Reliable and efficient mesh processing

Marco Attene

CNR – IMATI Genova

The National Research Council (CNR)

The CNR is the *largest public research institution in Italy*.

Mission: to perform, transfer and enhance research activities, to promote innovation and competitiveness of the national industrial system, to promote the internationalization of the national research system, to provide technologies and solutions to emerging public and private needs.

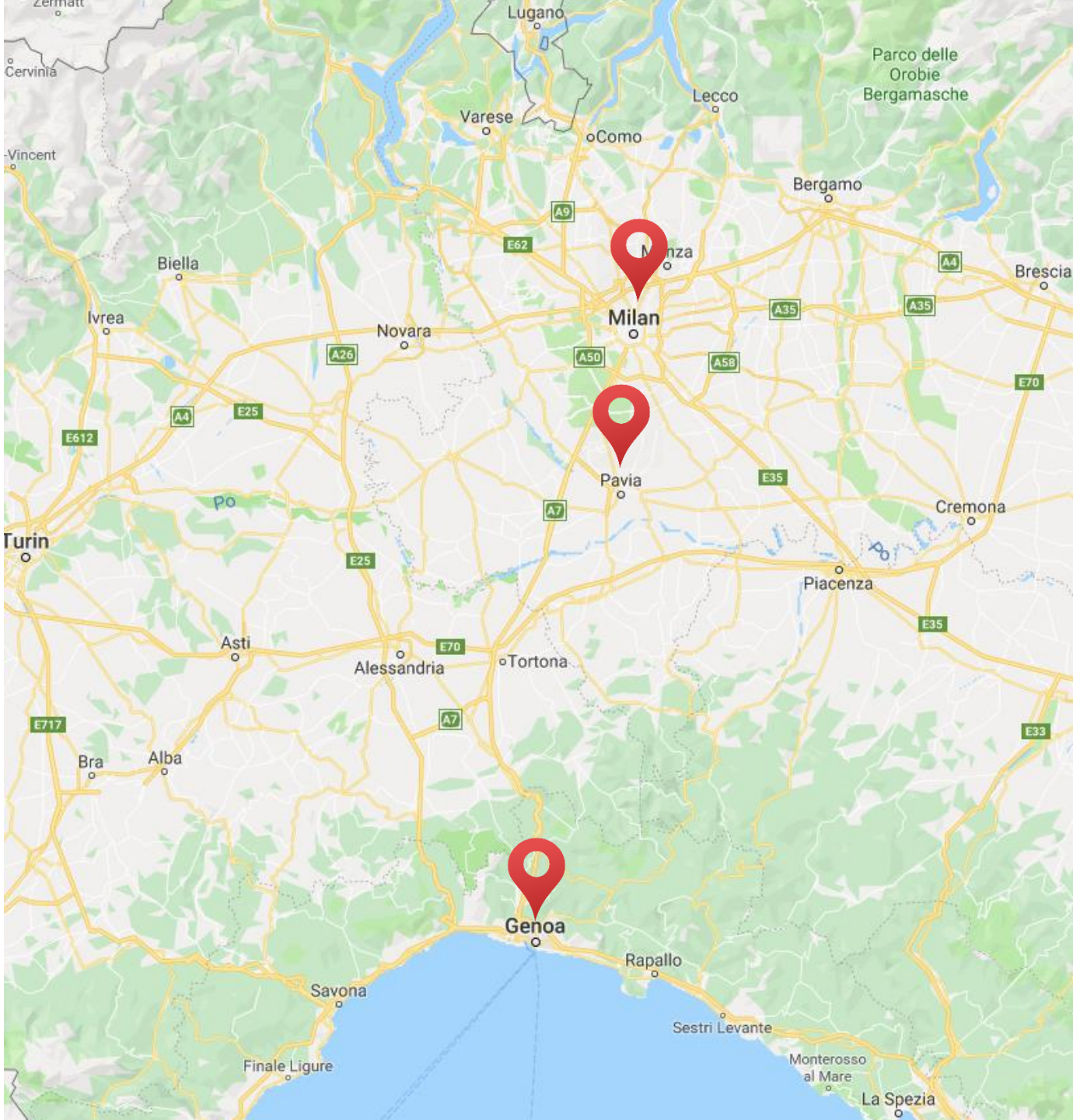
The scientific network:
105 Institutes,
7 Departments
17 Research Areas,
8.000 employees

imati

The Institute for Applied Mathematics and Information Technologies "Enrico Magenes" performs research in many areas of mathematics, computer science and their applications.

Three units:

- Pavia: *differential modeling and PDEs*
- Milano: *Stochastic Modeling and Data analysis*
- Genova: *Shape and Semanitics Modeling, Computing Architectures and HPC*



imati

The Institute for Ap
Information Technolo
performs research
mathematics, compu
applications.

Three units:

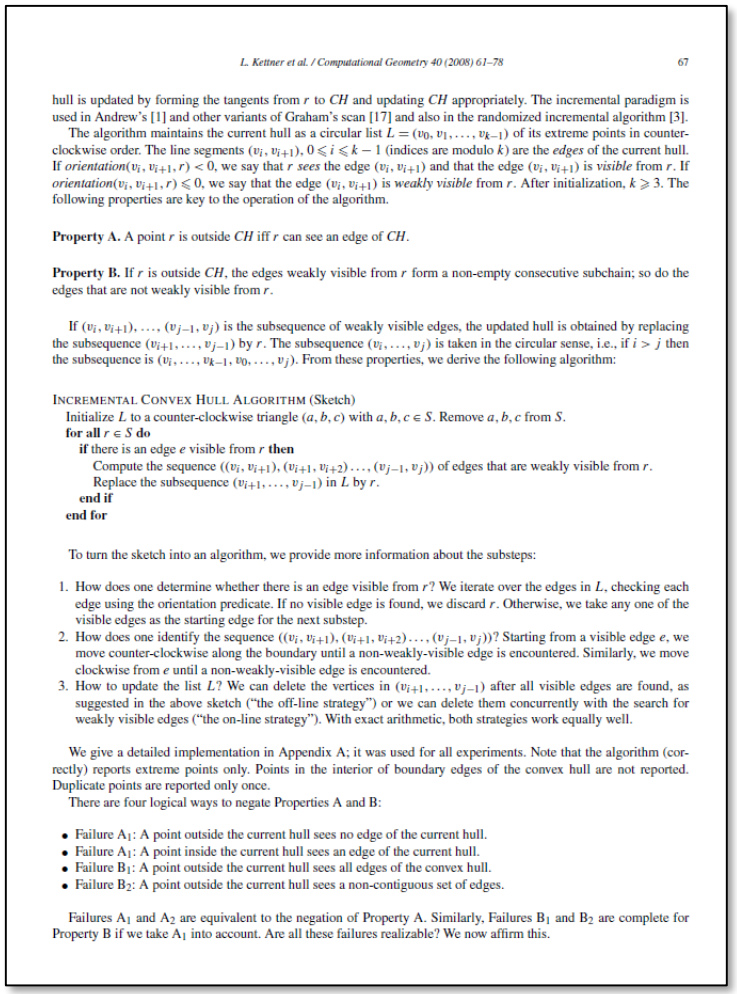
- Pavia: *differential modeling and PDEs*
- Milano: *Stochastic Modeling and Data analysis*
- Genova: *Shape and Semanitics Modeling, Computing Architectures and HPC*

Outline

- Geometric algorithm implementation – potential pitfalls
- Mesh repairing
- Robust geometric programming
- Libraries and paradigms
- Conclusions

Geometric Algorithm Implementation

Geometric Algorithm Implementation



hull is updated by forming the tangents from r to CH and updating CH appropriately. The incremental paradigm is used in Andrew's [1] and other variants of Graham's scan [17] and also in the randomized incremental algorithm [3].

The algorithm maintains the current hull as a circular list $L = (v_0, v_1, \ldots, v_{k-1})$ of its extreme points in counter-clockwise order. The line segments (v_i, v_{i+1}), $0 \leqslant i \leqslant k-1$ (indices are modulo k) are the *edges* of the current hull. If $\mathit{orientation}(v_i, v_{i+1}, r) < 0$, we say that r *sees* the edge (v_i, v_{i+1}) and that the edge (v_i, v_{i+1}) is *visible* from r. If $\mathit{orientation}(v_i, v_{i+1}, r) \leqslant 0$, we say that the edge (v_i, v_{i+1}) is *weakly visible* from r. After initialization, $k \geqslant 3$. The following properties are key to the operation of the algorithm.

Property A. A point r is outside CH iff r can see an edge of CH.

Property B. If r is outside CH, the edges weakly visible from r form a non-empty consecutive subchain; so do the edges that are not weakly visible from r.

If $(v_i, v_{i+1}), \ldots, (v_{j-1}, v_j)$ is the subsequence of weakly visible edges, the updated hull is obtained by replacing the subsequence $(v_{i+1}, \ldots, v_{j-1})$ by r. The subsequence $(v_i, \ldots, v_j)$ is taken in the circular sense, i.e., if $i > j$ then the subsequence is $(v_i, \ldots, v_{k-1}, v_0, \ldots, v_j)$. From these properties, we derive the following algorithm:

INCREMENTAL CONVEX HULL ALGORITHM (Sketch)
Initialize L to a counter-clockwise triangle (a, b, c) with $a, b, c \in S$. Remove a, b, c from S.
for all $r \in S$ **do**
  if there is an edge e visible from r **then**
    Compute the sequence $((v_i, v_{i+1}), (v_{i+1}, v_{i+2}) \ldots, (v_{j-1}, v_j))$ of edges that are weakly visible from r.
    Replace the subsequence $(v_{i+1}, \ldots, v_{j-1})$ in L by r.
  end if
end for

To turn the sketch into an algorithm, we provide more information about the substeps:

1. How does one determine whether there is an edge visible from r? We iterate over the edges in L, checking each edge using the orientation predicate. If no visible edge is found, we discard r. Otherwise, we take any one of the visible edges as the starting edge for the next substep.
2. How does one identify the sequence $((v_i, v_{i+1}), (v_{i+1}, v_{i+2}) \ldots, (v_{j-1}, v_j))$? Starting from a visible edge e, we move counter-clockwise along the boundary until a non-weakly-visible edge is encountered. Similarly, we move clockwise from e until a non-weakly-visible edge is encountered.
3. How to update the list L? We can delete the vertices in $(v_{i+1}, \ldots, v_{j-1})$ after all visible edges are found, as suggested in the above sketch ("the off-line strategy") or we can delete them concurrently with the search for weakly visible edges ("the on-line strategy"). With exact arithmetic, both strategies work equally well.

We give a detailed implementation in Appendix A; it was used for all experiments. Note that the algorithm (correctly) reports extreme points only. Points in the interior of boundary edges of the convex hull are not reported. Duplicate points are reported only once.

There are four logical ways to negate Properties A and B:

- Failure A_1: A point outside the current hull sees no edge of the current hull.
- Failure A_1: A point inside the current hull sees an edge of the current hull.
- Failure B_1: A point outside the current hull sees all edges of the convex hull.
- Failure B_2: A point outside the current hull sees a non-contiguous set of edges.

Failures A_1 and A_2 are equivalent to the negation of Property A. Similarly, Failures B_1 and B_2 are complete for Property B if we take A_1 into account. Are all these failures realizable? We now affirm this.

Geometric Algorithm Implementation

hull is updated by forming the tangents from r to CH and updating CH appropriately. The incremental paradigm is used in Andrew's [1] and other variants of Graham's scan [17] and also in the randomized incremental algorithm [3].

The algorithm maintains the current hull as a circular list $L = (v_0, v_1, \ldots, v_{k-1})$ of its extreme points in counter-clockwise order. The line segments (v_i, v_{i+1}), $0 \leqslant i \leqslant k-1$ (indices are modulo k) are the *edges* of the current hull. If $\mathit{orientation}(v_i, v_{i+1}, r) < 0$, we say that r *sees* the edge (v_i, v_{i+1}) and that the edge (v_i, v_{i+1}) is *visible* from r. If $\mathit{orientation}(v_i, v_{i+1}, r) \leqslant 0$, we say that the edge (v_i, v_{i+1}) is *weakly visible* from r. After initialization, $k \geqslant 3$. The following properties are key to the operation of the algorithm.

Property A. A point r is outside CH iff r can see an edge of CH.

Property B. If r is outside CH, the edges weakly visible from r form a non-empty consecutive subchain; so do the edges that are not weakly visible from r.

If $(v_i, v_{i+1}), \ldots, (v_{j-1}, v_j)$ is the subsequence of weakly visible edges, the updated hull is obtained by replacing the subsequence $(v_{i+1}, \ldots, v_{j-1})$ by r. The subsequence $(v_i, \ldots, v_j)$ is taken in the circular sense, i.e., if $i > j$ then the subsequence is $(v_i, \ldots, v_{k-1}, v_0, \ldots, v_j)$. From these properties, we derive the following algorithm:

INCREMENTAL CONVEX HULL ALGORITHM (Sketch)
Initialize L to a counter-clockwise triangle (a, b, c) with $a, b, c \in S$. Remove a, b, c from S.
for all $r \in S$ **do**
  if there is an edge e visible from r **then**
    Compute the sequence $((v_i, v_{i+1}), (v_{i+1}, v_{i+2}) \ldots, (v_{j-1}, v_j))$ of edges that are weakly visible from r.
    Replace the subsequence $(v_{i+1}, \ldots, v_{j-1})$ in L by r.
  end if
end for

To turn the sketch into an algorithm, we provide more information about the substeps:

1. How does one determine whether there is an edge visible from r? We iterate over the edges in L, checking each edge using the orientation predicate. If no visible edge is found, we discard r. Otherwise, we take any one of the visible edges as the starting edge for the next substep.
2. How does one identify the sequence $((v_i, v_{i+1}), (v_{i+1}, v_{i+2}) \ldots, (v_{j-1}, v_j))$? Starting from a visible edge e, we move counter-clockwise along the boundary until a non-weakly-visible edge is encountered. Similarly, we move clockwise from e until a non-weakly-visible edge is encountered.
3. How to update the list L? We can delete the vertices in $(v_{i+1}, \ldots, v_{j-1})$ after all visible edges are found, as suggested in the above sketch ("the off-line strategy") or we can delete them concurrently with the search for weakly visible edges ("the on-line strategy"). With exact arithmetic, both strategies work equally well.

We give a detailed implementation in Appendix A; it was used for all experiments. Note that the algorithm (correctly) reports extreme points only. Points in the interior of boundary edges of the convex hull are not reported. Duplicate points are reported only once.

There are four logical ways to negate Properties A and B:

- Failure A_1: A point outside the current hull sees no edge of the current hull.
- Failure A_1: A point inside the current hull sees an edge of the current hull.
- Failure B_1: A point outside the current hull sees all edges of the convex hull.
- Failure B_2: A point outside the current hull sees a non-contiguous set of edges.

Failures A_1 and A_2 are equivalent to the negation of Property A. Similarly, Failures B_1 and B_2 are complete for Property B if we take A_1 into account. Are all these failures realizable? We now affirm this.

Geometric Algorithm Implementation

hull is updated by forming the tangents from r to CH and updating CH appropriately. The incremental paradigm is used in Andrew's [1] and other variants of Graham's scan [17] and also in the randomized incremental algorithm [3].

The algorithm maintains the current hull as a circular list $L = (v_0, v_1, \ldots, v_{k-1})$ of its extreme points in counter-clockwise order. The line segments (v_i, v_{i+1}), $0 \leqslant i \leqslant k-1$ (indices are modulo k) are the *edges* of the current hull. If $\mathit{orientation}(v_i, v_{i+1}, r) < 0$, we say that r *sees* the edge (v_i, v_{i+1}) and that the edge (v_i, v_{i+1}) is *visible* from r. If $\mathit{orientation}(v_i, v_{i+1}, r) \leqslant 0$, we say that the edge (v_i, v_{i+1}) is *weakly visible* from r. After initialization, $k \geqslant 3$. The following properties are key to the operation of the algorithm.

Property A. A point r is outside CH iff r can see an edge of CH.

Property B. If r is outside CH, the edges weakly visible from r form a non-empty consecutive subchain; so do the edges that are not weakly visible from r.

If $(v_i, v_{i+1}), \ldots, (v_{j-1}, v_j)$ is the subsequence of weakly visible edges, the updated hull is obtained by replacing the subsequence $(v_{i+1}, \ldots, v_{j-1})$ by r. The subsequence $(v_i, \ldots, v_j)$ is taken in the circular sense, i.e., if $i > j$ then the subsequence is $(v_i, \ldots, v_{k-1}, v_0, \ldots, v_j)$. From these properties, we derive the following algorithm:

INCREMENTAL CONVEX HULL ALGORITHM (Sketch)
Initialize L to a counter-clockwise triangle (a, b, c) with $a, b, c \in S$. Remove a, b, c from S.
for all $r \in S$ **do**
  if there is an edge e visible from r **then**
    Compute the sequence $((v_i, v_{i+1}), (v_{i+1}, v_{i+2}) \ldots, (v_{j-1}, v_j))$ of edges that are weakly visible from r.
    Replace the subsequence $(v_{i+1}, \ldots, v_{j-1})$ in L by r.
  end if
end for

To turn the sketch into an algorithm, we provide more information about the substeps:

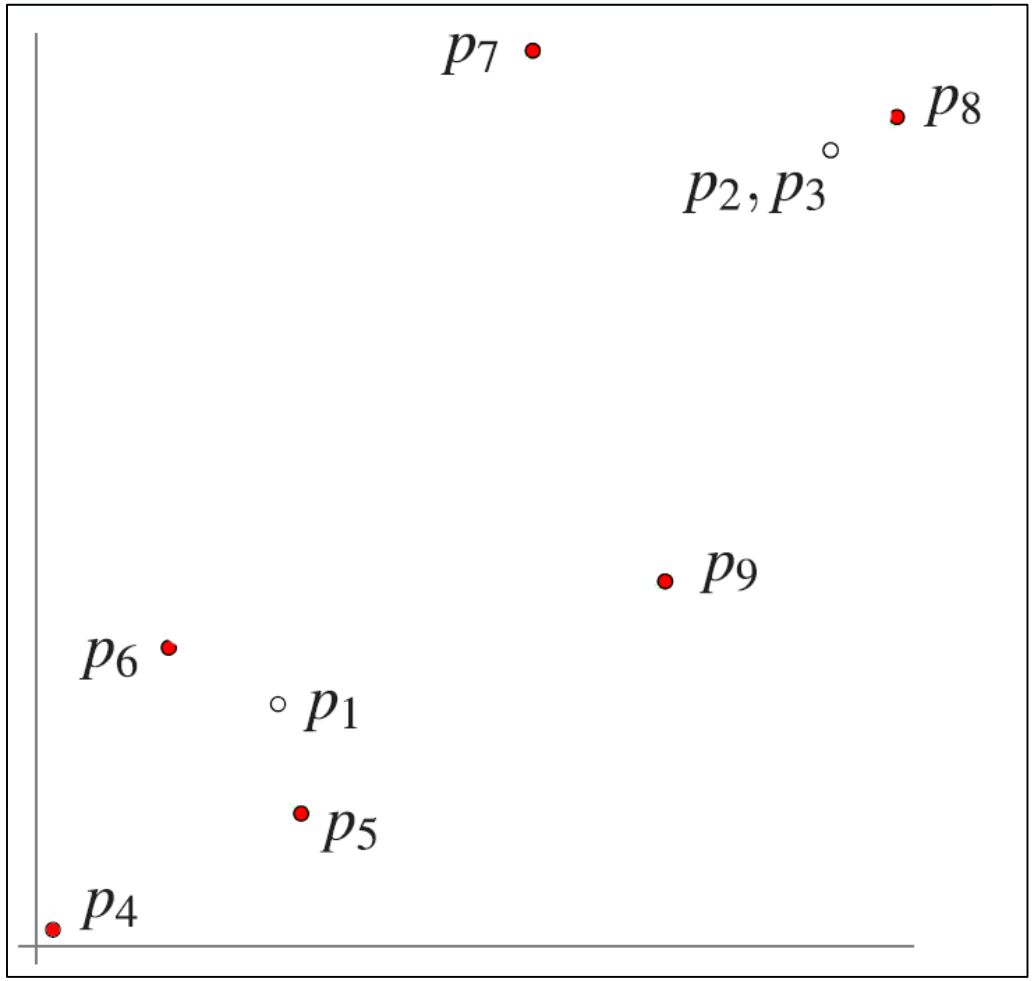
1. How does one determine whether there is an edge visible from r? We iterate over the edges in L, checking each edge using the orientation predicate. If no visible edge is found, we discard r. Otherwise, we take any one of the visible edges as the starting edge for the next substep.
2. How does one identify the sequence $((v_i, v_{i+1}), (v_{i+1}, v_{i+2}) \ldots, (v_{j-1}, v_j))$? Starting from a visible edge e, we move counter-clockwise along the boundary until a non-weakly-visible edge is encountered. Similarly, we move clockwise from e until a non-weakly-visible edge is encountered.
3. How to update the list L? We can delete the vertices in $(v_{i+1}, \ldots, v_{j-1})$ after all visible edges are found, as suggested in the above sketch ("the off-line strategy") or we can delete them concurrently with the search for weakly visible edges ("the on-line strategy"). With exact arithmetic, both strategies work equally well.

We give a detailed implementation in Appendix A; it was used for all experiments. Note that the algorithm (correctly) reports extreme points only. Points in the interior of boundary edges of the convex hull are not reported. Duplicate points are reported only once.

There are four logical ways to negate Properties A and B:

- Failure A_1: A point outside the current hull sees no edge of the current hull.
- Failure A_1: A point inside the current hull sees an edge of the current hull.
- Failure B_1: A point outside the current hull sees all edges of the convex hull.
- Failure B_2: A point outside the current hull sees a non-contiguous set of edges.

Failures A_1 and A_2 are equivalent to the negation of Property A. Similarly, Failures B_1 and B_2 are complete for Property B if we take A_1 into account. Are all these failures realizable? We now affirm this.



Geometric Algorithm Implementation

hull is updated by forming the tangents from r to CH and updating CH appropriately. The incremental paradigm is used in Andrew's [1] and other variants of Graham's scan [17] and also in the randomized incremental algorithm [3].

The algorithm maintains the current hull as a circular list $L = (v_0, v_1, \ldots, v_{k-1})$ of its extreme points in counter-clockwise order. The line segments (v_i, v_{i+1}), $0 \leqslant i \leqslant k-1$ (indices are modulo k) are the *edges* of the current hull. If $\mathit{orientation}(v_i, v_{i+1}, r) < 0$, we say that r *sees* the edge (v_i, v_{i+1}) and that the edge (v_i, v_{i+1}) is *visible* from r. If $\mathit{orientation}(v_i, v_{i+1}, r) \leqslant 0$, we say that the edge (v_i, v_{i+1}) is *weakly visible* from r. After initialization, $k \geqslant 3$. The following properties are key to the operation of the algorithm.

