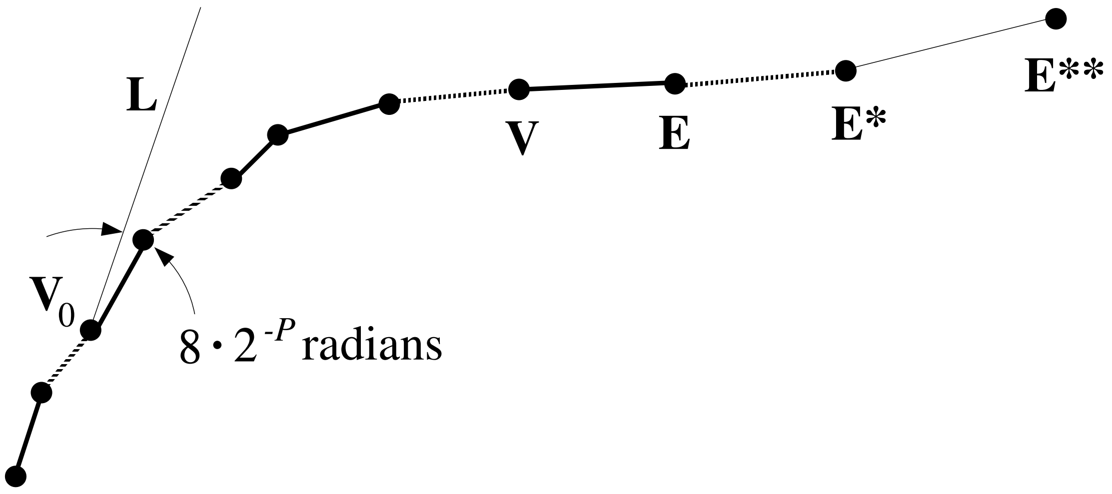

Figure 5: Polygonal Curve of Vertebrae and Extenders ($P = B_S$ for Single Precision Floating Point)

Section 4.2 proves that it is a 30μ-hull. Section 4.3 shows that removing the reflex vertices from the spine adds only 6μ more error.

A *spine* is a sequence of edges of two types: vertebrae and extenders. These edges satisfy three conditions:

1. the first edge is a vertebra, and each vertebra is followed by at most one extender;
2. each vertebra is rotated at least $18 \cdot 2^{-B_S}$ radians clockwise with respect to the preceding vertebra;
3. each extender is rotated between $-6 \cdot 2^{-B_S}$ radians and $24 \cdot 2^{-B_S}$ radians clockwise with respect to the preceding vertebra, and it is rotated at most $6 \cdot 2^{-B_S}$ radians clockwise with respect to the succeeding vertebra.

Figure 5 depicts a spine. The dark edges are vertebrae, and the dotted edges are extenders. The vertices **V** and **E** and the edge **E*E**** are discussed in the following paragraph. The vertex $\mathbf{V}_0$ and the line **L** are part of the statement of Lemma 10 below. Intuitively, the angles of the vertebrae form an increasing sequence (using the definition of the angle of an edge given in Section 3.2). An extender may be reflex with respect to one of its neighboring vertebrae, but only a little bit.

Like the Graham algorithm for growing a convex hull, we "grow" a spine from left to right by removing vertices from a monotonic sequence. At any point in the computation, we have a partially constructed spine followed by unprocessed edges. Let **VE** be the rightmost vertebra, and let **EE*** be the extender that follows it. If no extender follows it, then $\mathbf{E} = \mathbf{E^*}$ (zero-length extender). Let **E*E**** be the next unprocessed edge. RelativeSlopeCompare($\mathbf{E^*E^{**}}, \mathbf{VE}, \kappa$) is *true* if the clockwise rotation of **E*E**** with respect to **VE** is greater than κ, as calculated using rounded arithmetic. Recall that Section 3.2 describes how to compute RelativeSlopeCompare. Section 3.3 discusses the fact that RelativeSlopeCompare does not actually compute angles but that for the purposes of discussion and proof of error bound, one can assume it does.

A. If RelativeSlopeCompare($\mathbf{E^*E^{**}}, \mathbf{VE}, 21 \cdot 2^{-B_S}$) = *true*, make **E*E**** a vertebra.

B. If not, merge **E*E**** with the extender **EE***. That is, throw away **E*** if $\mathbf{E^*} \neq \mathbf{E}$. Vertex **E**** changes its name to **E***.