Property A. A point r is outside CH iff r can see an edge of CH.

Property B. If r is outside CH, the edges weakly visible from r form a non-empty consecutive subchain; so do the edges that are not weakly visible from r.

If $(v_i, v_{i+1}), \ldots, (v_{j-1}, v_j)$ is the subsequence of weakly visible edges, the updated hull is obtained by replacing the subsequence $(v_{i+1}, \ldots, v_{j-1})$ by r. The subsequence $(v_i, \ldots, v_j)$ is taken in the circular sense, i.e., if $i > j$ then the subsequence is $(v_i, \ldots, v_{k-1}, v_0, \ldots, v_j)$. From these properties, we derive the following algorithm:

INCREMENTAL CONVEX HULL ALGORITHM (Sketch)
Initialize L to a counter-clockwise triangle (a, b, c) with $a, b, c \in S$. Remove a, b, c from S.
for all $r \in S$ **do**
  if there is an edge e visible from r **then**
    Compute the sequence $((v_i, v_{i+1}), (v_{i+1}, v_{i+2}) \ldots, (v_{j-1}, v_j))$ of edges that are weakly visible from r.
    Replace the subsequence $(v_{i+1}, \ldots, v_{j-1})$ in L by r.
  end if
end for

To turn the sketch into an algorithm, we provide more information about the substeps:

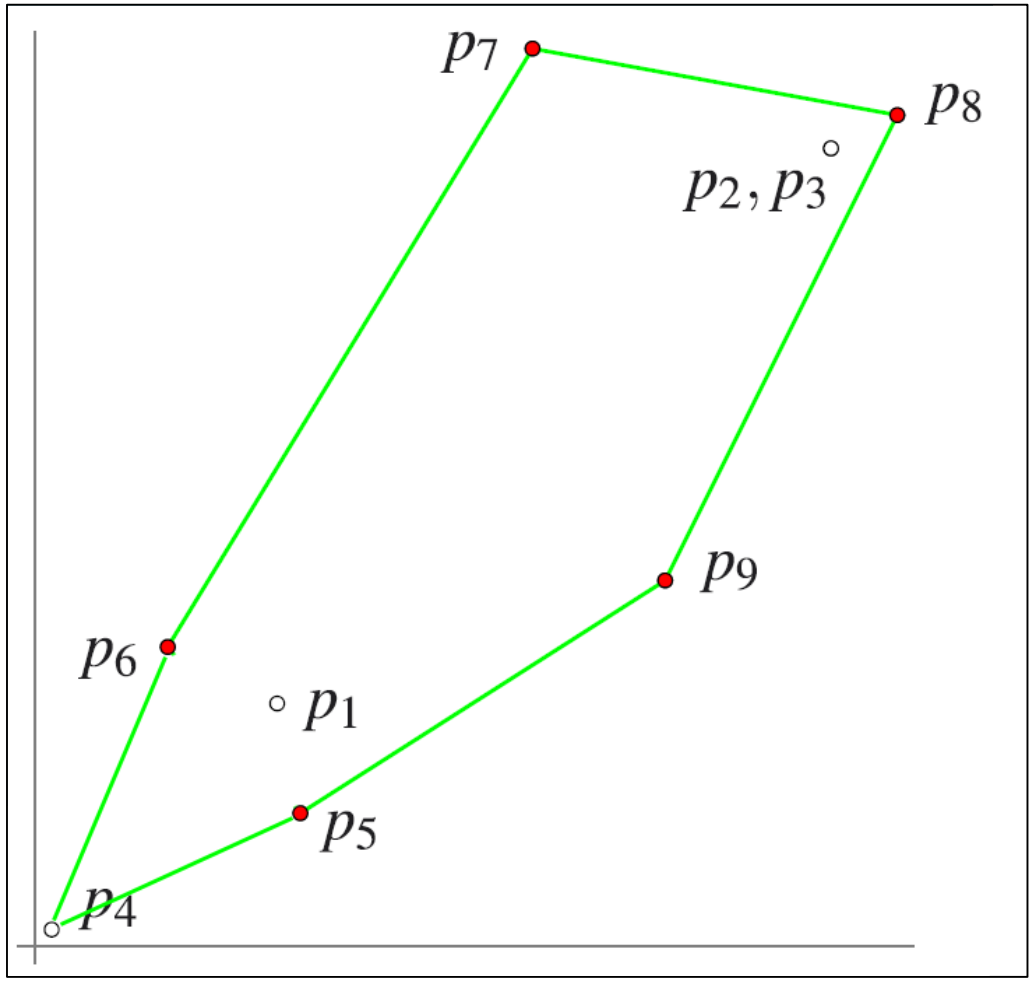
1. How does one determine whether there is an edge visible from r? We iterate over the edges in L, checking each edge using the orientation predicate. If no visible edge is found, we discard r. Otherwise, we take any one of the visible edges as the starting edge for the next substep.
2. How does one identify the sequence $((v_i, v_{i+1}), (v_{i+1}, v_{i+2}) \ldots, (v_{j-1}, v_j))$? Starting from a visible edge e, we move counter-clockwise along the boundary until a non-weakly-visible edge is encountered. Similarly, we move clockwise from e until a non-weakly-visible edge is encountered.
3. How to update the list L? We can delete the vertices in $(v_{i+1}, \ldots, v_{j-1})$ after all visible edges are found, as suggested in the above sketch ("the off-line strategy") or we can delete them concurrently with the search for weakly visible edges ("the on-line strategy"). With exact arithmetic, both strategies work equally well.

We give a detailed implementation in Appendix A; it was used for all experiments. Note that the algorithm (correctly) reports extreme points only. Points in the interior of boundary edges of the convex hull are not reported. Duplicate points are reported only once.

There are four logical ways to negate Properties A and B:

- Failure A_1: A point outside the current hull sees no edge of the current hull.
- Failure A_1: A point inside the current hull sees an edge of the current hull.
- Failure B_1: A point outside the current hull sees all edges of the convex hull.
- Failure B_2: A point outside the current hull sees a non-contiguous set of edges.

Failures A_1 and A_2 are equivalent to the negation of Property A. Similarly, Failures B_1 and B_2 are complete for Property B if we take A_1 into account. Are all these failures realizable? We now affirm this.



Geometric Algorithm Implementation

hull is updated by forming the tangents from r to CH and updating CH appropriately. The incremental paradigm is used in Andrew's [1] and other variants of Graham's scan [17] and also in the randomized incremental algorithm [3].

The algorithm maintains the current hull as a circular list $L = (v_0, v_1, \ldots, v_{k-1})$ of its extreme points in counter-clockwise order. The line segments (v_i, v_{i+1}), $0 \leqslant i \leqslant k-1$ (indices are modulo k) are the *edges* of the current hull. If $\mathit{orientation}(v_i, v_{i+1}, r) < 0$, we say that r *sees* the edge (v_i, v_{i+1}) and that the edge (v_i, v_{i+1}) is *visible* from r. If $\mathit{orientation}(v_i, v_{i+1}, r) \leqslant 0$, we say that the edge (v_i, v_{i+1}) is *weakly visible* from r. After initialization, $k \geqslant 3$. The following properties are key to the operation of the algorithm.

Property A. A point r is outside CH iff r can see an edge of CH.

Property B. If r is outside CH, the edges weakly visible from r form a non-empty consecutive subchain; so do the edges that are not weakly visible from r.

If $(v_i, v_{i+1}), \ldots, (v_{j-1}, v_j)$ is the subsequence of weakly visible edges, the updated hull is obtained by replacing the subsequence $(v_{i+1}, \ldots, v_{j-1})$ by r. The subsequence $(v_i, \ldots, v_j)$ is taken in the circular sense, i.e., if $i > j$ then the subsequence is $(v_i, \ldots, v_{k-1}, v_0, \ldots, v_j)$. From these properties, we derive the following algorithm:

INCREMENTAL CONVEX HULL ALGORITHM (Sketch)
Initialize L to a counter-clockwise triangle (a, b, c) with $a, b, c \in S$. Remove a, b, c from S.
for all $r \in S$ **do**
  if there is an edge e visible from r **then**
    Compute the sequence $((v_i, v_{i+1}), (v_{i+1}, v_{i+2}) \ldots, (v_{j-1}, v_j))$ of edges that are weakly visible from r.
    Replace the subsequence $(v_{i+1}, \ldots, v_{j-1})$ in L by r.
  end if
end for

To turn the sketch into an algorithm, we provide more information about the substeps:

1. How does one determine whether there is an edge visible from r? We iterate over the edges in L, checking each edge using the orientation predicate. If no visible edge is found, we discard r. Otherwise, we take any one of the visible edges as the starting edge for the next substep.
2. How does one identify the sequence $((v_i, v_{i+1}), (v_{i+1}, v_{i+2}) \ldots, (v_{j-1}, v_j))$? Starting from a visible edge e, we move counter-clockwise along the boundary until a non-weakly-visible edge is encountered. Similarly, we move clockwise from e until a non-weakly-visible edge is encountered.
3. How to update the list L? We can delete the vertices in $(v_{i+1}, \ldots, v_{j-1})$ after all visible edges are found, as suggested in the above sketch ("the off-line strategy") or we can delete them concurrently with the search for weakly visible edges ("the on-line strategy"). With exact arithmetic, both strategies work equally well.

We give a detailed implementation in Appendix A; it was used for all experiments. Note that the algorithm (correctly) reports extreme points only. Points in the interior of boundary edges of the convex hull are not reported. Duplicate points are reported only once.

There are four logical ways to negate Properties A and B:

- Failure A_1: A point outside the current hull sees no edge of the current hull.
- Failure A_1: A point inside the current hull sees an edge of the current hull.
- Failure B_1: A point outside the current hull sees all edges of the convex hull.
- Failure B_2: A point outside the current hull sees a non-contiguous set of edges.

Failures A_1 and A_2 are equivalent to the negation of Property A. Similarly, Failures B_1 and B_2 are complete for Property B if we take A_1 into account. Are all these failures realizable? We now affirm this.

But, in real world…

But, in real world…

hull is updated by forming the tangents from r to CH and updating CH appropriately. The incremental paradigm is used in Andrew's [1] and other variants of Graham's scan [17] and also in the randomized incremental algorithm [3].

The algorithm maintains the current hull as a circular list $L = (v_0, v_1, \ldots, v_{k-1})$ of its extreme points in counter-clockwise order. The line segments (v_i, v_{i+1}), $0 \leqslant i \leqslant k-1$ (indices are modulo k) are the *edges* of the current hull. If $\mathit{orientation}(v_i, v_{i+1}, r) < 0$, we say that r *sees* the edge (v_i, v_{i+1}) and that the edge (v_i, v_{i+1}) is *visible* from r. If $\mathit{orientation}(v_i, v_{i+1}, r) \leqslant 0$, we say that the edge (v_i, v_{i+1}) is *weakly visible* from r. After initialization, $k \geqslant 3$. The following properties are key to the operation of the algorithm.

Property A. A point r is outside CH iff r can see an edge of CH.

Property B. If r is outside CH, the edges weakly visible from r form a non-empty consecutive subchain; so do the edges that are not weakly visible from r.

If $(v_i, v_{i+1}), \ldots, (v_{j-1}, v_j)$ is the subsequence of weakly visible edges, the updated hull is obtained by replacing the subsequence $(v_{i+1}, \ldots, v_{j-1})$ by r. The subsequence $(v_i, \ldots, v_j)$ is taken in the circular sense, i.e., if $i > j$ then the subsequence is $(v_i, \ldots, v_{k-1}, v_0, \ldots, v_j)$. From these properties, we derive the following algorithm:

INCREMENTAL CONVEX HULL ALGORITHM (Sketch)
Initialize L to a counter-clockwise triangle (a, b, c) with $a, b, c \in S$. Remove a, b, c from S.
for all $r \in S$ **do**
  if there is an edge e visible from r **then**
    Compute the sequence $((v_i, v_{i+1}), (v_{i+1}, v_{i+2}) \ldots, (v_{j-1}, v_j))$ of edges that are weakly visible from r.
    Replace the subsequence $(v_{i+1}, \ldots, v_{j-1})$ in L by r.
  end if
end for

To turn the sketch into an algorithm, we provide more information about the substeps:

1. How does one determine whether there is an edge visible from r? We iterate over the edges in L, checking each edge using the orientation predicate. If no visible edge is found, we discard r. Otherwise, we take any one of the visible edges as the starting edge for the next substep.
2. How does one identify the sequence $((v_i, v_{i+1}), (v_{i+1}, v_{i+2}) \ldots, (v_{j-1}, v_j))$? Starting from a visible edge e, we move counter-clockwise along the boundary until a non-weakly-visible edge is encountered. Similarly, we move clockwise from e until a non-weakly-visible edge is encountered.
3. How to update the list L? We can delete the vertices in $(v_{i+1}, \ldots, v_{j-1})$ after all visible edges are found, as suggested in the above sketch ("the off-line strategy") or we can delete them concurrently with the search for weakly visible edges ("the on-line strategy"). With exact arithmetic, both strategies work equally well.

We give a detailed implementation in Appendix A; it was used for all experiments. Note that the algorithm (correctly) reports extreme points only. Points in the interior of boundary edges of the convex hull are not reported. Duplicate points are reported only once.

There are four logical ways to negate Properties A and B:

- Failure A_1: A point outside the current hull sees no edge of the current hull.
- Failure A_1: A point inside the current hull sees an edge of the current hull.
- Failure B_1: A point outside the current hull sees all edges of the convex hull.
- Failure B_2: A point outside the current hull sees a non-contiguous set of edges.

Failures A_1 and A_2 are equivalent to the negation of Property A. Similarly, Failures B_1 and B_2 are complete for Property B if we take A_1 into account. Are all these failures realizable? We now affirm this.

But, in real world…

hull is updated by forming the tangents from r to CH and updating CH appropriately. The incremental paradigm is used in Andrew's [1] and other variants of Graham's scan [17] and also in the randomized incremental algorithm [3].

The algorithm maintains the current hull as a circular list $L = (v_0, v_1, \ldots, v_{k-1})$ of its extreme points in counter-clockwise order. The line segments (v_i, v_{i+1}), $0 \leqslant i \leqslant k-1$ (indices are modulo k) are the *edges* of the current hull. If $\mathit{orientation}(v_i, v_{i+1}, r) < 0$, we say that r *sees* the edge (v_i, v_{i+1}) and that the edge (v_i, v_{i+1}) is *visible* from r. If $\mathit{orientation}(v_i, v_{i+1}, r) \leqslant 0$, we say that the edge (v_i, v_{i+1}) is *weakly visible* from r. After initialization, $k \geqslant 3$. The following properties are key to the operation of the algorithm.

Property A. A point r is outside CH iff r can see an edge of CH.

Property B. If r is outside CH, the edges weakly visible from r form a non-empty consecutive subchain; so do the edges that are not weakly visible from r.

If $(v_i, v_{i+1}), \ldots, (v_{j-1}, v_j)$ is the subsequence of weakly visible edges, the updated hull is obtained by replacing the subsequence $(v_{i+1}, \ldots, v_{j-1})$ by r. The subsequence $(v_i, \ldots, v_j)$ is taken in the circular sense, i.e., if $i > j$ then the subsequence is $(v_i, \ldots, v_{k-1}, v_0, \ldots, v_j)$. From these properties, we derive the following algorithm:

INCREMENTAL CONVEX HULL ALGORITHM (Sketch)
Initialize L to a counter-clockwise triangle (a, b, c) with $a, b, c \in S$. Remove a, b, c from S.
for all $r \in S$ **do**
  if there is an edge e visible from r **then**
    Compute the sequence $((v_i, v_{i+1}), (v_{i+1}, v_{i+2}) \ldots, (v_{j-1}, v_j))$ of edges that are weakly visible from r.
    Replace the subsequence $(v_{i+1}, \ldots, v_{j-1})$ in L by r.
  end if
end for

To turn the sketch into an algorithm, we provide more information about the substeps:

1. How does one determine whether there is an edge visible from r? We iterate over the edges in L, checking each edge using the orientation predicate. If no visible edge is found, we discard r. Otherwise, we take any one of the visible edges as the starting edge for the next substep.
2. How does one identify the sequence $((v_i, v_{i+1}), (v_{i+1}, v_{i+2}) \ldots, (v_{j-1}, v_j))$? Starting from a visible edge e, we move counter-clockwise along the boundary until a non-weakly-visible edge is encountered. Similarly, we move clockwise from e until a non-weakly-visible edge is encountered.
3. How to update the list L? We can delete the vertices in $(v_{i+1}, \ldots, v_{j-1})$ after all visible edges are found, as suggested in the above sketch ("the off-line strategy") or we can delete them concurrently with the search for weakly visible edges ("the on-line strategy"). With exact arithmetic, both strategies work equally well.

We give a detailed implementation in Appendix A; it was used for all experiments. Note that the algorithm (correctly) reports extreme points only. Points in the interior of boundary edges of the convex hull are not reported. Duplicate points are reported only once.

There are four logical ways to negate Properties A and B:

- Failure A_1: A point outside the current hull sees no edge of the current hull.
- Failure A_1: A point inside the current hull sees an edge of the current hull.
- Failure B_1: A point outside the current hull sees all edges of the convex hull.
- Failure B_2: A point outside the current hull sees a non-contiguous set of edges.

Failures A_1 and A_2 are equivalent to the negation of Property A. Similarly, Failures B_1 and B_2 are complete for Property B if we take A_1 into account. Are all these failures realizable? We now affirm this.

But, in real world…

hull is updated by forming the tangents from r to CH and updating CH appropriately. The incremental paradigm is used in Andrew's [1] and other variants of Graham's scan [17] and also in the randomized incremental algorithm [3].

The algorithm maintains the current hull as a circular list $L = (v_0, v_1, \ldots, v_{k-1})$ of its extreme points in counter-clockwise order. The line segments (v_i, v_{i+1}), $0 \leqslant i \leqslant k-1$ (indices are modulo k) are the *edges* of the current hull. If $\mathit{orientation}(v_i, v_{i+1}, r) < 0$, we say that r *sees* the edge (v_i, v_{i+1}) and that the edge (v_i, v_{i+1}) is *visible* from r. If $\mathit{orientation}(v_i, v_{i+1}, r) \leqslant 0$, we say that the edge (v_i, v_{i+1}) is *weakly visible* from r. After initialization, $k \geqslant 3$. The following properties are key to the operation of the algorithm.

Property A. A point r is outside CH iff r can see an edge of CH.

Property B. If r is outside CH, the edges weakly visible from r form a non-empty consecutive subchain; so do the edges that are not weakly visible from r.

If $(v_i, v_{i+1}), \ldots, (v_{j-1}, v_j)$ is the subsequence of weakly visible edges, the updated hull is obtained by replacing the subsequence $(v_{i+1}, \ldots, v_{j-1})$ by r. The subsequence $(v_i, \ldots, v_j)$ is taken in the circular sense, i.e., if $i > j$ then the subsequence is $(v_i, \ldots, v_{k-1}, v_0, \ldots, v_j)$. From these properties, we derive the following algorithm:

INCREMENTAL CONVEX HULL ALGORITHM (Sketch)
Initialize L to a counter-clockwise triangle (a, b, c) with $a, b, c \in S$. Remove a, b, c from S.
for all $r \in S$ **do**
  if there is an edge e visible from r **then**
    Compute the sequence $((v_i, v_{i+1}), (v_{i+1}, v_{i+2}) \ldots, (v_{j-1}, v_j))$ of edges that are weakly visible from r.
    Replace the subsequence $(v_{i+1}, \ldots, v_{j-1})$ in L by r.
  end if
end for

To turn the sketch into an algorithm, we provide more information about the substeps:

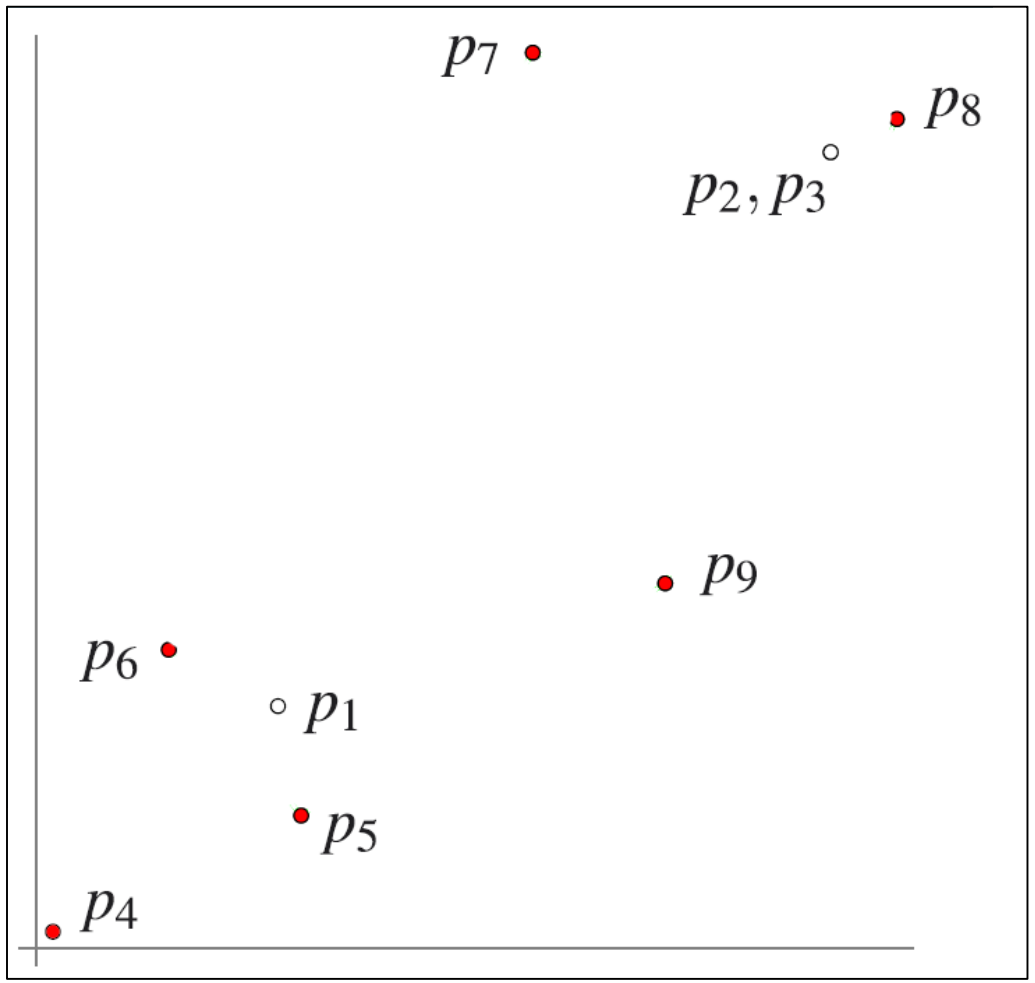
1. How does one determine whether there is an edge visible from r? We iterate over the edges in L, checking each edge using the orientation predicate. If no visible edge is found, we discard r. Otherwise, we take any one of the visible edges as the starting edge for the next substep.
2. How does one identify the sequence $((v_i, v_{i+1}), (v_{i+1}, v_{i+2}) \ldots, (v_{j-1}, v_j))$? Starting from a visible edge e, we move counter-clockwise along the boundary until a non-weakly-visible edge is encountered. Similarly, we move clockwise from e until a non-weakly-visible edge is encountered.
3. How to update the list L? We can delete the vertices in $(v_{i+1}, \ldots, v_{j-1})$ after all visible edges are found, as suggested in the above sketch ("the off-line strategy") or we can delete them concurrently with the search for weakly visible edges ("the on-line strategy"). With exact arithmetic, both strategies work equally well.