C. If RelativeSlopeCompare($\mathbf{VE}, \mathbf{EE^*}, 3 \cdot 2^{-B_S}$) = *true* (newly created extender forms reflex angle with vertebra), throw away **E** also. Newly created edge **VE*** is added to list of unprocessed edges.

4.2 Proof of Algorithm

We prove the correctness of the algorithm in two stages. First we prove that it satisfies conditions 1, 2, and 3. Then we show that the generated spine is a 30μ-hull.

Lemma 9 *Each time rules A, B, and C are applied, the partially constructed spine satisfies conditions 1, 2, and 3.*

Proof. Most of the proof follows immediately from the fact that we can measure angles with accuracy plus or minus $3 \cdot 2^{-B_S}$ (Lemma 6). The most difficult part is showing that the new extender created by growth rule B satisfies condition 3.

Condition 1 is easy to check. Condition 2 is a consequence of rule A. An edge becomes a new vertebra only if its calculated rotation with respect to the previous is at least $21 \cdot 2^{-B_S}$ greater. This implies that the actual angle is at least $18 \cdot 2^{-B_S}$ greater. The $24 \cdot 2^{-B_S}$ bound in in Condition 3 is a consequence of rule B. An edge is merged with the current extender if its rotation appears less than $21 \cdot 2^{-B_S}$. This implies that the actual angle is no more than $24 \cdot 2^{-B_S}$. If the extender satisfies this bound before we merge the new edge, it will satisfy it afterwards.

Rule C ensures that $-6 \cdot 2^{-B_S}$ bound holds in condition 3. For if it did not, then the calculated angle of the extender with respect to the vertebra would be at least $-3 \cdot 2^{-B_S}$, and rule C would delete it. Finally, Rule A ensures that the $6 \cdot 2^{-B_S}$ bound holds in condition 3. The calculated angle of the extender with respect to the preceding vertebra is at most $21 \cdot 2^{-B_S}$, and the calculated angle of the succeeding vertebra with respect to the preceding vertebra is at least $21 \cdot 2^{-B_S}$. Both these computations can be off by $3 \cdot 2^{-B_S}$, and thus the total error is bounded by $6 \cdot 2^{-B_S}$. □

Lemma 10 *Let* $\mathbf{V}_0$ *be a vertex which is the left endpoint of a vertebra, and let* $\mathbf{L}$ *be the line through* $\mathbf{V}_0$ *rotated* $6 \cdot 2^{-B_S}$ *radians counter-clockwise with respect to the vertebra. Every point of the spine to the right of* $\mathbf{V}_0$ *lies to the right of* $\mathbf{L}$ *(Figure 5).*

Proof. Condition 3 implies that the succeeding extender may be parallel to $\mathbf{L}$ at worst. Each subsequent vertebra is rotated at least $18 \cdot 2^{-B_S}$ clockwise with respect to the vertebra of $\mathbf{V}_0$, and each subsequent extender is rotated at least $12 \cdot 2^{-B_S}$. □

Lemma 11 *Suppose we run the spine algorithm and at time* s, $\mathbf{V}_0$ *is the left endpoint of a vertebra* $\mathbf{V}_0\mathbf{E}s$, *and at some later time* t, *it is the left endpoint of a different vertebra* $\mathbf{V}_0\mathbf{E}t$. *Then* $\mathbf{V}_0\mathbf{E}t$ *will be rotated counter-clockwise with respect to* $\mathbf{V}_0\mathbf{E}s$.

Proof. It suffices to assume that when $\mathbf{V}_0\mathbf{E}s$ was deleted, $\mathbf{V}_0\mathbf{E}t$ was the next vertebra created. However, this means that $\mathbf{V}_0\mathbf{E}t$ was created from an unprocessed edge generated by rule C. This edge is rotated counter-clockwise from $\mathbf{V}_0\mathbf{E}s$ because $\mathbf{V}_0\mathbf{E}s\mathbf{E}t$ is a reflex angle. □

Lemma 12 *Let* $\mathbf{V}_0\mathbf{V}_0*$ *be any vertebra in the final result. For all points* $\mathbf{P}$ *in* S *to the right of* $\mathbf{V}_0$, $\mathbf{V}_0\mathbf{P}$ *is rotated at most* $6 \cdot 2^{-B_S}$ *counter-clockwise with respect to* $\mathbf{V}_0\mathbf{V}_0*$. *In other words,* $\mathbf{P}$ *lies to the right of* $\mathbf{L}$ *in Figure 5.*