We give a detailed implementation in Appendix A; it was used for all experiments. Note that the algorithm (correctly) reports extreme points only. Points in the interior of boundary edges of the convex hull are not reported. Duplicate points are reported only once.

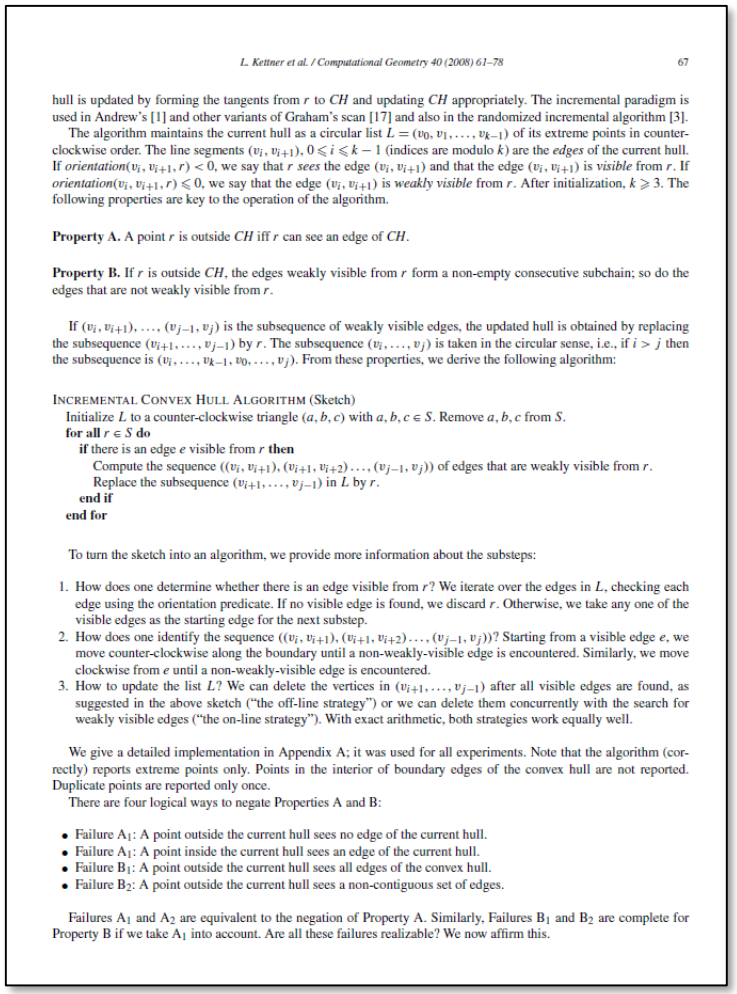
There are four logical ways to negate Properties A and B:

- Failure A_1: A point outside the current hull sees no edge of the current hull.
- Failure A_1: A point inside the current hull sees an edge of the current hull.
- Failure B_1: A point outside the current hull sees all edges of the convex hull.
- Failure B_2: A point outside the current hull sees a non-contiguous set of edges.

Failures A_1 and A_2 are equivalent to the negation of Property A. Similarly, Failures B_1 and B_2 are complete for Property B if we take A_1 into account. Are all these failures realizable? We now affirm this.



But, in real world…



hull is updated by forming the tangents from r to CH and updating CH appropriately. The incremental paradigm is used in Andrew's [1] and other variants of Graham's scan [17] and also in the randomized incremental algorithm [3].

The algorithm maintains the current hull as a circular list $L = (v_0, v_1, \ldots, v_{k-1})$ of its extreme points in counter-clockwise order. The line segments (v_i, v_{i+1}), $0 \leqslant i \leqslant k-1$ (indices are modulo k) are the *edges* of the current hull. If $\mathit{orientation}(v_i, v_{i+1}, r) < 0$, we say that r *sees* the edge (v_i, v_{i+1}) and that the edge (v_i, v_{i+1}) is *visible* from r. If $\mathit{orientation}(v_i, v_{i+1}, r) \leqslant 0$, we say that the edge (v_i, v_{i+1}) is *weakly visible* from r. After initialization, $k \geqslant 3$. The following properties are key to the operation of the algorithm.

Property A. A point r is outside CH iff r can see an edge of CH.

Property B. If r is outside CH, the edges weakly visible from r form a non-empty consecutive subchain; so do the edges that are not weakly visible from r.

If $(v_i, v_{i+1}), \ldots, (v_{j-1}, v_j)$ is the subsequence of weakly visible edges, the updated hull is obtained by replacing the subsequence $(v_{i+1}, \ldots, v_{j-1})$ by r. The subsequence $(v_i, \ldots, v_j)$ is taken in the circular sense, i.e., if $i > j$ then the subsequence is $(v_i, \ldots, v_{k-1}, v_0, \ldots, v_j)$. From these properties, we derive the following algorithm:

INCREMENTAL CONVEX HULL ALGORITHM (Sketch)
Initialize L to a counter-clockwise triangle (a, b, c) with $a, b, c \in S$. Remove a, b, c from S.
for all $r \in S$ **do**
  if there is an edge e visible from r **then**
    Compute the sequence $((v_i, v_{i+1}), (v_{i+1}, v_{i+2}) \ldots, (v_{j-1}, v_j))$ of edges that are weakly visible from r.
    Replace the subsequence $(v_{i+1}, \ldots, v_{j-1})$ in L by r.
  end if
end for

To turn the sketch into an algorithm, we provide more information about the substeps:

1. How does one determine whether there is an edge visible from r? We iterate over the edges in L, checking each edge using the orientation predicate. If no visible edge is found, we discard r. Otherwise, we take any one of the visible edges as the starting edge for the next substep.
2. How does one identify the sequence $((v_i, v_{i+1}), (v_{i+1}, v_{i+2}) \ldots, (v_{j-1}, v_j))$? Starting from a visible edge e, we move counter-clockwise along the boundary until a non-weakly-visible edge is encountered. Similarly, we move clockwise from e until a non-weakly-visible edge is encountered.
3. How to update the list L? We can delete the vertices in $(v_{i+1}, \ldots, v_{j-1})$ after all visible edges are found, as suggested in the above sketch ("the off-line strategy") or we can delete them concurrently with the search for weakly visible edges ("the on-line strategy"). With exact arithmetic, both strategies work equally well.

We give a detailed implementation in Appendix A; it was used for all experiments. Note that the algorithm (correctly) reports extreme points only. Points in the interior of boundary edges of the convex hull are not reported. Duplicate points are reported only once.

There are four logical ways to negate Properties A and B:

- Failure A_1: A point outside the current hull sees no edge of the current hull.
- Failure A_1: A point inside the current hull sees an edge of the current hull.
- Failure B_1: A point outside the current hull sees all edges of the convex hull.
- Failure B_2: A point outside the current hull sees a non-contiguous set of edges.

Failures A_1 and A_2 are equivalent to the negation of Property A. Similarly, Failures B_1 and B_2 are complete for Property B if we take A_1 into account. Are all these failures realizable? We now affirm this.

But, in real world…

hull is updated by forming the tangents from r to CH and updating CH appropriately. The incremental paradigm is used in Andrew's [1] and other variants of Graham's scan [17] and also in the randomized incremental algorithm [3].

The algorithm maintains the current hull as a circular list $L = (v_0, v_1, \ldots, v_{k-1})$ of its extreme points in counter-clockwise order. The line segments (v_i, v_{i+1}), $0 \leqslant i \leqslant k-1$ (indices are modulo k) are the *edges* of the current hull. If $\mathit{orientation}(v_i, v_{i+1}, r) < 0$, we say that r *sees* the edge (v_i, v_{i+1}) and that the edge (v_i, v_{i+1}) is *visible* from r. If $\mathit{orientation}(v_i, v_{i+1}, r) \leqslant 0$, we say that the edge (v_i, v_{i+1}) is *weakly visible* from r. After initialization, $k \geqslant 3$. The following properties are key to the operation of the algorithm.

Property A. A point r is outside CH iff r can see an edge of CH.

Property B. If r is outside CH, the edges weakly visible from r form a non-empty consecutive subchain; so do the edges that are not weakly visible from r.

If $(v_i, v_{i+1}), \ldots, (v_{j-1}, v_j)$ is the subsequence of weakly visible edges, the updated hull is obtained by replacing the subsequence $(v_{i+1}, \ldots, v_{j-1})$ by r. The subsequence $(v_i, \ldots, v_j)$ is taken in the circular sense, i.e., if $i > j$ then the subsequence is $(v_i, \ldots, v_{k-1}, v_0, \ldots, v_j)$. From these properties, we derive the following algorithm:

INCREMENTAL CONVEX HULL ALGORITHM (Sketch)
Initialize L to a counter-clockwise triangle (a, b, c) with $a, b, c \in S$. Remove a, b, c from S.
for all $r \in S$ **do**
  if there is an edge e visible from r **then**
    Compute the sequence $((v_i, v_{i+1}), (v_{i+1}, v_{i+2}) \ldots, (v_{j-1}, v_j))$ of edges that are weakly visible from r.
    Replace the subsequence $(v_{i+1}, \ldots, v_{j-1})$ in L by r.
  end if
end for

To turn the sketch into an algorithm, we provide more information about the substeps:

1. How does one determine whether there is an edge visible from r? We iterate over the edges in L, checking each edge using the orientation predicate. If no visible edge is found, we discard r. Otherwise, we take any one of the visible edges as the starting edge for the next substep.
2. How does one identify the sequence $((v_i, v_{i+1}), (v_{i+1}, v_{i+2}) \ldots, (v_{j-1}, v_j))$? Starting from a visible edge e, we move counter-clockwise along the boundary until a non-weakly-visible edge is encountered. Similarly, we move clockwise from e until a non-weakly-visible edge is encountered.
3. How to update the list L? We can delete the vertices in $(v_{i+1}, \ldots, v_{j-1})$ after all visible edges are found, as suggested in the above sketch ("the off-line strategy") or we can delete them concurrently with the search for weakly visible edges ("the on-line strategy"). With exact arithmetic, both strategies work equally well.

We give a detailed implementation in Appendix A; it was used for all experiments. Note that the algorithm (correctly) reports extreme points only. Points in the interior of boundary edges of the convex hull are not reported. Duplicate points are reported only once.

There are four logical ways to negate Properties A and B:

- Failure A_1: A point outside the current hull sees no edge of the current hull.
- Failure A_1: A point inside the current hull sees an edge of the current hull.
- Failure B_1: A point outside the current hull sees all edges of the convex hull.
- Failure B_2: A point outside the current hull sees a non-contiguous set of edges.

Failures A_1 and A_2 are equivalent to the negation of Property A. Similarly, Failures B_1 and B_2 are complete for Property B if we take A_1 into account. Are all these failures realizable? We now affirm this.

But, in real world…

hull is updated by forming the tangents from r to CH and updating CH appropriately. The incremental paradigm is used in Andrew's [1] and other variants of Graham's scan [17] and also in the randomized incremental algorithm [3].

The algorithm maintains the current hull as a circular list $L = (v_0, v_1, \ldots, v_{k-1})$ of its extreme points in counter-clockwise order. The line segments (v_i, v_{i+1}), $0 \leqslant i \leqslant k-1$ (indices are modulo k) are the *edges* of the current hull. If $\mathit{orientation}(v_i, v_{i+1}, r) < 0$, we say that r *sees* the edge (v_i, v_{i+1}) and that the edge (v_i, v_{i+1}) is *visible* from r. If $\mathit{orientation}(v_i, v_{i+1}, r) \leqslant 0$, we say that the edge (v_i, v_{i+1}) is *weakly visible* from r. After initialization, $k \geqslant 3$. The following properties are key to the operation of the algorithm.

Property A. A point r is outside CH iff r can see an edge of CH.

Property B. If r is outside CH, the edges weakly visible from r form a non-empty consecutive subchain; so do the edges that are not weakly visible from r.

If $(v_i, v_{i+1}), \ldots, (v_{j-1}, v_j)$ is the subsequence of weakly visible edges, the updated hull is obtained by replacing the subsequence $(v_{i+1}, \ldots, v_{j-1})$ by r. The subsequence $(v_i, \ldots, v_j)$ is taken in the circular sense, i.e., if $i > j$ then the subsequence is $(v_i, \ldots, v_{k-1}, v_0, \ldots, v_j)$. From these properties, we derive the following algorithm:

INCREMENTAL CONVEX HULL ALGORITHM (Sketch)
Initialize L to a counter-clockwise triangle (a, b, c) with $a, b, c \in S$. Remove a, b, c from S.
for all $r \in S$ **do**
  if there is an edge e visible from r **then**
    Compute the sequence $((v_i, v_{i+1}), (v_{i+1}, v_{i+2}) \ldots, (v_{j-1}, v_j))$ of edges that are weakly visible from r.
    Replace the subsequence $(v_{i+1}, \ldots, v_{j-1})$ in L by r.
  end if
end for

To turn the sketch into an algorithm, we provide more information about the substeps:

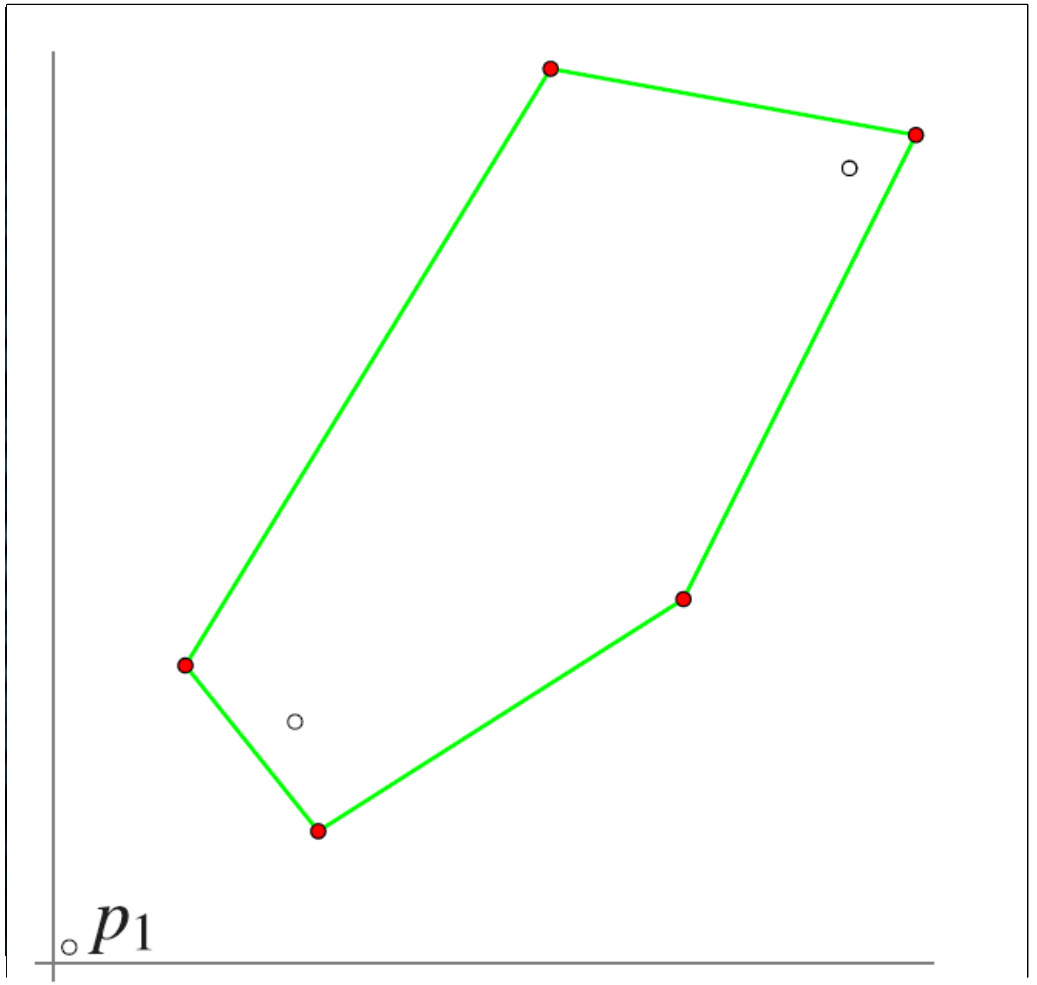
1. How does one determine whether there is an edge visible from r? We iterate over the edges in L, checking each edge using the orientation predicate. If no visible edge is found, we discard r. Otherwise, we take any one of the visible edges as the starting edge for the next substep.
2. How does one identify the sequence $((v_i, v_{i+1}), (v_{i+1}, v_{i+2}) \ldots, (v_{j-1}, v_j))$? Starting from a visible edge e, we move counter-clockwise along the boundary until a non-weakly-visible edge is encountered. Similarly, we move clockwise from e until a non-weakly-visible edge is encountered.
3. How to update the list L? We can delete the vertices in $(v_{i+1}, \ldots, v_{j-1})$ after all visible edges are found, as suggested in the above sketch ("the off-line strategy") or we can delete them concurrently with the search for weakly visible edges ("the on-line strategy"). With exact arithmetic, both strategies work equally well.

We give a detailed implementation in Appendix A; it was used for all experiments. Note that the algorithm (correctly) reports extreme points only. Points in the interior of boundary edges of the convex hull are not reported. Duplicate points are reported only once.

There are four logical ways to negate Properties A and B:

- Failure A_1: A point outside the current hull sees no edge of the current hull.
- Failure A_1: A point inside the current hull sees an edge of the current hull.
- Failure B_1: A point outside the current hull sees all edges of the convex hull.
- Failure B_2: A point outside the current hull sees a non-contiguous set of edges.

Failures A_1 and A_2 are equivalent to the negation of Property A. Similarly, Failures B_1 and B_2 are complete for Property B if we take A_1 into account. Are all these failures realizable? We now affirm this.



But, in real world…

hull is updated by forming the tangents from r to CH and updating CH appropriately. The incremental paradigm is used in Andrew's [1] and other variants of Graham's scan [17] and also in the randomized incremental algorithm [3].

The algorithm maintains the current hull as a circular list $L = (v_0, v_1, \ldots, v_{k-1})$ of its extreme points in counter-clockwise order. The line segments (v_i, v_{i+1}), $0 \leqslant i \leqslant k-1$ (indices are modulo k) are the *edges* of the current hull. If $\mathit{orientation}(v_i, v_{i+1}, r) < 0$, we say that r *sees* the edge (v_i, v_{i+1}) and that the edge (v_i, v_{i+1}) is *visible* from r. If $\mathit{orientation}(v_i, v_{i+1}, r) \leqslant 0$, we say that the edge (v_i, v_{i+1}) is *weakly visible* from r. After initialization, $k \geqslant 3$. The following properties are key to the operation of the algorithm.

Property A. A point r is outside CH iff r can see an edge of CH.

Property B. If r is outside CH, the edges weakly visible from r form a non-empty consecutive subchain; so do the edges that are not weakly visible from r.

If $(v_i, v_{i+1}), \ldots, (v_{j-1}, v_j)$ is the subsequence of weakly visible edges, the updated hull is obtained by replacing the subsequence $(v_{i+1}, \ldots, v_{j-1})$ by r. The subsequence $(v_i, \ldots, v_j)$ is taken in the circular sense, i.e., if $i > j$ then the subsequence is $(v_i, \ldots, v_{k-1}, v_0, \ldots, v_j)$. From these properties, we derive the following algorithm:

INCREMENTAL CONVEX HULL ALGORITHM (Sketch)
Initialize L to a counter-clockwise triangle (a, b, c) with $a, b, c \in S$. Remove a, b, c from S.
for all $r \in S$ **do**
  if there is an edge e visible from r **then**
    Compute the sequence $((v_i, v_{i+1}), (v_{i+1}, v_{i+2}) \ldots, (v_{j-1}, v_j))$ of edges that are weakly visible from r.
    Replace the subsequence $(v_{i+1}, \ldots, v_{j-1})$ in L by r.
  end if
end for

To turn the sketch into an algorithm, we provide more information about the substeps:

1. How does one determine whether there is an edge visible from r? We iterate over the edges in L, checking each edge using the orientation predicate. If no visible edge is found, we discard r. Otherwise, we take any one of the visible edges as the starting edge for the next substep.
2. How does one identify the sequence $((v_i, v_{i+1}), (v_{i+1}, v_{i+2}) \ldots, (v_{j-1}, v_j))$? Starting from a visible edge e, we move counter-clockwise along the boundary until a non-weakly-visible edge is encountered. Similarly, we move clockwise from e until a non-weakly-visible edge is encountered.
3. How to update the list L? We can delete the vertices in $(v_{i+1}, \ldots, v_{j-1})$ after all visible edges are found, as suggested in the above sketch ("the off-line strategy") or we can delete them concurrently with the search for weakly visible edges ("the on-line strategy"). With exact arithmetic, both strategies work equally well.

We give a detailed implementation in Appendix A; it was used for all experiments. Note that the algorithm (correctly) reports extreme points only. Points in the interior of boundary edges of the convex hull are not reported. Duplicate points are reported only once.

There are four logical ways to negate Properties A and B:

- Failure A_1: A point outside the current hull sees no edge of the current hull.
- Failure A_1: A point inside the current hull sees an edge of the current hull.
- Failure B_1: A point outside the current hull sees all edges of the convex hull.
- Failure B_2: A point outside the current hull sees a non-contiguous set of edges.

Failures A_1 and A_2 are equivalent to the negation of Property A. Similarly, Failures B_1 and B_2 are complete for Property B if we take A_1 into account. Are all these failures realizable? We now affirm this.

OK.

Let's try with another one…

But, in real world…

hull is updated by forming the tangents from r to CH and updating CH appropriately. The incremental paradigm is used in Andrew's [1] and other variants of Graham's scan [17] and also in the randomized incremental algorithm [3].

The algorithm maintains the current hull as a circular list $L = (v_0, v_1, \ldots, v_{k-1})$ of its extreme points in counter-clockwise order. The line segments (v_i, v_{i+1}), $0 \leqslant i \leqslant k-1$ (indices are modulo k) are the *edges* of the current hull. If $\mathit{orientation}(v_i, v_{i+1}, r) < 0$, we say that r *sees* the edge (v_i, v_{i+1}) and that the edge (v_i, v_{i+1}) is *visible* from r. If $\mathit{orientation}(v_i, v_{i+1}, r) \leqslant 0$, we say that the edge (v_i, v_{i+1}) is *weakly visible* from r. After initialization, $k \geqslant 3$. The following properties are key to the operation of the algorithm.

Property A. A point r is outside CH iff r can see an edge of CH.

Property B. If r is outside CH, the edges weakly visible from r form a non-empty consecutive subchain; so do the edges that are not weakly visible from r.

If $(v_i, v_{i+1}), \ldots, (v_{j-1}, v_j)$ is the subsequence of weakly visible edges, the updated hull is obtained by replacing the subsequence $(v_{i+1}, \ldots, v_{j-1})$ by r. The subsequence $(v_i, \ldots, v_j)$ is taken in the circular sense, i.e., if $i > j$ then the subsequence is $(v_i, \ldots, v_{k-1}, v_0, \ldots, v_j)$. From these properties, we derive the following algorithm:

INCREMENTAL CONVEX HULL ALGORITHM (Sketch)
Initialize L to a counter-clockwise triangle (a, b, c) with $a, b, c \in S$. Remove a, b, c from S.
for all $r \in S$ **do**
  if there is an edge e visible from r **then**
    Compute the sequence $((v_i, v_{i+1}), (v_{i+1}, v_{i+2}) \ldots, (v_{j-1}, v_j))$ of edges that are weakly visible from r.
    Replace the subsequence $(v_{i+1}, \ldots, v_{j-1})$ in L by r.
  end if
end for

To turn the sketch into an algorithm, we provide more information about the substeps:

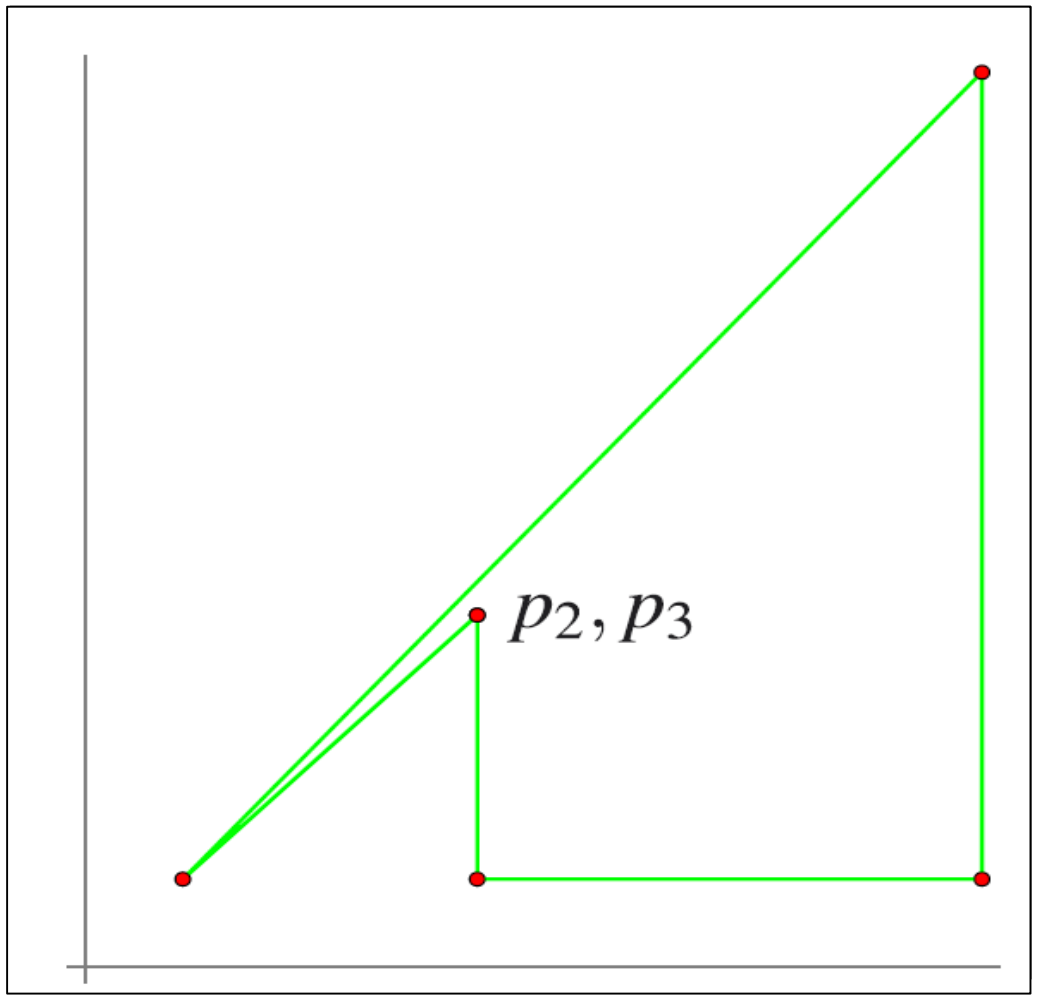
1. How does one determine whether there is an edge visible from r? We iterate over the edges in L, checking each edge using the orientation predicate. If no visible edge is found, we discard r. Otherwise, we take any one of the visible edges as the starting edge for the next substep.
2. How does one identify the sequence $((v_i, v_{i+1}), (v_{i+1}, v_{i+2}) \ldots, (v_{j-1}, v_j))$? Starting from a visible edge e, we move counter-clockwise along the boundary until a non-weakly-visible edge is encountered. Similarly, we move clockwise from e until a non-weakly-visible edge is encountered.
3. How to update the list L? We can delete the vertices in $(v_{i+1}, \ldots, v_{j-1})$ after all visible edges are found, as suggested in the above sketch ("the off-line strategy") or we can delete them concurrently with the search for weakly visible edges ("the on-line strategy"). With exact arithmetic, both strategies work equally well.

We give a detailed implementation in Appendix A; it was used for all experiments. Note that the algorithm (correctly) reports extreme points only. Points in the interior of boundary edges of the convex hull are not reported. Duplicate points are reported only once.

There are four logical ways to negate Properties A and B:

- Failure A_1: A point outside the current hull sees no edge of the current hull.
- Failure A_1: A point inside the current hull sees an edge of the current hull.
- Failure B_1: A point outside the current hull sees all edges of the convex hull.
- Failure B_2: A point outside the current hull sees a non-contiguous set of edges.

Failures A_1 and A_2 are equivalent to the negation of Property A. Similarly, Failures B_1 and B_2 are complete for Property B if we take A_1 into account. Are all these failures realizable? We now affirm this.



But, in real world…

hull is updated by forming the tangents from r to CH and updating CH appropriately. The incremental paradigm is used in Andrew's [1] and other variants of Graham's scan [17] and also in the randomized incremental algorithm [3].

The algorithm maintains the current hull as a circular list $L = (v_0, v_1, \ldots, v_{k-1})$ of its extreme points in counter-clockwise order. The line segments (v_i, v_{i+1}), $0 \leqslant i \leqslant k-1$ (indices are modulo k) are the *edges* of the current hull. If $\mathit{orientation}(v_i, v_{i+1}, r) < 0$, we say that r *sees* the edge (v_i, v_{i+1}) and that the edge (v_i, v_{i+1}) is *visible* from r. If $\mathit{orientation}(v_i, v_{i+1}, r) \leqslant 0$, we say that the edge (v_i, v_{i+1}) is *weakly visible* from r. After initialization, $k \geqslant 3$. The following properties are key to the operation of the algorithm.

Property A. A point r is outside CH iff r can see an edge of CH.

Property B. If r is outside CH, the edges weakly visible from r form a non-empty consecutive subchain; so do the edges that are not weakly visible from r.

If $(v_i, v_{i+1}), \ldots, (v_{j-1}, v_j)$ is the subsequence of weakly visible edges, the updated hull is obtained by replacing the subsequence $(v_{i+1}, \ldots, v_{j-1})$ by r. The subsequence $(v_i, \ldots, v_j)$ is taken in the circular sense, i.e., if $i > j$ then the subsequence is $(v_i, \ldots, v_{k-1}, v_0, \ldots, v_j)$. From these properties, we derive the following algorithm:

INCREMENTAL CONVEX HULL ALGORITHM (Sketch)
Initialize L to a counter-clockwise triangle (a, b, c) with $a, b, c \in S$. Remove a, b, c from S.
for all $r \in S$ **do**
  if there is an edge e visible from r **then**
    Compute the sequence $((v_i, v_{i+1}), (v_{i+1}, v_{i+2}) \ldots, (v_{j-1}, v_j))$ of edges that are weakly visible from r.
    Replace the subsequence $(v_{i+1}, \ldots, v_{j-1})$ in L by r.
  end if
end for

To turn the sketch into an algorithm, we provide more information about the substeps:

1. How does one determine whether there is an edge visible from r? We iterate over the edges in L, checking each edge using the orientation predicate. If no visible edge is found, we discard r. Otherwise, we take any one of the visible edges as the starting edge for the next substep.
2. How does one identify the sequence $((v_i, v_{i+1}), (v_{i+1}, v_{i+2}) \ldots, (v_{j-1}, v_j))$? Starting from a visible edge e, we move counter-clockwise along the boundary until a non-weakly-visible edge is encountered. Similarly, we move clockwise from e until a non-weakly-visible edge is encountered.
3. How to update the list L? We can delete the vertices in $(v_{i+1}, \ldots, v_{j-1})$ after all visible edges are found, as suggested in the above sketch ("the off-line strategy") or we can delete them concurrently with the search for weakly visible edges ("the on-line strategy"). With exact arithmetic, both strategies work equally well.

We give a detailed implementation in Appendix A; it was used for all experiments. Note that the algorithm (correctly) reports extreme points only. Points in the interior of boundary edges of the convex hull are not reported. Duplicate points are reported only once.

There are four logical ways to negate Properties A and B:

- Failure A_1: A point outside the current hull sees no edge of the current hull.
- Failure A_1: A point inside the current hull sees an edge of the current hull.
- Failure B_1: A point outside the current hull sees all edges of the convex hull.
- Failure B_2: A point outside the current hull sees a non-contiguous set of edges.

Failures A_1 and A_2 are equivalent to the negation of Property A. Similarly, Failures B_1 and B_2 are complete for Property B if we take A_1 into account. Are all these failures realizable? We now affirm this.

But, in real world…

hull is updated by forming the tangents from r to CH and updating CH appropriately. The incremental paradigm is used in Andrew's [1] and other variants of Graham's scan [17] and also in the randomized incremental algorithm [3].

The algorithm maintains the current hull as a circular list $L = (v_0, v_1, \ldots, v_{k-1})$ of its extreme points in counter-clockwise order. The line segments (v_i, v_{i+1}), $0 \leqslant i \leqslant k-1$ (indices are modulo k) are the *edges* of the current hull. If $\mathit{orientation}(v_i, v_{i+1}, r) < 0$, we say that r *sees* the edge (v_i, v_{i+1}) and that the edge (v_i, v_{i+1}) is *visible* from r. If $\mathit{orientation}(v_i, v_{i+1}, r) \leqslant 0$, we say that the edge (v_i, v_{i+1}) is *weakly visible* from r. After initialization, $k \geqslant 3$. The following properties are key to the operation of the algorithm.

Property A. A point r is outside CH iff r can see an edge of CH.

Property B. If r is outside CH, the edges weakly visible from r form a non-empty consecutive subchain; so do the edges that are not weakly visible from r.

If $(v_i, v_{i+1}), \ldots, (v_{j-1}, v_j)$ is the subsequence of weakly visible edges, the updated hull is obtained by replacing the subsequence $(v_{i+1}, \ldots, v_{j-1})$ by r. The subsequence $(v_i, \ldots, v_j)$ is taken in the circular sense, i.e., if $i > j$ then the subsequence is $(v_i, \ldots, v_{k-1}, v_0, \ldots, v_j)$. From these properties, we derive the following algorithm:

INCREMENTAL CONVEX HULL ALGORITHM (Sketch)
Initialize L to a counter-clockwise triangle (a, b, c) with $a, b, c \in S$. Remove a, b, c from S.
for all $r \in S$ **do**
  if there is an edge e visible from r **then**
    Compute the sequence $((v_i, v_{i+1}), (v_{i+1}, v_{i+2}) \ldots, (v_{j-1}, v_j))$ of edges that are weakly visible from r.
    Replace the subsequence $(v_{i+1}, \ldots, v_{j-1})$ in L by r.
  end if
end for

To turn the sketch into an algorithm, we provide more information about the substeps:

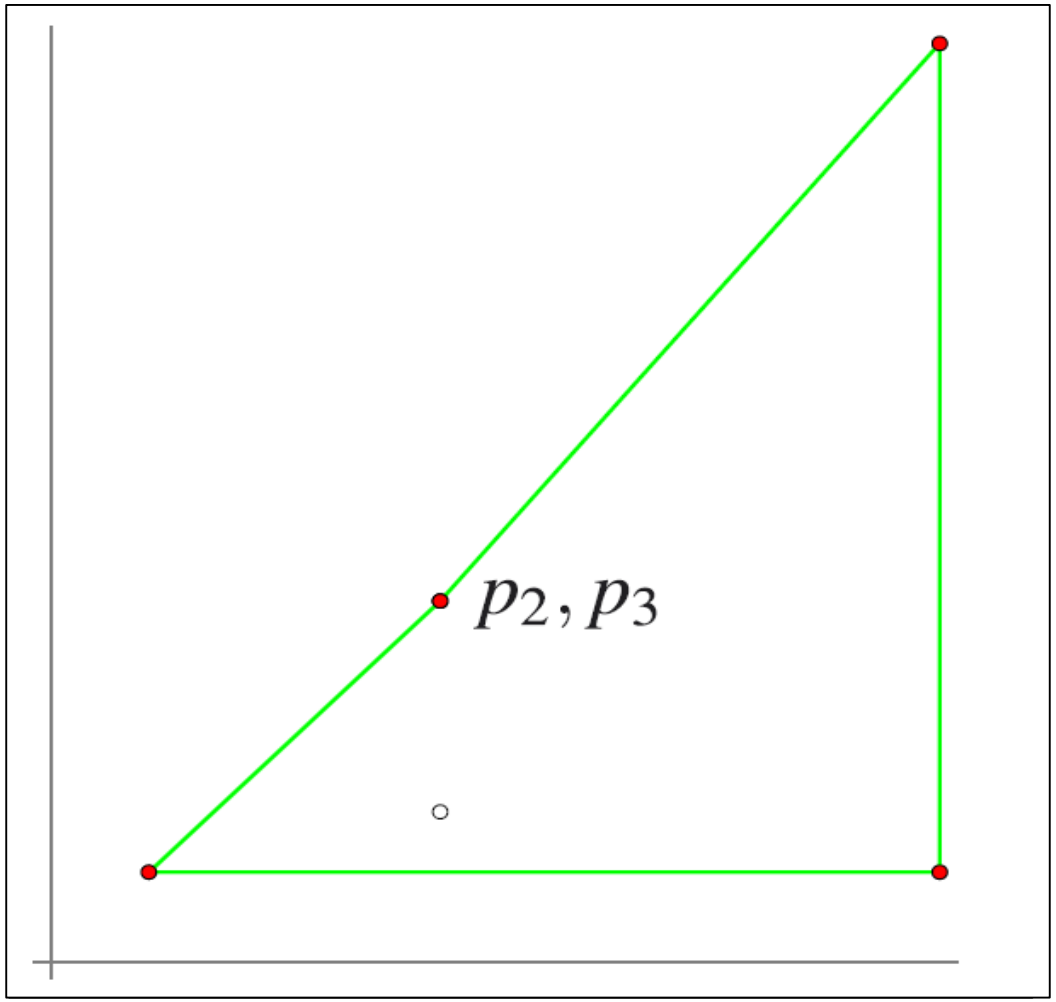
1. How does one determine whether there is an edge visible from r? We iterate over the edges in L, checking each edge using the orientation predicate. If no visible edge is found, we discard r. Otherwise, we take any one of the visible edges as the starting edge for the next substep.
2. How does one identify the sequence $((v_i, v_{i+1}), (v_{i+1}, v_{i+2}) \ldots, (v_{j-1}, v_j))$? Starting from a visible edge e, we move counter-clockwise along the boundary until a non-weakly-visible edge is encountered. Similarly, we move clockwise from e until a non-weakly-visible edge is encountered.
3. How to update the list L? We can delete the vertices in $(v_{i+1}, \ldots, v_{j-1})$ after all visible edges are found, as suggested in the above sketch ("the off-line strategy") or we can delete them concurrently with the search for weakly visible edges ("the on-line strategy"). With exact arithmetic, both strategies work equally well.

We give a detailed implementation in Appendix A; it was used for all experiments. Note that the algorithm (correctly) reports extreme points only. Points in the interior of boundary edges of the convex hull are not reported. Duplicate points are reported only once.

There are four logical ways to negate Properties A and B:

- Failure A_1: A point outside the current hull sees no edge of the current hull.
- Failure A_1: A point inside the current hull sees an edge of the current hull.
- Failure B_1: A point outside the current hull sees all edges of the convex hull.
- Failure B_2: A point outside the current hull sees a non-contiguous set of edges.

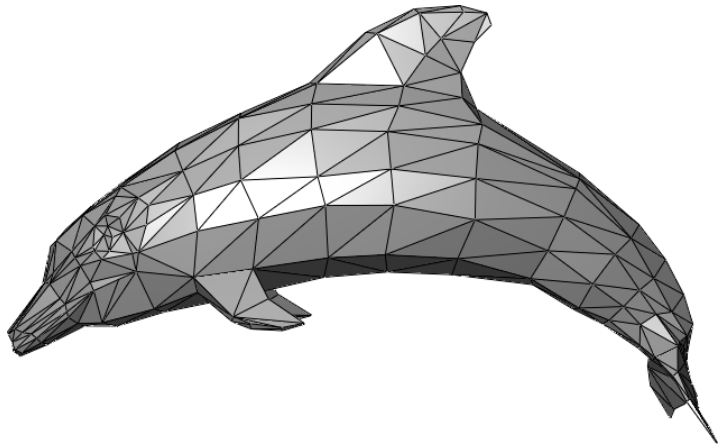
Failures A_1 and A_2 are equivalent to the negation of Property A. Similarly, Failures B_1 and B_2 are complete for Property B if we take A_1 into account. Are all these failures realizable? We now affirm this.



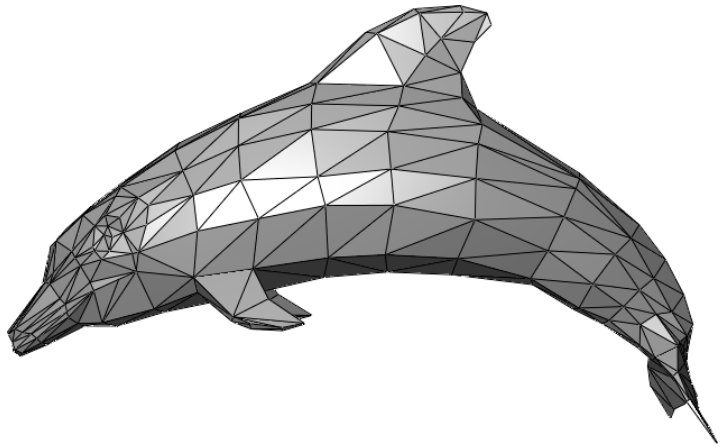
But, in real world…

Ok, let’s restart from the beginning

Let M be a triangle mesh…



Ok, let’s restart from the beginning



Let M be a triangle mesh…

Many different definitions in literature, not always compatible with each other ☹

Is a triangle mesh a simplicial complex ?

Is a triangle mesh a simplicial complex ?

- For most authors/papers, the answer is **YES**
  - Sometimes this is an implicit assumption

Is a triangle mesh a simplicial complex ?

- For most authors/papers, the answer is **YES**
  - Sometimes this is an implicit assumption
- More precisely, an Euclidean simplicial complex
  - Not to be confused with an *abstract simplicial complex*

Is a triangle mesh a simplicial complex ?

- For most authors/papers, the answer is **YES**
  - Sometimes this is an implicit assumption
- More precisely, an Euclidean simplicial complex
  - Not to be confused with an *abstract simplicial complex*
- Possible additional reqs:
  - 2-manifold, no boundary, genus 0, …

Is a triangle mesh a simplicial complex ?

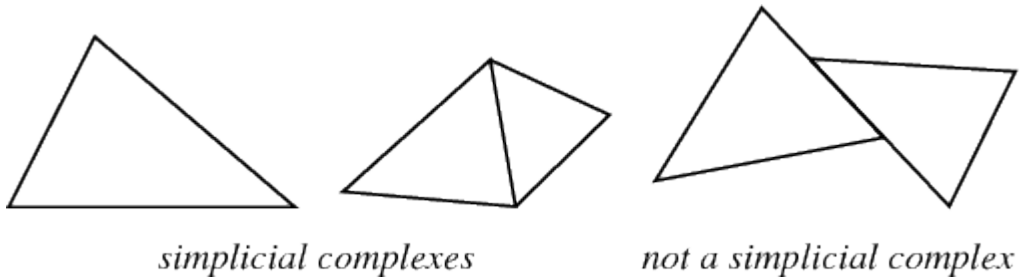
- For most authors/papers, the answer is **YES**
  - Sometimes this is an implicit assumption
- More precisely, an Euclidean simplicial complex
  - Not to be confused with an *abstract simplicial complex*
- Possible additional reqs:
  - 2-manifold, no boundary, genus 0, …

Definition
A simplicial complex K in R^n is a collection of simplices in R^n s.t.:
1. Every face of a simplex of K is in K
2. The intersection of any two simplices of K is a face of both

Is a triangle mesh a simplicial complex ?