Proof. Assume there is some $\mathbf{P}$ that violates this lemma. When we first encounter $\mathbf{P}$ at time t, the point $\mathbf{V}$ will be the left endpoint of some vertebra, perhaps not the same as the one in the final result. However, if it is not the same, it will be rotated clockwise with respect to the final vertebra, by Lemma 11. Furthermore, the rightmost vertebra $\mathbf{VE}$ at time t will be rotated even more clockwise. On the other hand, $\mathbf{V}$ lies to the right of $\mathbf{L}$ (Lemma 10) and to the right of $\mathbf{V}_0$. Therefore $\mathbf{VP}$ will be rotated counter-clockwise with respect to $\mathbf{V}_0\mathbf{P}$. As a consequence, angle $\mathbf{EVP}$ will be greater than $6 \cdot 2^{-B_S}$ radians, and growth rule C will assure that $\mathbf{E}$ will be deleted. Ultimately, all the vertices between $\mathbf{V}_0$ and $\mathbf{P}$ will be deleted in the same fashion. This is a contradiction because we would then delete $\mathbf{V}_0*$ from the final result. □

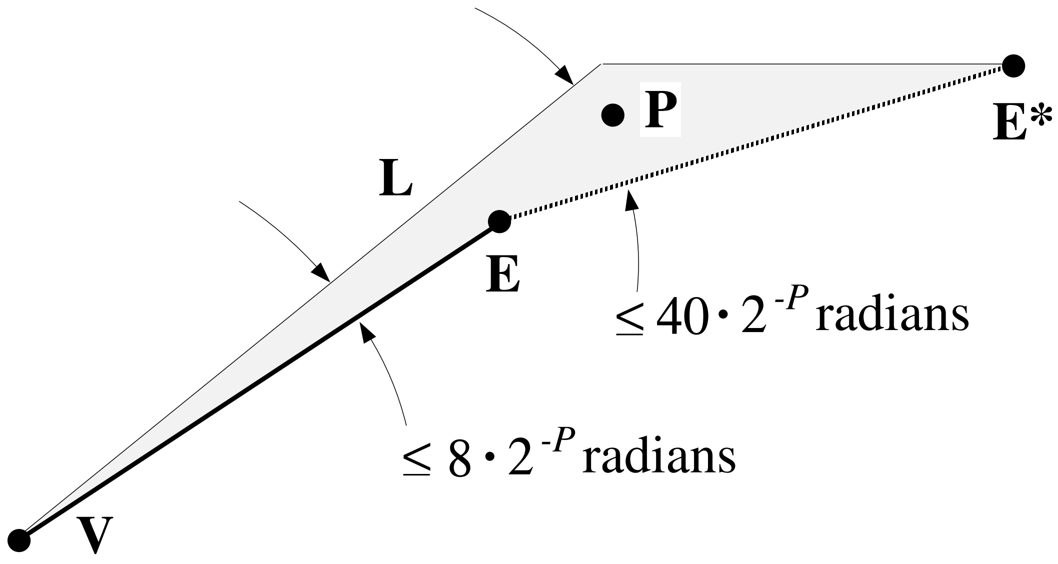

Figure 6: Region Trimmed by Extract ($P = B_S$ for Single Precision Floating Point)

Lemma 13 *No point lies farther than* 30μ *outside the spine.*

Proof. It suffices to prove this bound for all points **P** in the monotonic sequence. Let **VE** and **EE*** be the vertebra-extender pair such that **P** appears somewhere in the original monotonic sequence between **V** and **E***. Lemma 12 implies **P** lies in a region depicted in Figure 6. Line **L** forms an angle of $6 \cdot 2^{-B_S}$ radians with respect to **VE**, and **EE*** forms an angle of at most $24 \cdot 2^{-B_S}$ radians in the other direction (condition 3). Therefore angle **E*VP** is bounded by $30 \cdot 2^{-B_S}$ radians. The distance from **P** to the spine is bounded by

$$\begin{aligned} \delta(\mathbf{P}, \mathbf{VE^*}) &< 30 \cdot 2^{-B_S} |\mathbf{VE^*}| \\ &< 30 \cdot 2^{-B_S} \cdot \Delta = 30\mu. \end{aligned}$$

This completes the proof that the spine is a 30μ-hull. □

4.3 Removing Reflex Angles

The spine may still have some reflex angles. This section describes a simple post-processing stage which removes these angles without introducing much more error.