- For most authors/papers, the answer is **YES**
  - Sometimes this is an implicit assumption
- More precisely, an Euclidean simplicial complex
  - Not to be confused with an *abstract simplicial complex*
- Possible additional reqs:
  - 2-manifold, no boundary, genus 0, …

Definition
A simplicial complex K in R^n is a collection of simplices in R^n s.t.:
1. Every face of a simplex of K is in K
2. The intersection of any two simplices of K is a face of both



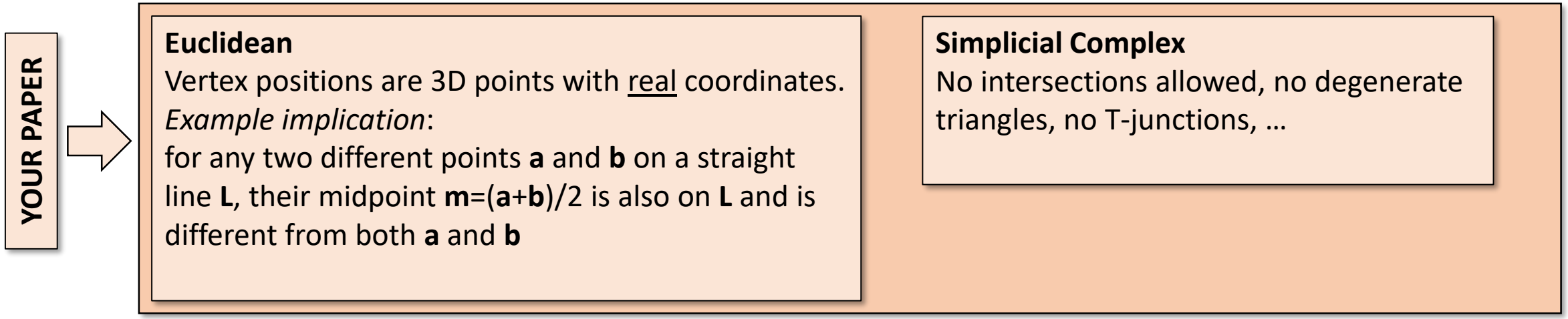
Euclidean Simplicial Complex

Euclidean Simplicial Complex

Euclidean
Vertex positions are 3D points with real coordinates.
Example implication:
for any two different points **a** and **b** on a straight line **L**, their midpoint **m**=(**a**+**b**)/2 is also on **L** and is different from both **a** and **b**

Simplicial Complex
No intersections allowed, no degenerate triangles, no T-junctions, …

Euclidean Simplicial Complex



Euclidean Simplicial Complex

YOUR PAPER

Euclidean
Vertex positions are 3D points with real coordinates.
Example implication:
for any two different points **a** and **b** on a straight line **L**, their midpoint **m**=(**a**+**b**)/2 is also on **L** and is different from both **a** and **b**

Simplicial Complex
No intersections allowed, no degenerate triangles, no T-junctions, …

Floating point arithmetic
Vertex coordinates can assume a finite number of different values
Example implication:
m is probably not on **L**
m might be coincident to either **a** or **b**

Euclidean Simplicial Complex

YOUR PAPER

Euclidean
Vertex positions are 3D points with real coordinates.
Example implication:
for any two different points **a** and **b** on a straight line **L**, their midpoint **m**=(**a**+**b**)/2 is also on **L** and is different from both **a** and **b**

Simplicial Complex
No intersections allowed, no degenerate triangles, no T-junctions, …

Floating point arithmetic
Vertex coordinates can assume a finite number of different values
Example implication:
m is probably not on **L**
m might be coincident to either **a** or **b**

YOUR CODE

Euclidean Simplicial Complex

YOUR PAPER

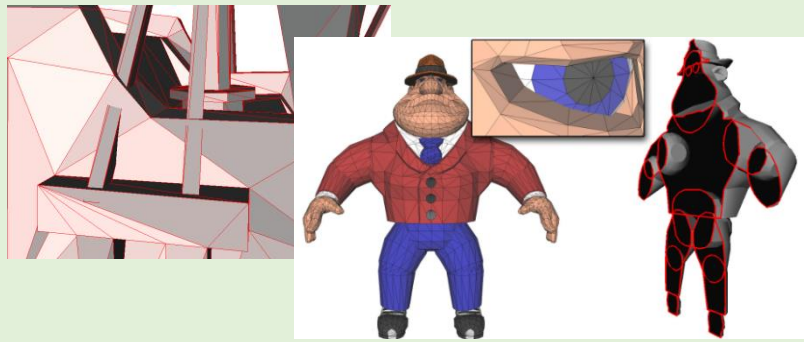
Euclidean
Vertex positions are 3D points with real coordinates.
Example implication:
for any two different points **a** and **b** on a straight line **L**, their midpoint **m**=(**a**+**b**)/2 is also on **L** and is different from both **a** and **b**

Simplicial Complex
No intersections allowed, no degenerate triangles, no T-junctions, …

Floating point arithmetic
Vertex coordinates can assume a finite number of different values
Example implication:
m is probably not on **L**
m might be coincident to either **a** or **b**

YOUR CODE

Real-world models



Euclidean Simplicial Complex

YOUR PAPER

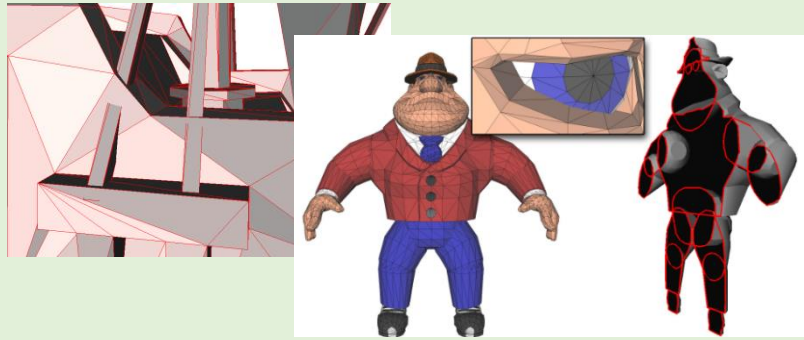
Euclidean
Vertex positions are 3D points with real coordinates.
Example implication:
for any two different points **a** and **b** on a straight line **L**, their midpoint **m**=(**a**+**b**)/2 is also on **L** and is different from both **a** and **b**

Simplicial Complex
No intersections allowed, no degenerate triangles, no T-junctions, …

Floating point arithmetic
Vertex coordinates can assume a finite number of different values
Example implication:
m is probably not on **L**
m might be coincident to either **a** or **b**

YOUR CODE

Real-world models



YOUR INPUT

Euclidean Simplicial Complex

YOUR PAPER

Euclidean
Vertex positions are 3D points with real coordinates.
Example implication:
for any two different points **a** and **b** on a straight line **L**, their midpoint **m**=(**a**+**b**)/2 is also on **L** and is different from both **a** and **b**

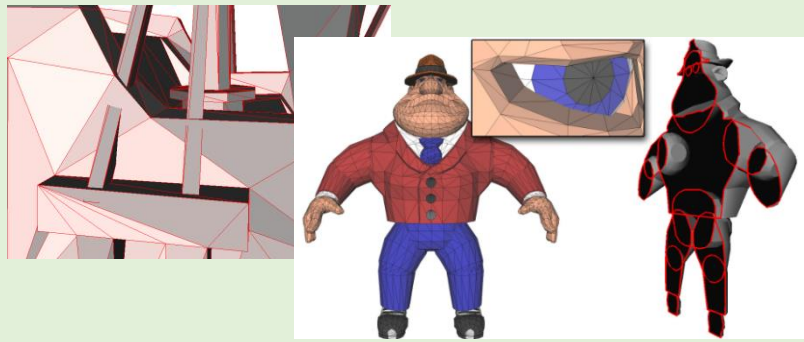
Simplicial Complex
No intersections allowed, no degenerate triangles, no T-junctions, …

Possible additional requirements
2-manifold, no boundary, genus 0, …

Floating point arithmetic
Vertex coordinates can assume a finite number of different values
Example implication:
m is probably not on **L**
m might be coincident to either **a** or **b**

YOUR CODE

Real-world models



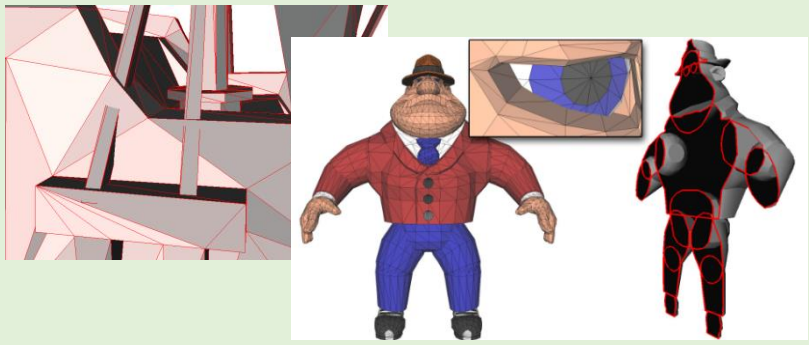
YOUR INPUT

Bridge the gap

Bad input -> Mesh checking/repairing

Weak code -> Robust programming

Real-world models



YOUR INPUT

Floating point arithmetic

Vertex coordinates can assume a finite number of different values

Example implication:

m is probably not on **L**

m might be coincident to either **a** or **b**

YOUR CODE

Mesh repairing

MOTIVATION

- *Real world* meshes often contain various defects, depending on their origin.
- But many applications assume *ideal* meshes free from defects or flaws.

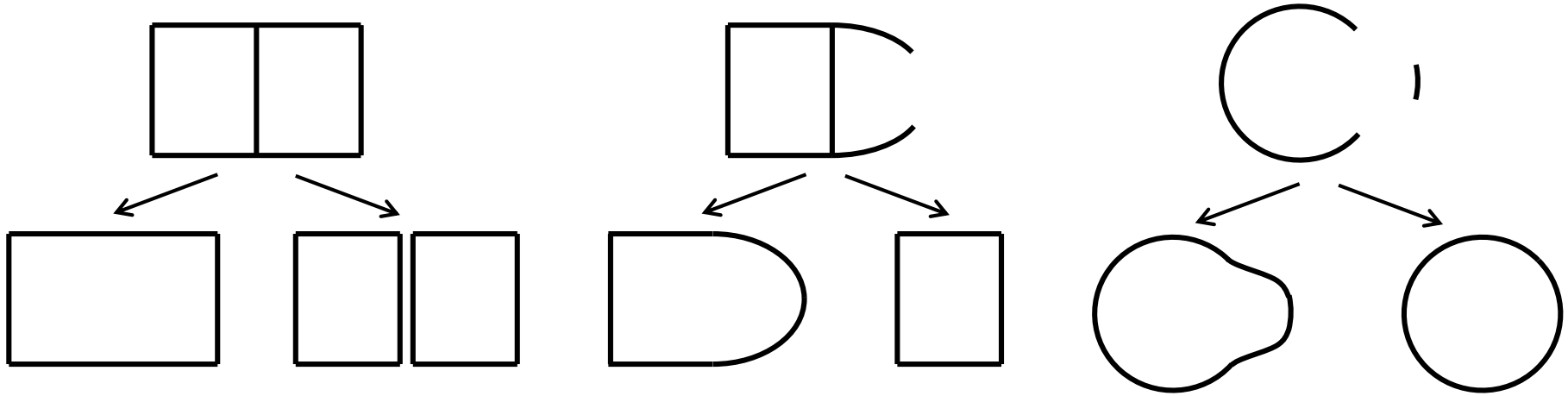
- *Mesh Repairing* adapts raw mesh models to specific application requirements.

MOTIVATION

- Complexity of the repair task is often underestimated by non-experts.
  - A large difference between „looks good“ and „is good“
  - Most repair algorithms focus on certain defect types and ignore or even introduce others.

Generic Mesh Repairing

- The *general* mesh repair problem is ill-posed
  - Inherent ambiguities (topologic & geometric)
  - Solution: application-specific context knowledge, heuristics, interactive user input ...



THE APPLICATION PERSPECTIVE

- Categorization of:
  - Defect types
  - Upstream applications
    - based on typical characteristics/defects of produced meshes.
  - Downstream applications
    - based on typical requirements on consumed meshes.
  - Repair approaches
    - along with specific requirements and guarantees

Marco Attene, Marcel Campen and Leif Kobbelt. **Polygon Mesh Repairing: an application perspective**
ACM Computing Surveys, 2013

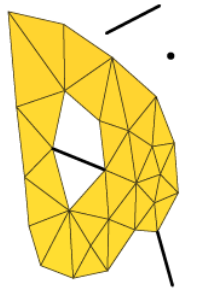
THE APPLICATION PERSPECTIVE

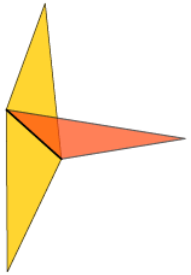
- Categorization of:
  - Defect types
  - Upstream applications
    - based on typical characteristics/defects of produced meshes.
  - Downstream applications
    - based on typical requirements on consumed meshes.
  - Repair approaches
    - along with specific requirements and guarantees

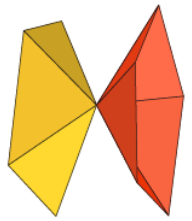
Marco Attene, Marcel Campen and Leif Kobbelt. **Polygon Mesh Repairing: an application perspective**
ACM Computing Surveys, 2013

DEFECT TYPES

- Local connectivity
  - Isolated vertices
    - „A vertex that is not incident to any edge“
  - Dangling edges
    - „Edges without any incident triangles“
  - Singular edges
    - „Edges with more than two incident triangles“
  - Singular vertices
    - „Vertices with a non-disc neighborhood“

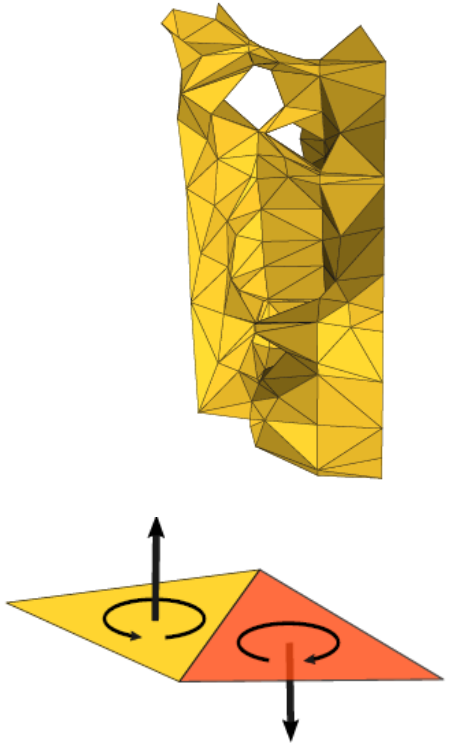






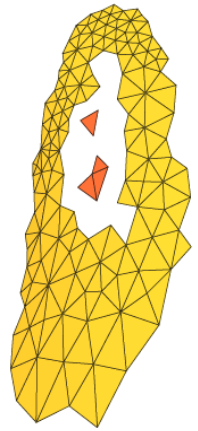
DEFECT TYPES

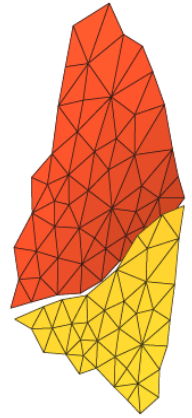
- Global topology
  - Topological noise
    - „Tiny spurious handles or tunnels“
    - „Tiny disconnected components“
    - „Unwanted cavities“
  - Orientation
    - „Incoherently oriented faces“



DEFECT TYPES

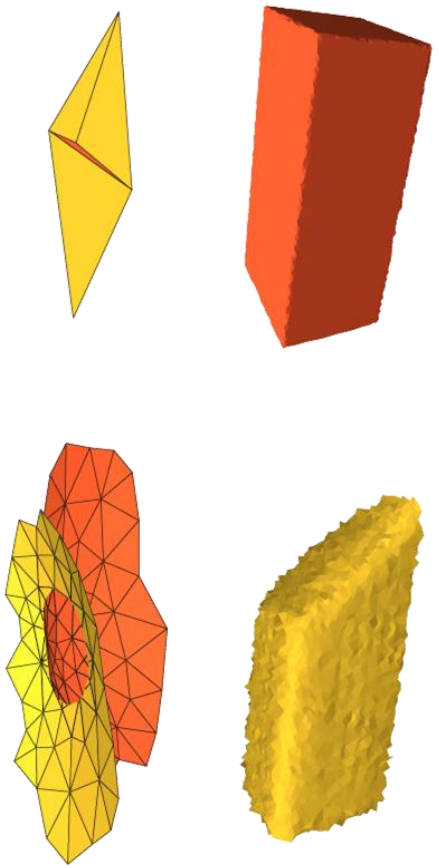
- Geometry
  - Holes
    - „Missing pieces within a surface“
      - e.g. due to occlusions during capturing
  - Gaps
    - „Missing pieces between surfaces“
      - e.g. due to inconsistent tessellation routines
  - Cracks / T-Junctions





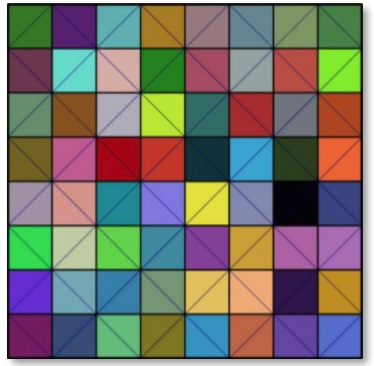
DEFECT TYPES

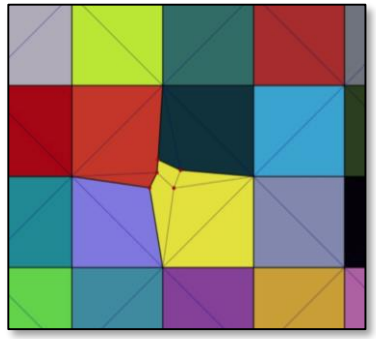
- Geometry
  - Degenerate elements
    - „Triangles with (near-)zero area“
  - Self-intersections
    - „Non-manifold geometric realization“
  - Sharp feature chamfering
    - „Aliasing artifacts due to sampling pattern“
  - Data noise
    - „Additive noise due to measurement imprecision“



GENERAL POSITION

- Assume that points are in general position…
- What if they are not?
  - Can this be considered a *defect*?
  - Geometric perturbation
  - Symbolic perturbation
- Simulation of Simplicity
  - Not really a repairing method (no change in geometry)
  - May require repairing aftwerwards (e.g. degenerate elems)





H. Edelsbrunner, E. P. Mücke. 1990. **Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms**. ACM TOG 9, 1, 66-104.

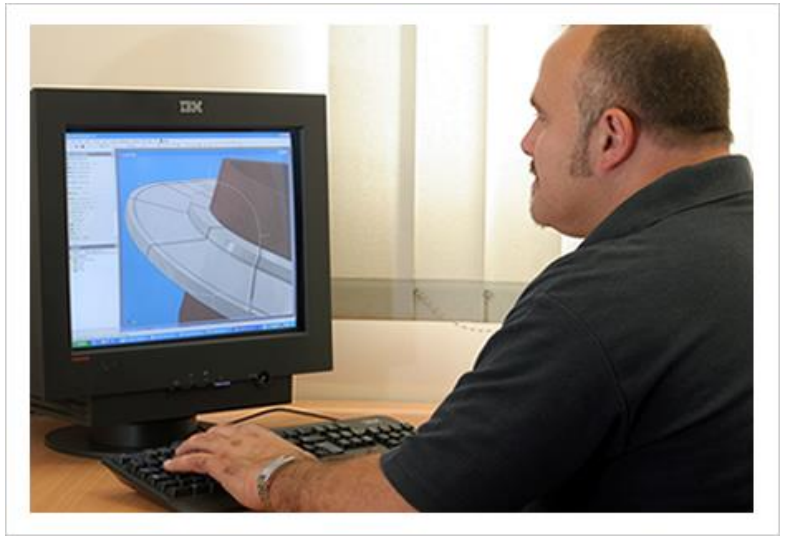
UPSTREAM APPLICATIONS

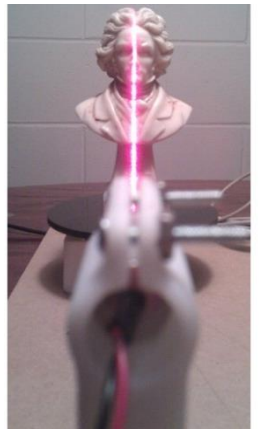
- Upstream applications (or *sources*) characterized by:
  - *Nature*
    - (physical) real-world data <-> (virtual) concepts
  - *Approach*
    - *… employed to convert data to polygon mesh*

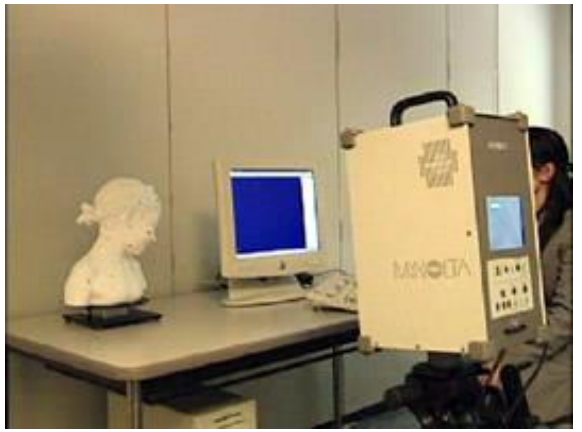
- Both aspects can be the source of defects and flaws.

UPSTREAM APPLICATIONS

- Nature
  - *Designed*
    - Basic concept is an abstraction
    - Problems due to:
      - Inaccuracies in the modeling process
      - Inconsistencies in the description/representation
  - *Digitized*
    - Measurement of real-world phenomenon
    - Problems due to:
      - Measurement inaccuracies
      - Measurement limitations







UPSTREAM APPLICATIONS

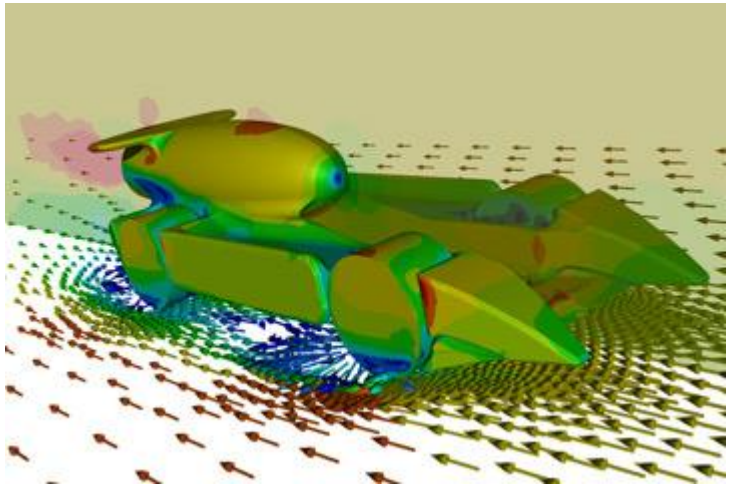
Nature	noise	holes	gaps	intersections	degeneracies	singularities	topolog. noise	aliasing
Digitized (physical)	**X**	**X**					**X**	**X**
Designed (virtual)			**X**	**X**	x	**X**		

UPSTREAM APPLICATIONS

Approach	noise	holes	gaps	intersections	degeneracies	singularities	topolog. noise	aliasing
Tessellation			**X**	**X**	x			
Depth image fusion				**X**	x	x		
Raster data contouring					x	**X**		
Implicit function contouring					x		x	**X**
Reconstruction from points		x	x				x	x
Height field triangulation								
Solid model boundary extract.						**X**		

DOWNSTREAM APPLICATIONS

- We consider prototypical requirements of a sample of the wide application spectrum
  - *Visualization*
  - *Modeling*
  - *Rapid Prototyping*
  - *Processing*
  - *Simulation*



DOWNSTREAM APPLICATIONS

Application Group	noise	holes	gaps	intersections	degeneracies	singularities	topolog. noise	aliasing
Visualization	x	**X**	x					x
Modeling		**X**	**X**		**X**	x	x	
Rapid Prototyping		**X**	**X**	**X**		**X**		
Processing	**X**	**X**	**X**	x	**X**	**X**	x	x
Simulation	**X**	**X**	**X**	**X**	**X**	**X**	**X**	x

REPAIR APPROACHES

- We distinguish between two types:
  - Local:
    - Handling defects individually by local modifications.
    - Low invasiveness, but only few guarantees.
  - Global:
    - Typically based on a complete remeshing.
    - High robustness, but often loss of detail.
    - More plausible ambiguity resolution possible.