If an extender *might* form a reflex angle with one of the neighboring vertebra (if the calculated angle is less than $3 \cdot 2^{-B_S}$), we remove the appropriate endpoint. Because of the large angle (at least $18 \cdot 2^{-B_S}$) between successive vertebra, we never remove both endpoints of an extender. Hence, removing potentially reflex angles trims off another region of thickness at most 6μ and alters the orientation of the vertebra by at most $6 \cdot 2^{-B_S}$ radians. Since vertebra differ in orientation by at least $18 \cdot 2^{-B_S}$ radians before the final trimming, they differ by at least $6 \cdot 2^{-B_S}$ radians afterward, and thus all angles are measurably convex. Total error is 36μ. Removing the reflex vertices only requires linear time. Therefore we shown have the following.

Theorem 14 *Using floating point arithmetic with P-bit mantissa and rounding unit $\mu = 2^{-B_S}\Delta$, one can construct a convex 36μ-hull of a set of n points in $\mathrm{O}(n \log n)$ time. To obtain B bits of accuracy, only $(B+6)$-bit arithmetic is required.*

If we cared to, we could replace $21 \cdot 2^{-B_S}$ in rule A by a larger angle $21 \cdot 2^{-B_S} + 2\epsilon/\Delta$. Then, in the post-processing stage, we would remove any angles whose calculated value is less than $\epsilon/\Delta + 3 \cdot 2^{-B_S}$. This would ensure that all the angles[5] are measurably greater than ϵ/Δ at the cost of increasing the total error to $4\epsilon + 36\mu$.

5 Conclusion

Our approximate hull constructions have optimal complexity of $O(n \log n)$ and optimal error bounds of a constant times ϵ plus a constant times μ. As discussed in the introduction, the $\Omega(n \log n)$ cost of sorting the vertices by x-coordinate is fixed and independent of the algorithm used to construct the hull thereafter. For the actual hull construction, our technique can achieve double precision accuracy on double precision inputs with less cost than the exact algorithm. Constructing the ϵ-strongly convex $(12\epsilon + 54\mu)$-hull requires only a linear number of additional arithmetic operations. For any amount of input precision, our robust algorithm can perform this construction with less cost than the exact algorithm.

An ϵ-strongly convex polygon has the property that all angles are less than $\pi - \epsilon/\Delta$ radians, where Δ is the diameter of the polygon. However, if that property is the only one we desire, than for any positive angle α, we can can construct a polygon whose angles are all less than $\pi - \alpha$ which is a $(4\alpha\Delta + 36\mu)$-hull. Thus the constructed polygons have all the desirable properties outlined in the introduction.

In the introduction, we made the claim that the ability to construct strongly convex objects should make it easier to construct the Minkowski sum of two polygonal regions, or their intersection, using approximate arithmetic. For both of these operations, there exist fast algorithms for convex polygons which do not work on general polygons. In the case of the Minkowski sum, the fast algorithm merges the two sets of edges so that the resulting set is ordered by angle. This algorithm would not work if the edges of one polygon attain the same angle multiple times, as may happen in a general polygon. The robust algorithm will not work if the edges *appear* (as a result of roundoff error) to attain the same angle multiple times. However, if the error in measuring angles is α, the input polygons will not appear to have parallel edges if all of its angles are less than $\pi - \alpha$. One fast way to intersect convex polygons is to look at them in the dual space in which each edge becomes a point and each vertex becomes a line. Again we merge the lists of points by angle and then compute the convex hull. In this case we might need the results of this paper twice: first to assure proper merging by angle, second to compute the convex hull in the dual space. Robust Minkowski sums and polygon intersections are a subject for another paper, but it is clear that the ability to robustly construct strongly convex polygons is a prerequisite.

Another claim made in the introduction is that strongly robust polygons make it possible to apply Euclidean transformations to a polygon. For example, suppose we wished to construct the intersection of two polygons: the first one as it is, and the second after a rotation and translation. For a bounded number of translations, there will be some ϵ such that an ϵ-strongly convex polygon will remain convex after that number of translations. If we wish to use a robust intersection algorithm, we can use a slightly larger value of ϵ. If we have to apply more transformations than we anticipated, we will still have to recompute the convex hull of the transformed polygon, but the use of strongly convex polygons eliminates the need to recompute the convex hull *each* time we perform a transformation.