STATE OF THE ART

- Algorithms exist to fix any individual defect discussed so far
  - Ref to Attene, Campen, Kobbelt (2013) for comprehensive description
- A specific defect is rarely the only defect
- While fixing one, you may introduce another
- Need to carefully study repairing workflows and/or integrated algorithms

SOME EXAMPLES

- MeshFIX
  - extremely fast, lightweight and robust
  - raw digitized meshes
  - may introduce macroscopic distortions
- As-exact-as-possible STL fix
  - Extremely precise and reasonably fast
  - designed meshes
  - only outer geometry
- Polymender
  - Extremely fast, lightweight and robust
  - Any mesh
  - Distortion everywhere, large triangle count
- TetWild
  - extremely robust
  - any mesh
  - Distortion everywhere, large triangle count, slow

MeshFIX - Raw digitized meshes

- We can assume that:
  - Samples are rather uniformly spaced
  - Model is densely sampled (opposed to sparse tessellated NURBS)
- Objective:
  - If an area is free of errors, keep it as it is
- What is the typical input?
  - An indexed face set, possibly non manifold, self-intersecting, with degenerate faces, holes, …

MeshFIX - Raw digitized meshes

- We can assume that:
  - Samples are rather uniformly spaced
  - Model is densely sampled (opposed to sparse tessellated NURBS)
- Objective:
  - If an area is free of errors, keep it as it is
- What is the typical input?
  - An indexed face set, possibly non manifold, self-intersecting, with degenerate faces, holes, …

MeshFIX - repairing pipeline

- Sequence of local approaches
- Creates a valid watertight polyhedral surface
- Works in two successive phases:
  - Topology reconstruction
  - Geometry correction

Topology reconstruction

INPUT: ndexed face set (e.g. an OFF file)

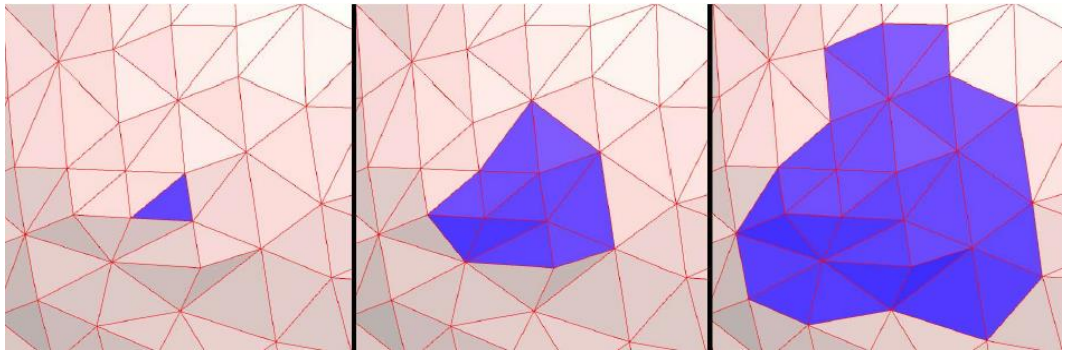
1) Convert to an abstract simplicial complex
2) Convert the complex to a combinatorial manifold
3) Orient consistently (and possibly cut unorientable manifolds)
4) Remove spurious components
5) Fill holes

OUTPUT: a single combinatorial oriented manifold with no boundary

Simplicial Neighborhoods

- For the “geometry correction” phase, we make use of the notion of simplicial neighborhood

1) The simplicial neighborhood N(t) is the set of all the simplexes which share at least a vertex with the triangle ‘t’
2) The i’th order simplicial neighborhood $N_i(t)$ is defined as N(N(…N(N(t))…)), with ‘i’ nested levels

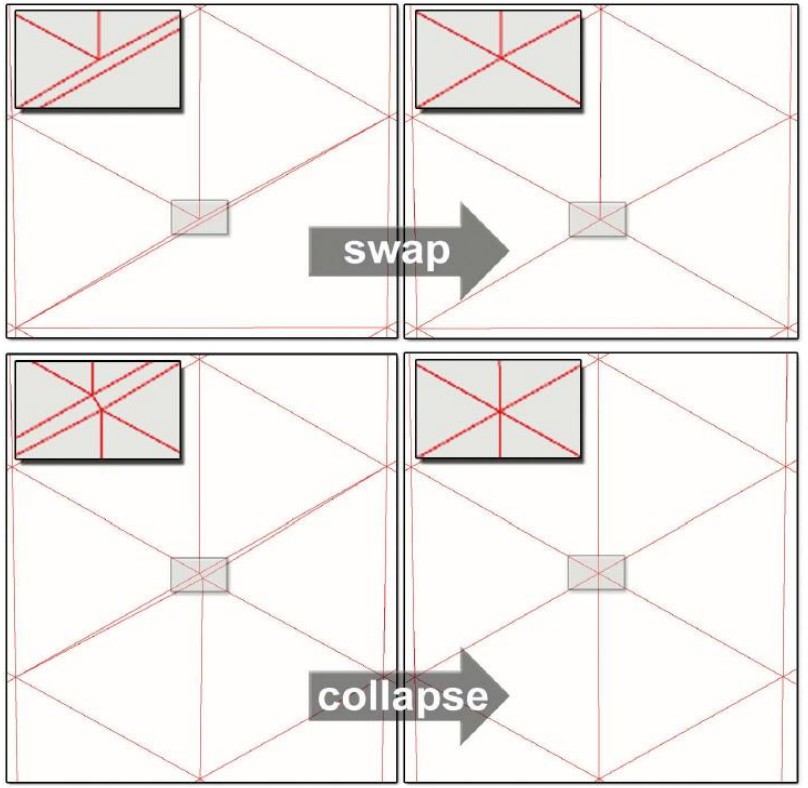


Geometry correction: step 1

- Remove (nearly) degenerate triangles

Require: A combinatorial manifold $\mathcal{M}$ and an integer threshold $max\_iterations$
Ensure: A combinatorial manifold $\mathcal{M}'$ and a status notice (success/failure)

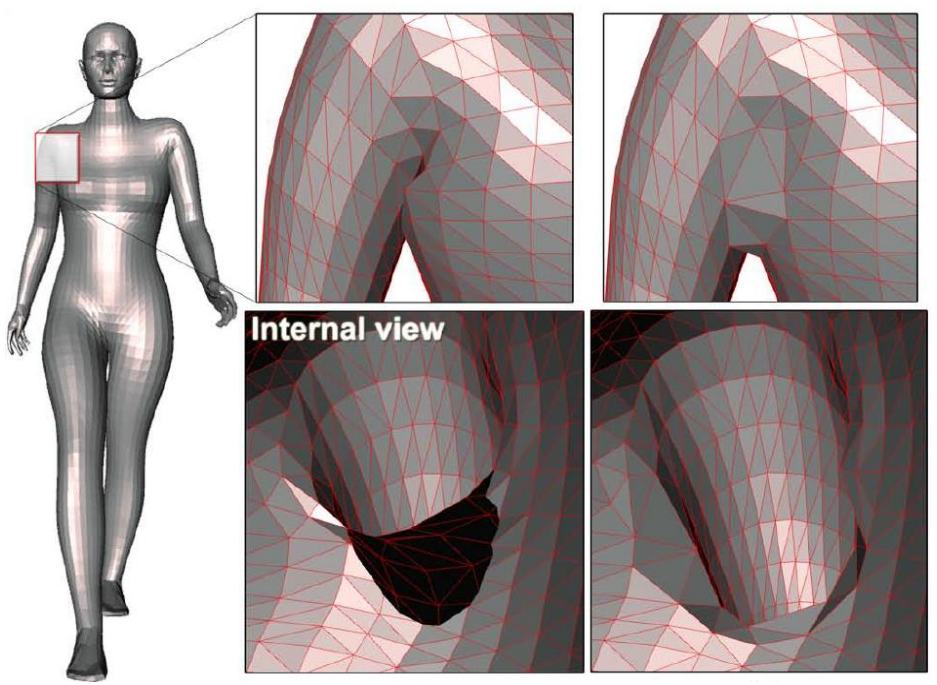
1: $\mathcal{M}' := \mathcal{M}$
2: Let S be the set of all the triangles of $\mathcal{M}'$
3: **for** $k = 1$ to $max\_iterations$ **do**
4: Run the swap/collapse algorithm within S
5: Let T be the set of degeneracies in S untreatable due to topological constraints
6: **if** $T = \emptyset$ **then**
7: terminate with success /* $\mathcal{M}'$ *is degeneracy free* */
8: **end if**
9: Let R be the union of the $k^{th} - order$ simplicial neighborhoods of the t_is $\in T$
10: Remove R from $\mathcal{M}'$
11: Remove possible disconnected components from $\mathcal{M}'$
12: Patch the remaining gaps with a new set P of triangles
13: $S := P$
14: **end for**



Geometry correction: step 2

- Remove intersecting triangles

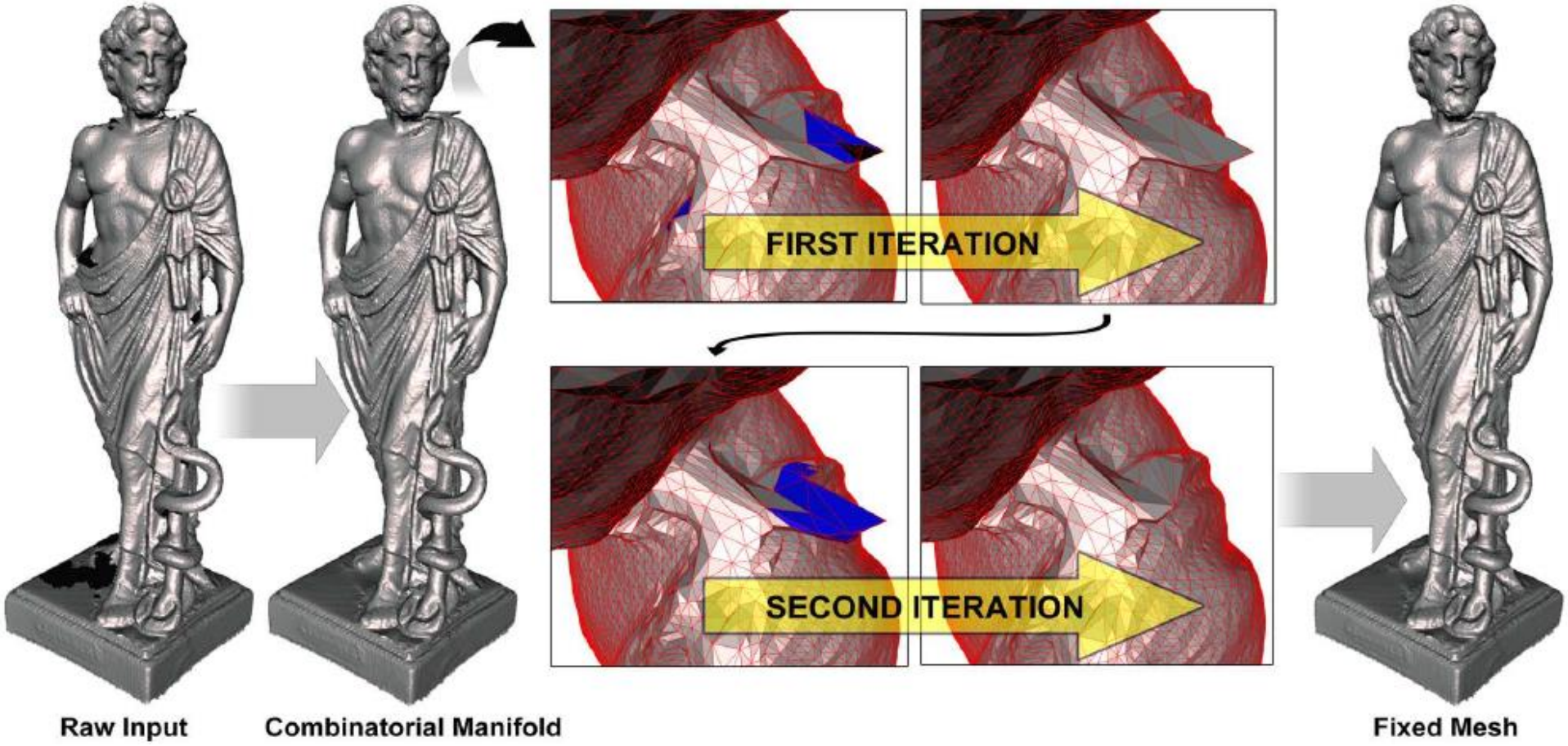
1: $\mathcal{M}' := \mathcal{M}$
2: Let S be the set of all the triangles of $\mathcal{M}'$
3: Let G be a uniform 100^3 voxel grid tightly enclosing $\mathcal{M}'$
4: **for** $k = 0$ to $max\_iterations$ **do**
5: Let H be the set of voxels intersecting at least a triangle of S
6: Check for triangle-triangle intersections within each voxel of H
7: Let T be the set of intersecting triangles detected above
8: **if** $T = \emptyset$ **then**
9: terminate with success /* $\mathcal{M}'$ *is not self-intersecting* */
10: **end if**
11: Let R be the union of the $k^{th} - order$ simplicial neighborhoods of all $t \in T$
12: Remove R from $\mathcal{M}'$
13: Remove possible disconnected components from $\mathcal{M}'$
14: Patch the remaining gaps with a new set P of triangles
15: $S := P$
16: **end for**



Geometry correction: iteration

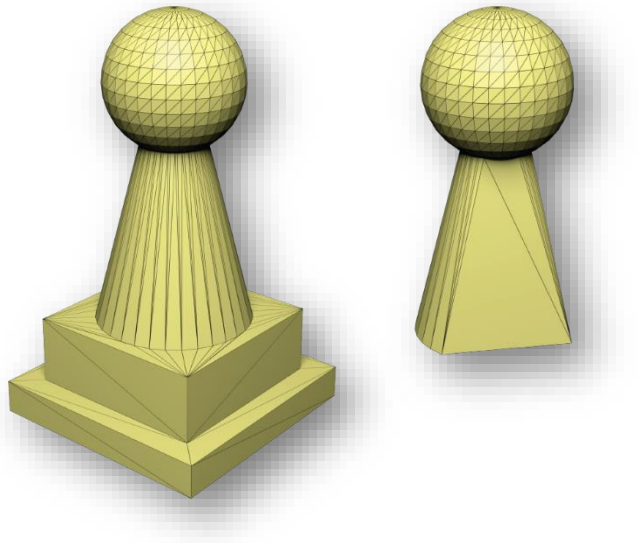
- While patching holes to remove self-intersections, new degenerate or nearly degenerate triangles may appear, and/or new intersections may be created
- So, after step 2 we check for degeneracies and intersections again and, if any, we repeat steps 1 and 2, until no more defects are left
- This is not guaranteed to converge, but it normally does in practical cases

Example


FIRST ITERATION
SECOND ITERATION
Raw Input
Combinatorial Manifold
Fixed Mesh

MeshFIX – Concluding remarks

- MeshFIX
  - extremely fast, lightweight and robust
  - raw digitized meshes
  - may introduce macroscopic distortions if used on inappropriate models



Marco Attene. **A lightweight approach to repair digitized polygon meshes.**
The Visual Computer, 2010

As-exact-as-possible repair for 3D printing

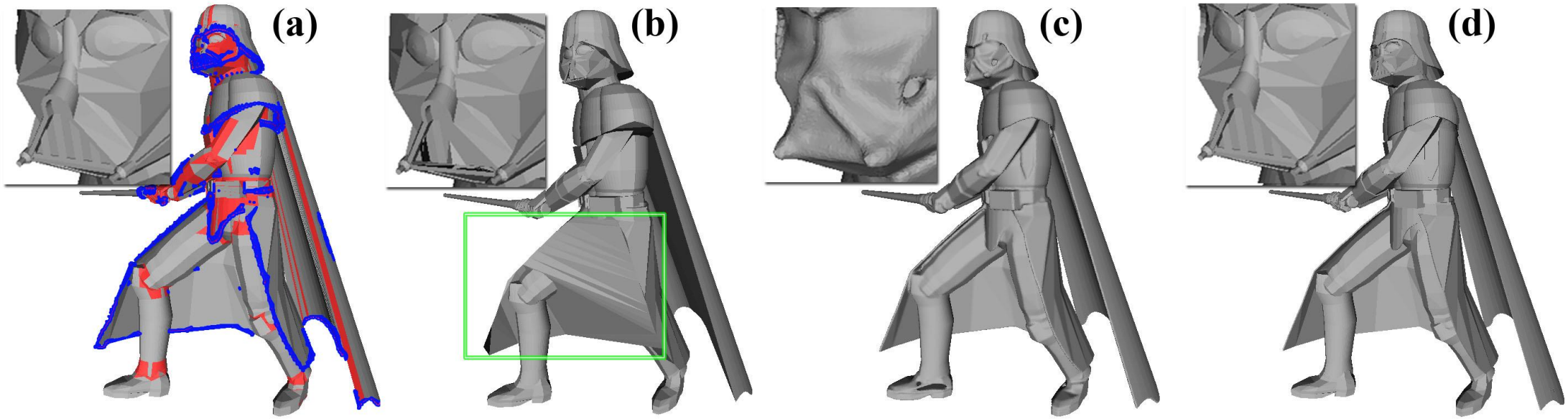
- Tessellated CAD models
- Typical input for slicers is STL
- The class of STL files is larger than the class of printable models
  - There exist well-formed STLs that cannot be printed
- What are the conditions that make a model «printable»?
- How can we repair an unprintable STL to make it printable?

As-exact-as-possible repair for 3D printing

- Tessellated CAD models
- Typical input for slicers is STL
- The class of STL files is larger than the class of printable models
  - There exist well-formed STLs that cannot be printed
- What are the conditions that make a model «printable»?
- How can we repair an unprintable STL to make it printable?

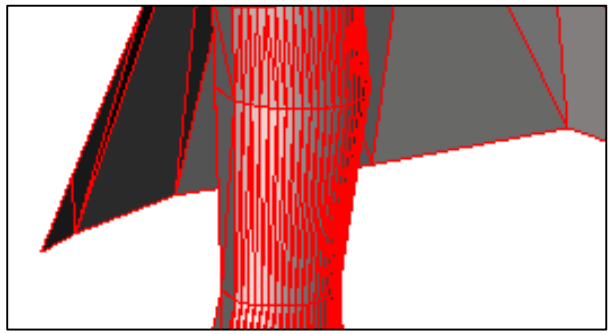
As-exact-as-possible

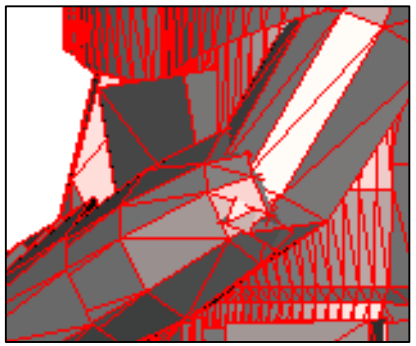
- What you see is what you get
  - Assume that the artist wants the print to appear exactly as the rendered model



As-exact-as-possible ≠ exact

- Artists may approximate thin parts by zero-thickness sheets of triangles
- But printers cannot extrude zero-thickness material!
- If extrusion diameter is ε, we may turn our sheets to ε-thick solids





What is «printable»?

Definition. Printable STL

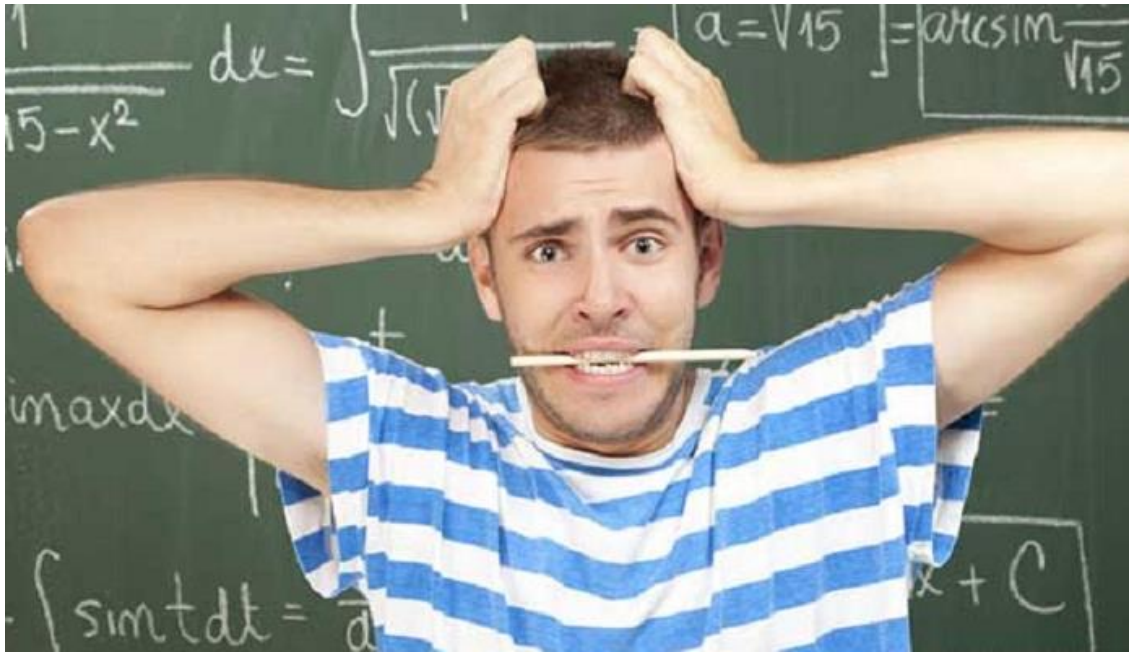
An STL model T is printable if there exists a T-induced mesh whose realization is a closed and manifold-connected polyhedron that coincides with its outer hull.

Marco Attene, (2018) "As-exact-as-possible repair of unprintable STL files", Rapid Prototyping Journal, Vol. 24 Issue: 5, pp.855-864, https://doi.org/10.1108/RPJ-11-2016-0185

What is «printable»?