Our algorithm for constructing ϵ-strongly convex polygons is practical. An empirical study shows that it generally introduces only ϵ error, which is the lower bound, instead of the upper bound of 12ϵ. Figures 7 and 8 show that output of the algorithm for $\epsilon = 0.5$ and $\epsilon = 1$, respectively, on a large convex polygon. One can see that the trimmed polygon lies only about ϵ distant from the original convex hull. Currently we are implementing the robust version. We expect that the typical error will be much smaller than 54μ.

As we have seen, our techniques of spine construction and trimming are successful in two dimensions. The logical next step is to look for generalizations to higher dimensions. Similar results in three or more dimension would have a great practical impact on the cost of computing convex hulls and Voronoi diagrams of non-planar sets of points.

[5] These are the *external* angles. An external angle of α corresponds to an internal angle of $\pi - \alpha$.

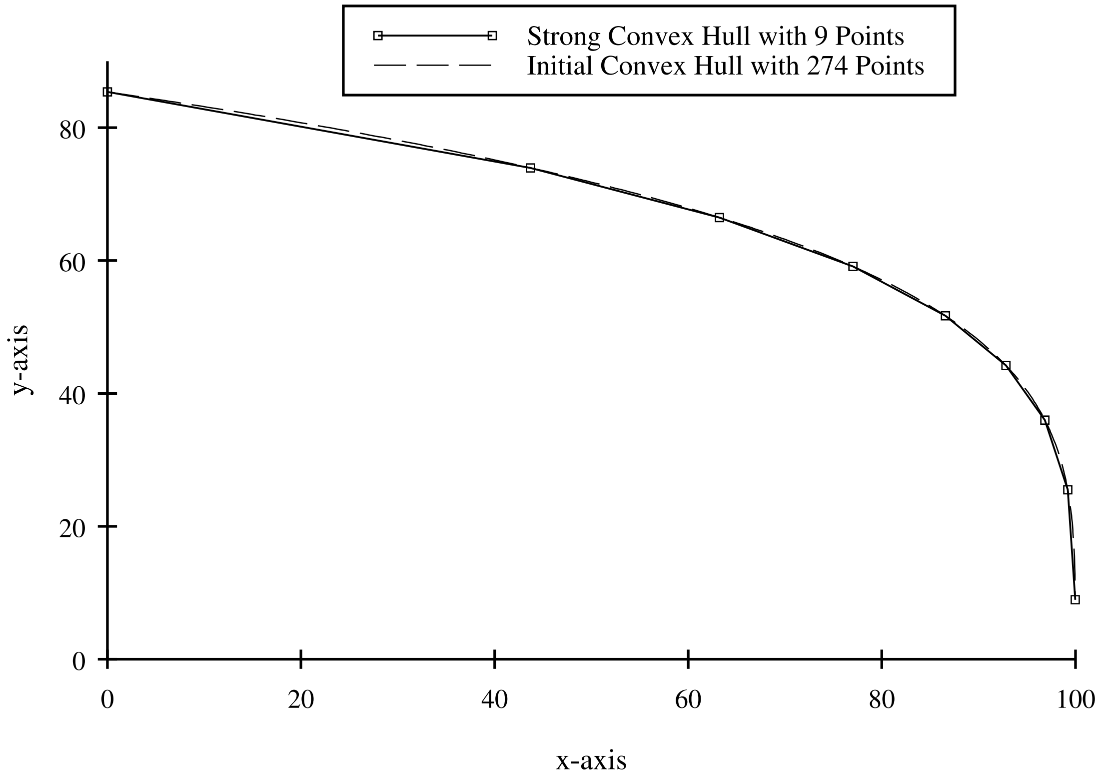

Figure 7: ϵ-Strongly Convex Polygon: $\epsilon = 0.5$

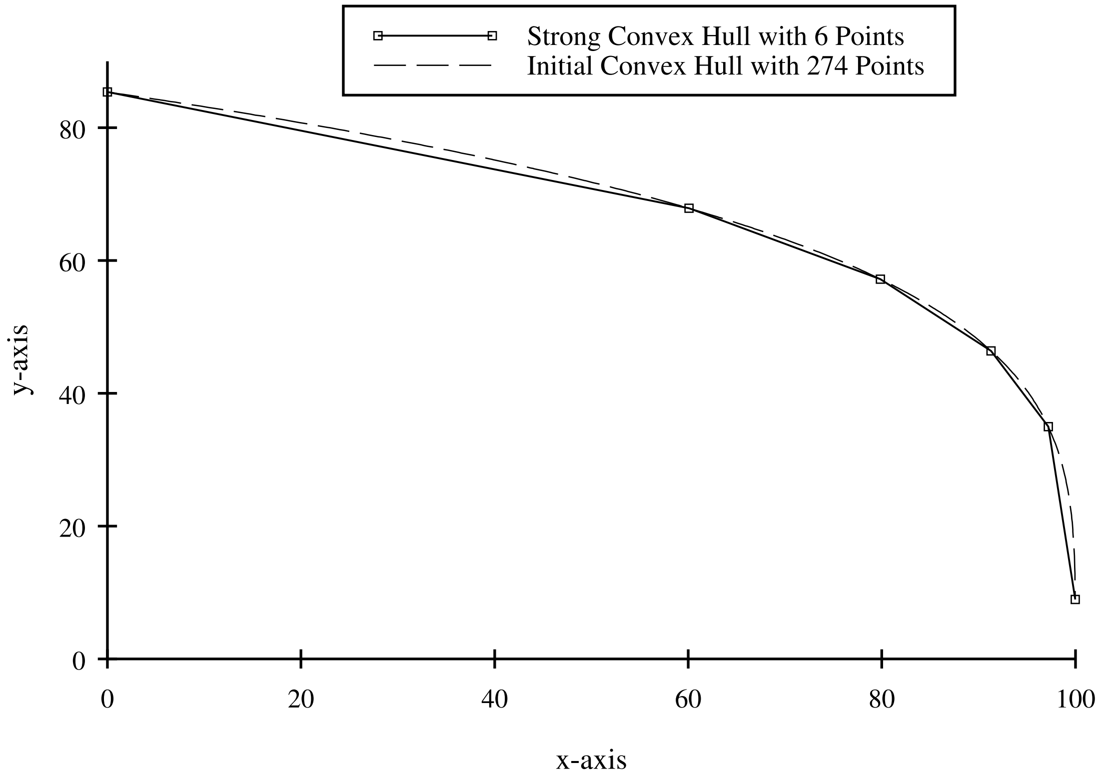

Figure 8: ϵ-Strongly Convex Polygon: $\epsilon = 1$

References

[1] Steven Fortune. Stable Maintenance of Point-Set Triangulation in Two Dimensions. In *30th Annual Symposium on the Foundations of Computer Science*, IEEE, October 1989.

[2] Ronald Graham. An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set. *Information Processing Letters*, 1:132–133, 1972.