Definition. Printable STL

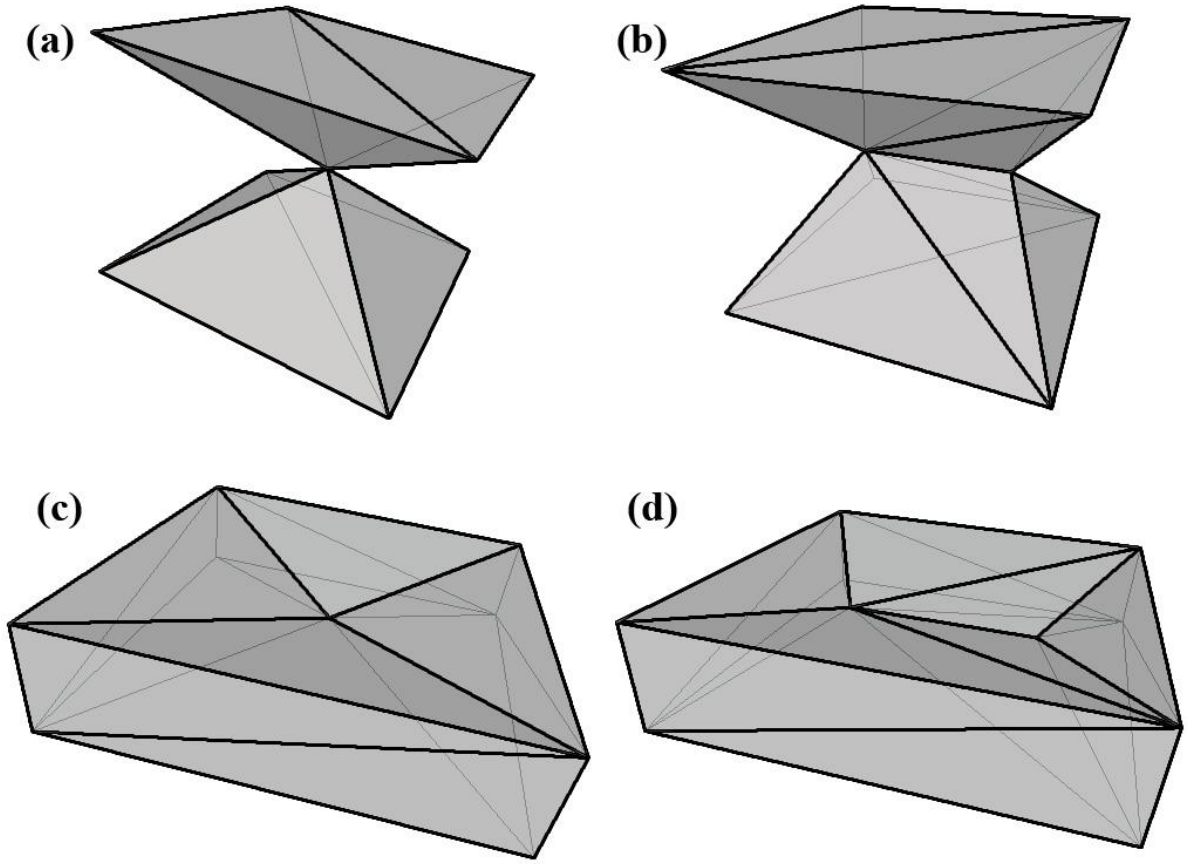
An STL model T is printable if there exists a T-induced mesh whose realization is a closed and manifold-connected polyhedron that coincides with its outer hull.



Marco Attene, (2018) "As-exact-as-possible repair of unprintable STL files", Rapid Prototyping Journal, Vol. 24 Issue: 5, pp.855-864, https://doi.org/10.1108/RPJ-11-2016-0185

What is «printable»?

- A «sufficiently connected» solid
  - No boundary
  - No intersections
  - No «occluded» parts
    - Reachable from infinity
  - Manifold-connected
    - I.e. Still connected after removal of singular simplexes



Overview of the repairing process

Input STL: consider vertex position and triangles as the only reliable information

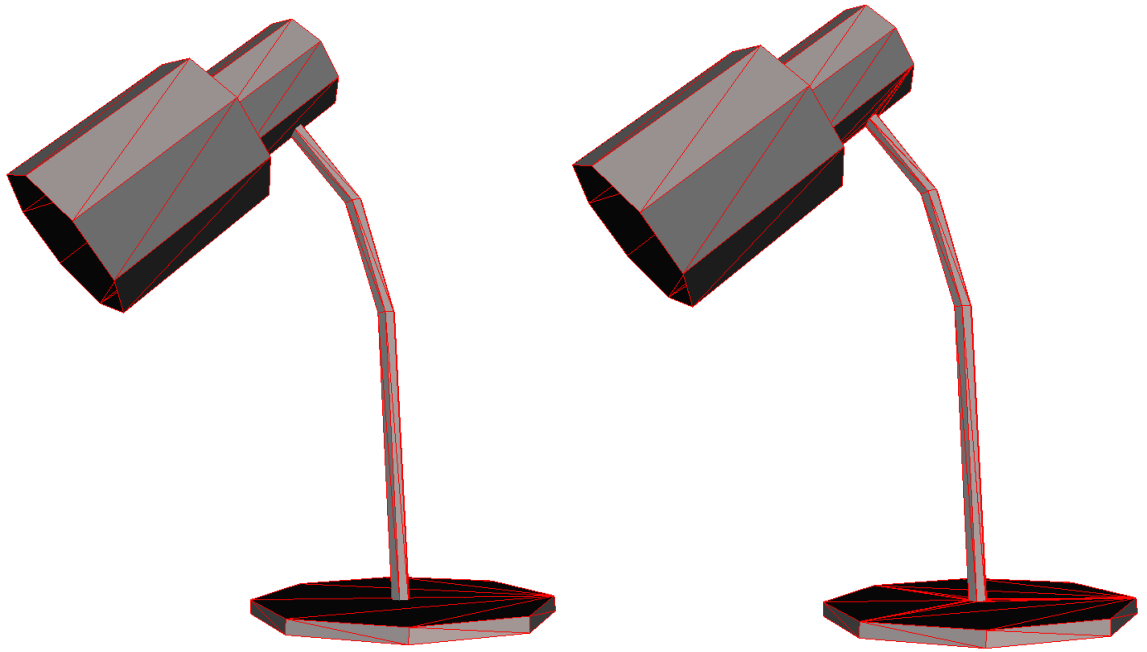
- Ignore normals/orientation
  - zero-thickness sheets <-> orientation?

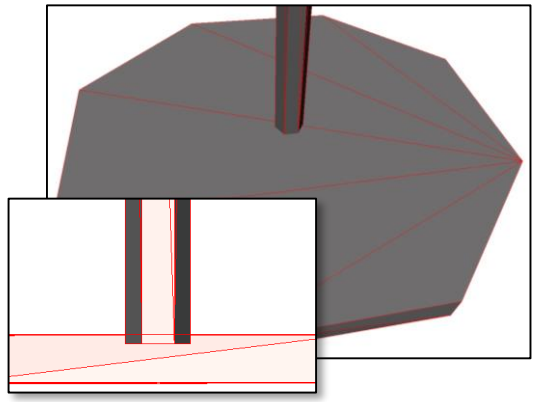
Convert STL to printable STL

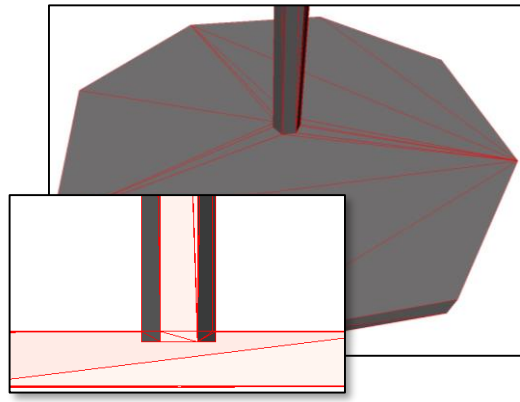
1. Convert to Euclidean Simplicial Complex
   1. We shall omit «Euclidean» from now on
2. Simplicial complex to outer hull
3. Outer hull to printable solid (i.e. thicken possible sheets)

STL to Simplicial Complex

1. Unify coincident vertices
2. Remove zero area triangles
3. Remove duplicated triangles
4. Resolve intersections

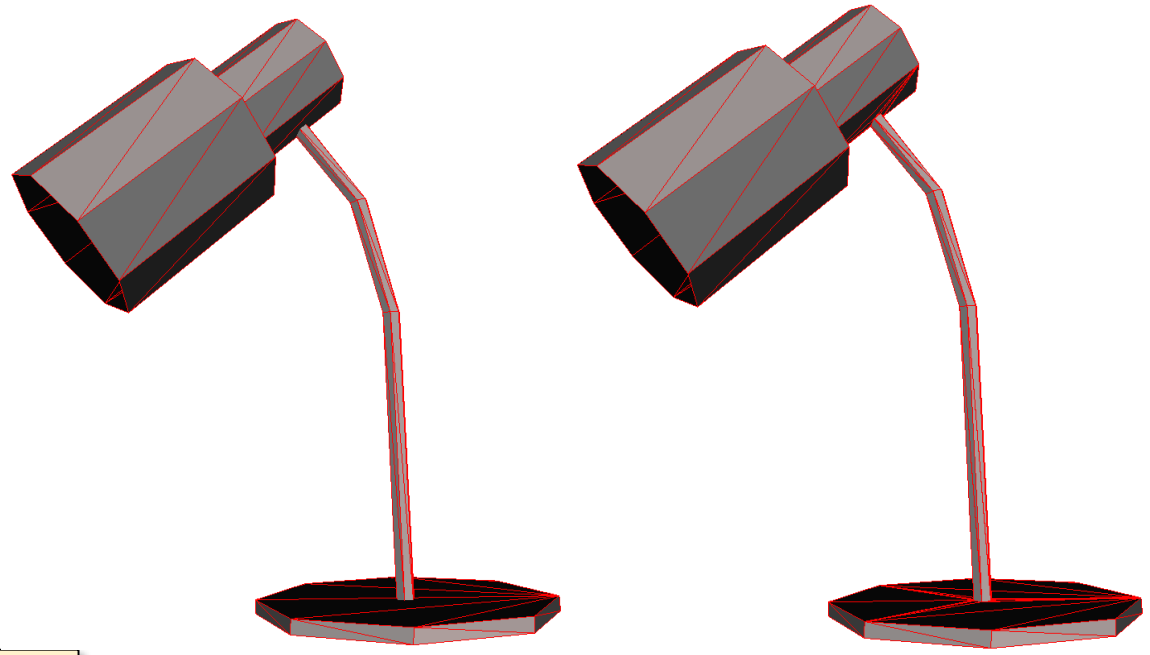




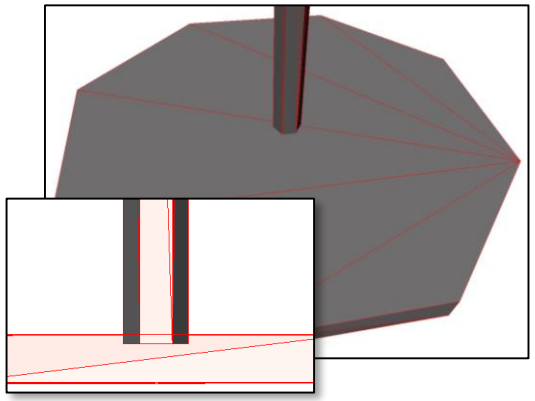


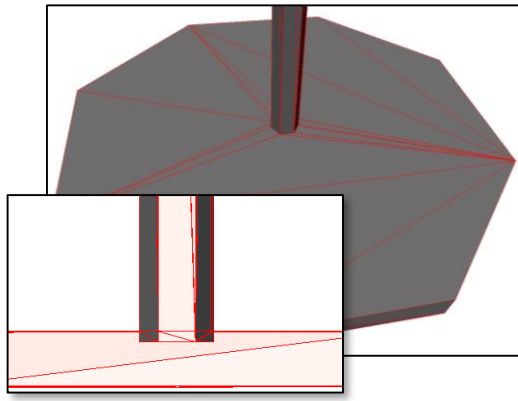
STL to Simplicial Complex

1. Unify coincident vertices
2. Remove zero area triangles
3. Remove duplicated triangles
4. Resolve intersections

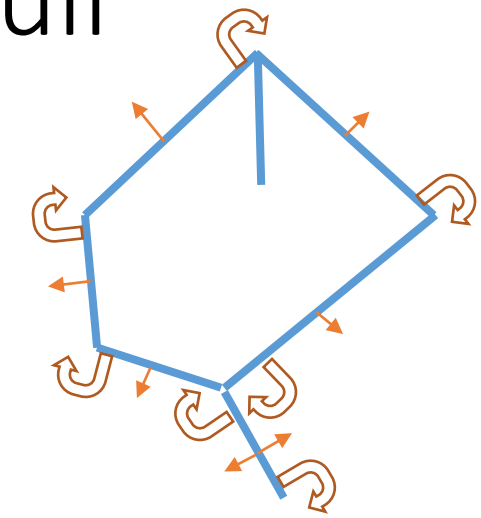


Definition
A simplicial complex K in R^n is a collection of simplices in R^n s.t.:
1. Every face of a simplex of K is in K
2. The intersection of any two simplices of K is a face of both

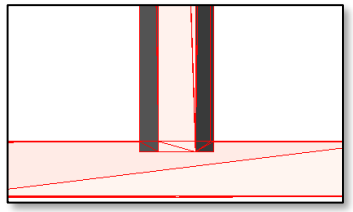


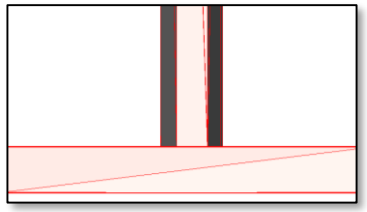


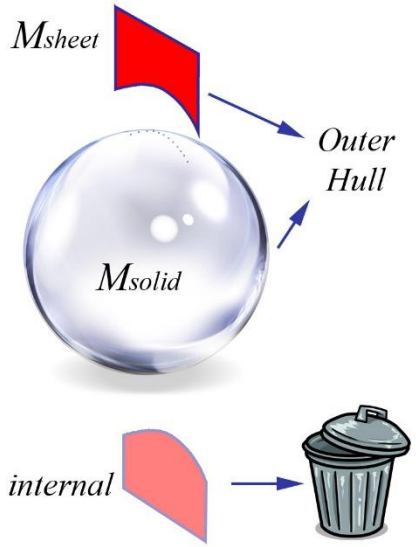
Simplicial Complex to outer hull



1. Each triangle has two sides
2. Select a seed triangle
    1. Tag its «outer» side
3. Propagate the tag to adjacent triangles
    1. Across edges only
    2. Propagate on one triangle at each step
        1. Across manifold edges: easy
        2. Across boundary edges: double orientation
        3. Across singular edges: select «outmost» triangles

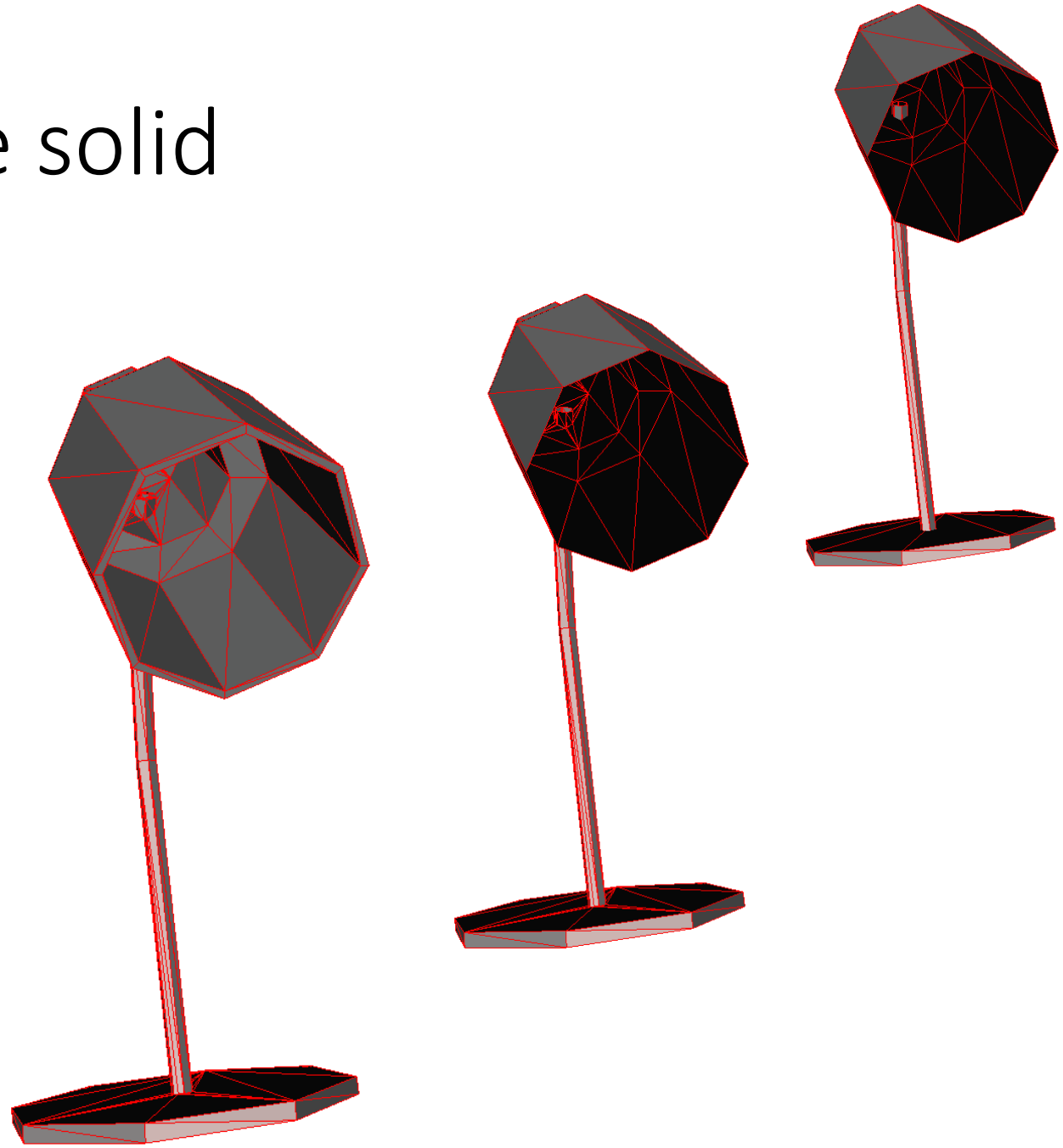






Outer hull to printable solid

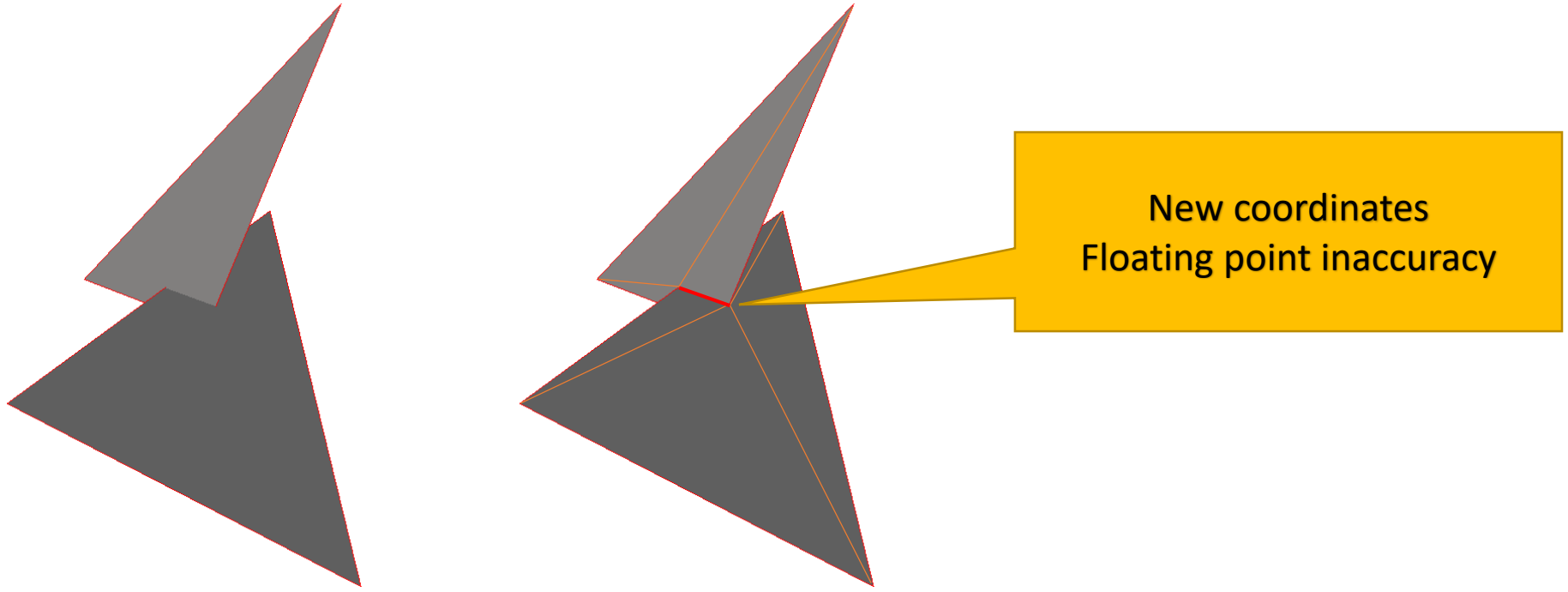
1. If there are no sheets
    1. TERMINATE
2. Else
    1. thicken the sheets
    2. Resolve possible intersections once again
    3. Track the outer hull once again



Devil is in the details…

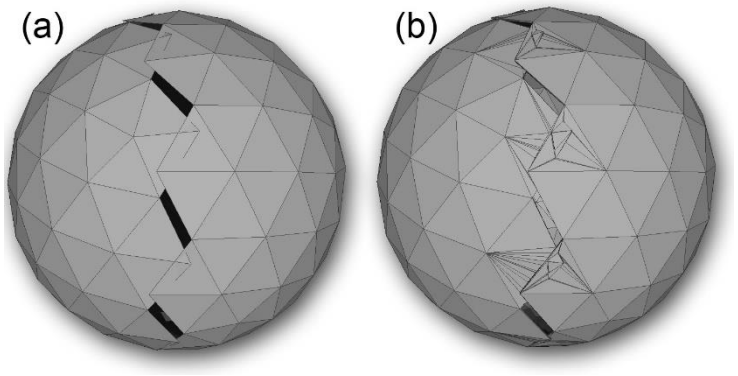
Resolving intersections

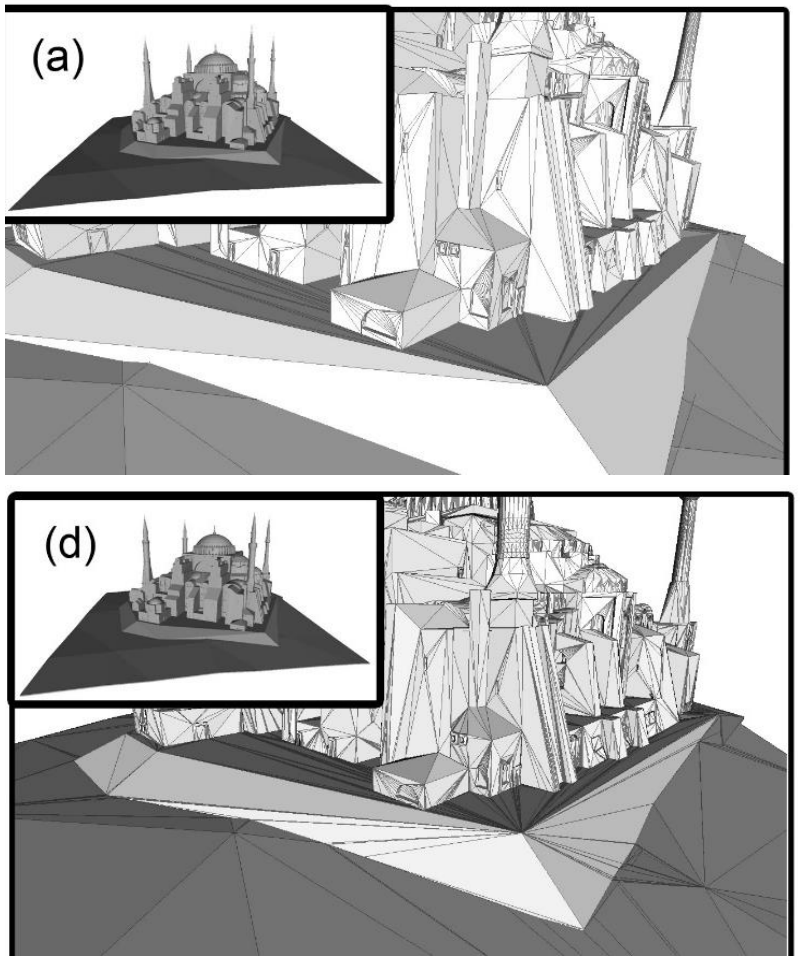
- If two simplexes intersect, create a new simplex representing the intersection and split



As-exact-as-possible STL fix – Concluding remarks

- As-exact-as-possible STL fix
  - Extremely precise and reasonably fast
  - designed meshes
  - only outer geometry
  - Surface holes are not patched

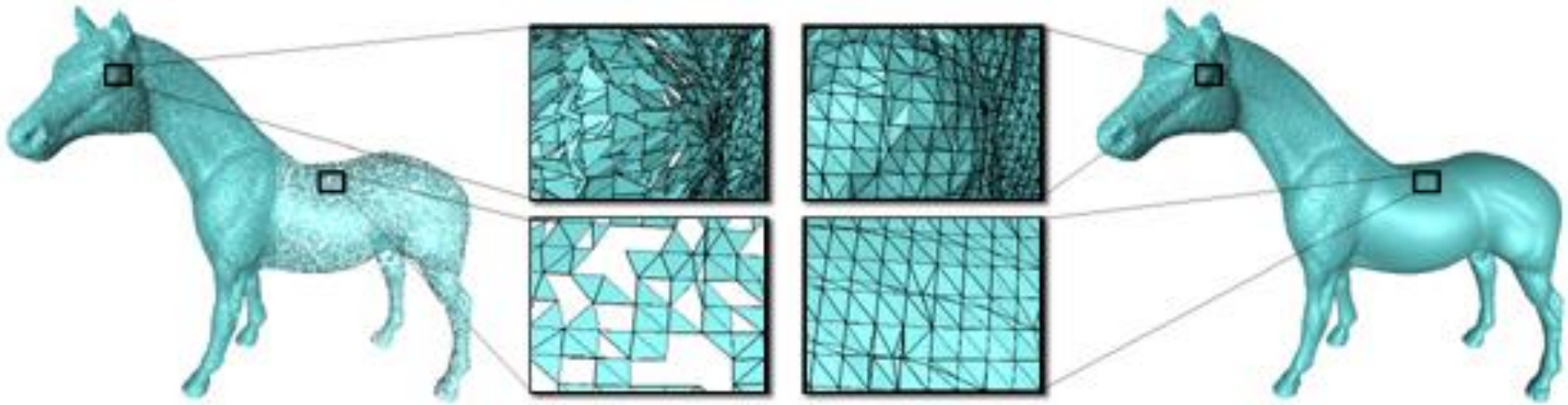




Marco Attene, (2018) "As-exact-as-possible repair of unprintable STL files", Rapid Prototyping Journal, Vol. 24 Issue: 5, pp.855-864, https://doi.org/10.1108/RPJ-11-2016-0185

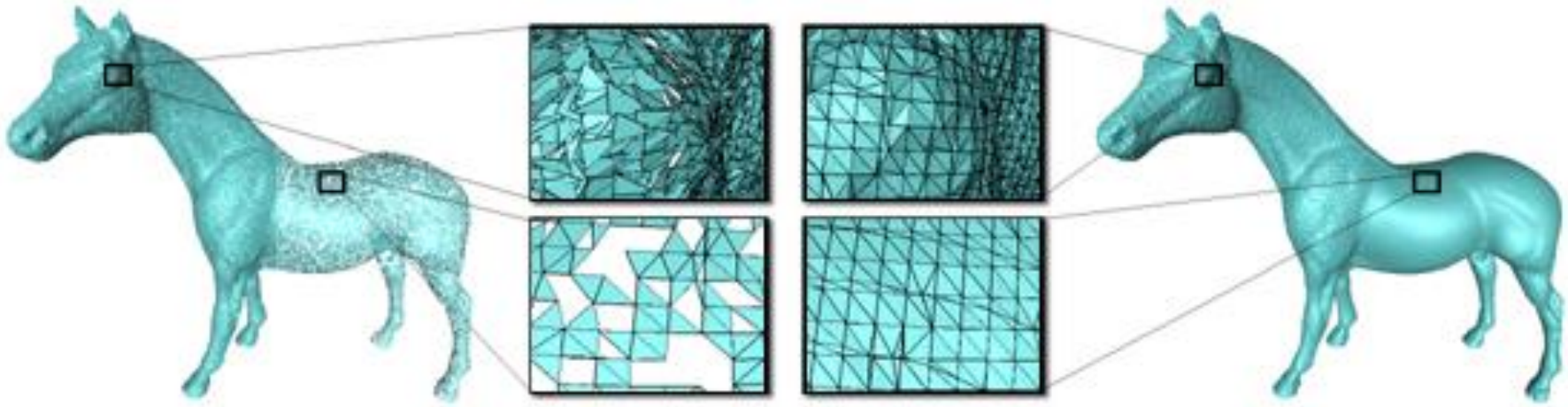
Polymender

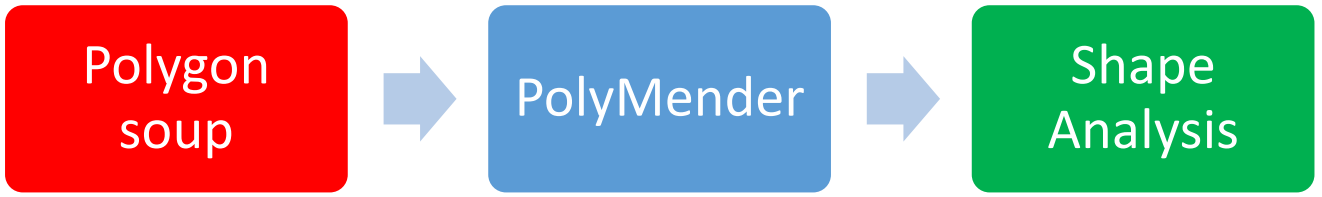
- Generates a closed surface by computing and contouring an intermediate volumetric grid denoting the inside/outside space of the input model



Polymender

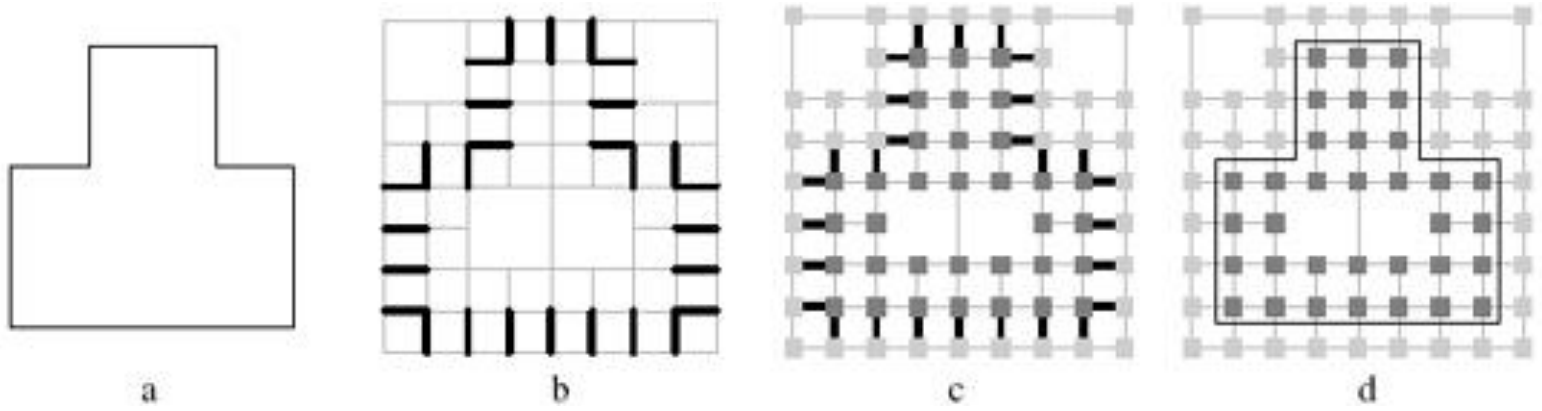
- Generates a closed surface by computing and contouring an intermediate volumetric grid denoting the inside/outside space of the input model





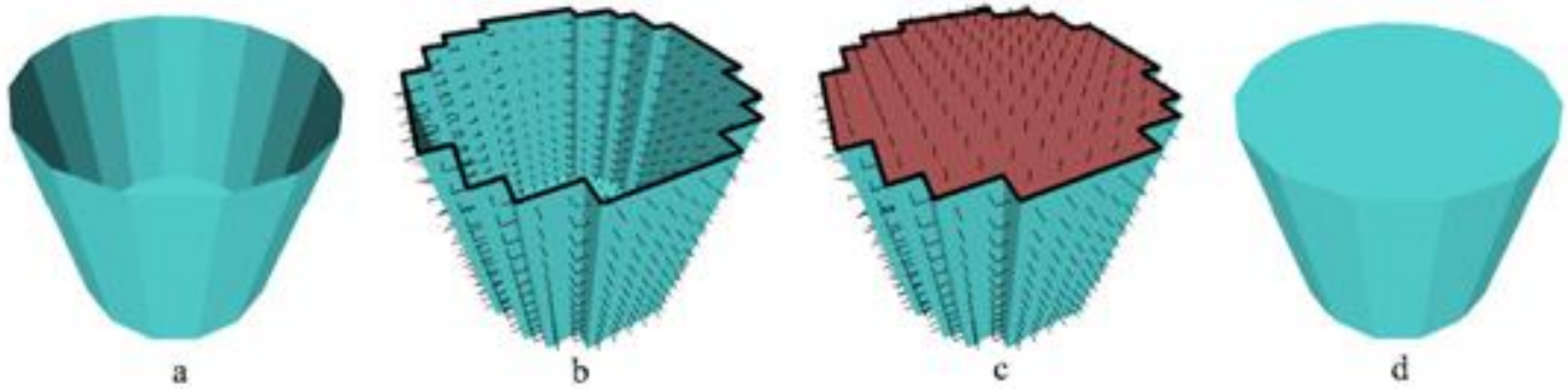
Polymender - overview

a) Input
b) Scan-conversion:
   - rasterize model, and mark intersection edges.
c) Sign generation:
   - so that each cell edge intersecting the model should exhibit a sign change
d) Surface reconstruction



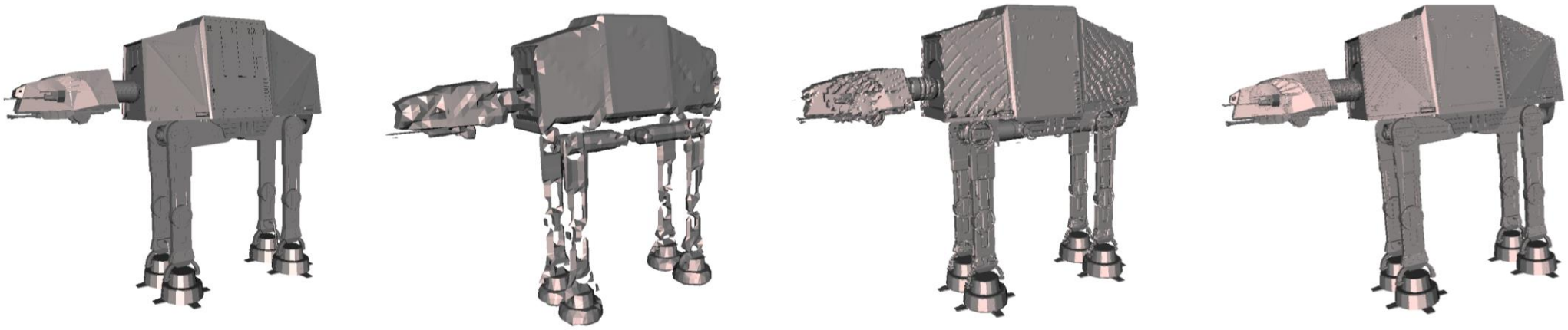
Polymender – sign generation

- Each edge in dual surface (b, quads <-> intersection edges) must have even number of incident quads
- Add/remove intersection edges in primal grid so that (1) holds
  - Divide-and-conquer patching of odd-valence dual edges (c)



Polymender – Concluding remarks

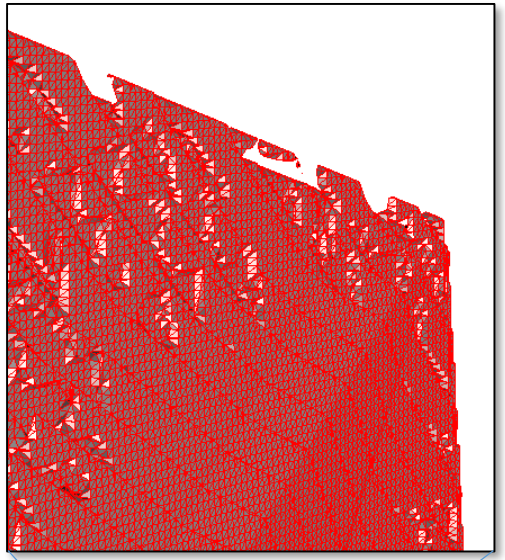
- Polymender
  - Extremely fast, lightweight and robust
  - Any mesh
  - Distortion everywhere
    - Inaccuracy vs. triangle count



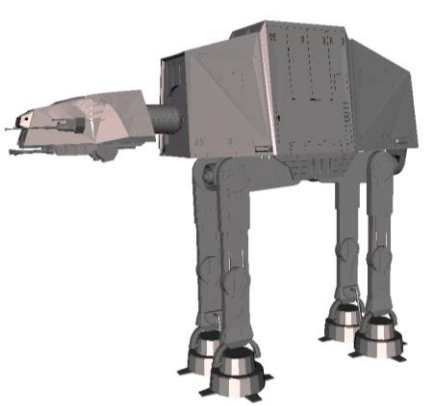
Input (65K faces) 12K 300K 6.8M

Tao Ju, (2004) “Robust Repair of polygonal models", ACM TOG Vol. 23 (SIGGRAPH 2004), pp. 888-895.

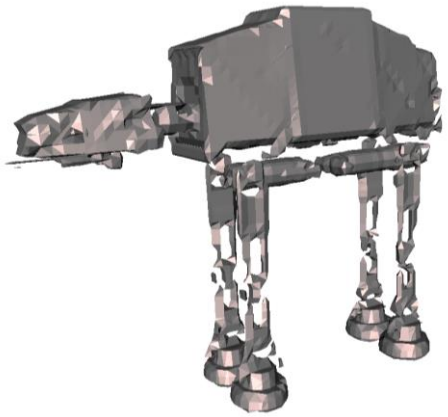
Polymender – Concluding remarks



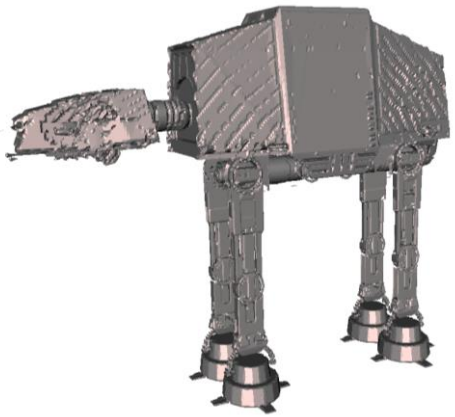
- Polymender
  - Extremely fast, lightweight and robust
  - Any mesh
  - Distortion everywhere
    - Inaccuracy vs. triangle count



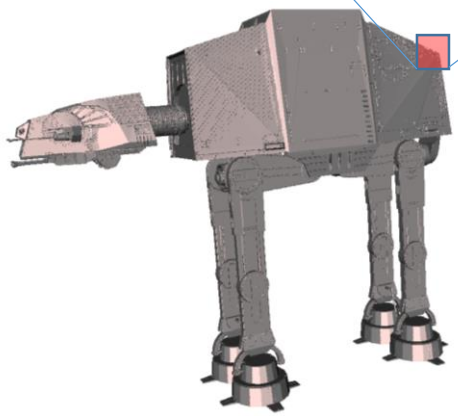
Input (65K faces)



12K



300K

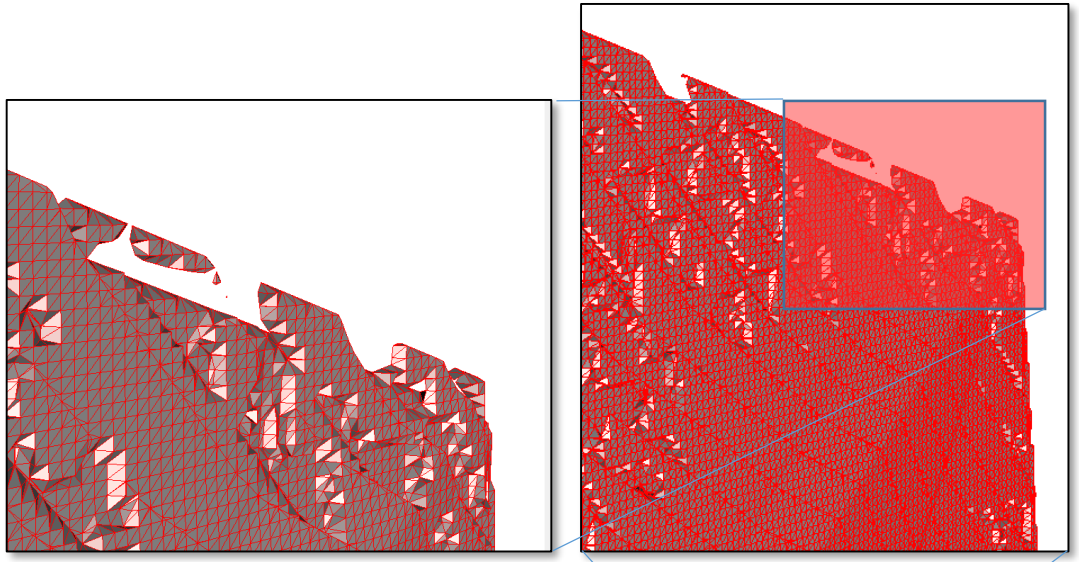


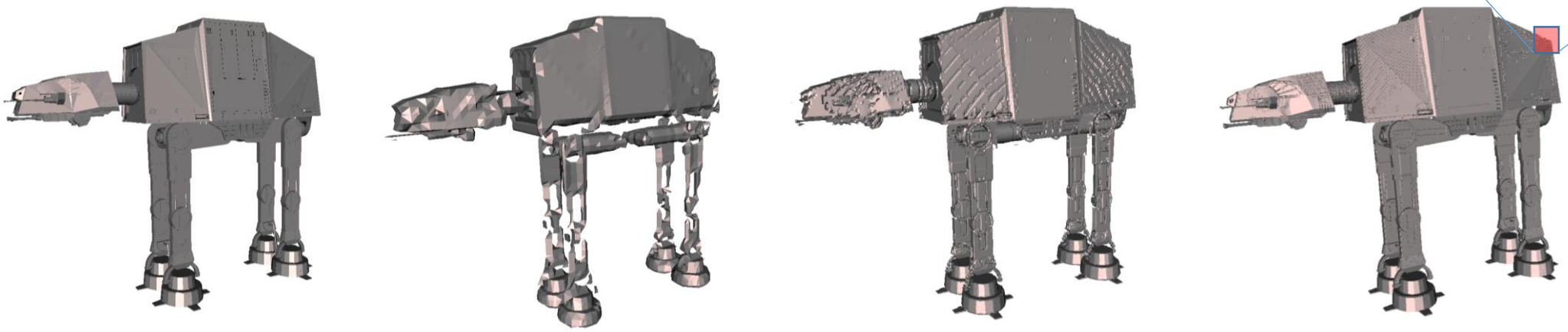
6.8M

Tao Ju, (2004) “Robust Repair of polygonal models", ACM TOG Vol. 23 (SIGGRAPH 2004), pp. 888-895.

Polymender – Concluding remarks

- Polymender
  - Extremely fast, lightweight and robust
  - Any mesh
  - Distortion everywhere
    - Inaccuracy vs. triangle count



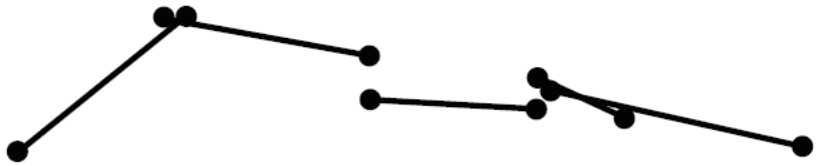


Input (65K faces) 12K 300K 6.8M

Tao Ju, (2004) “Robust Repair of polygonal models", ACM TOG Vol. 23 (SIGGRAPH 2004), pp. 888-895.