[3] Leonidas Guibas, David Salesin, and Jorge Stolfi. Epsilon Geometry: Building Robust Algorithms from Imprecise Computations. In *Proceedings of the Symposium on Computational Geometry*, pages 208–217, ACM, July 1989.

[4] Leonidas Guibas, David Salesin, and Jorge Stolfi. Constructing Strongly Convex Approximate Hulls with Inaccurate Primitives. In *Proceedings of the SIGAL International Symposium on Algorithms*, Tokyo, Japan, August 16-18, 1990.

[5] Victor Milenkovic. Calculating approximate curve arrangements using rounded arithmetic. In *Proceedings of the Symposium on Computational Geometry*, ACM, pages 197–207, June 1989.

[6] Victor Milenkovic. Double Precision Geometry: A General Technique for Calculating Line and Segment Intersections Using Rounded Arithmetic. In *30th Annual Symposium on the Foundations of Computer Science*, IEEE, October 1989.

[7] Victor J. Milenkovic. *Verifiable Implementations of Geometric Algorithms using Finite Precision Arithmetic*. Ph.D. Thesis, Carnegie-Mellon, 1988. Technical Report CMU-CS-88-168, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, July 1988.

[8] Kokichi Sugihara and Masao Iri. Construction of the Voronoi Diagram for One Million Generators in Single-Precision Arithmetic Presented at First Canadian Conference on Computational Geometry, August 21-25, 1989, Montreal, Canada. Available as preprint, Department of Mathematical Engineering and Information Physics, Faculty of Engineering, University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan.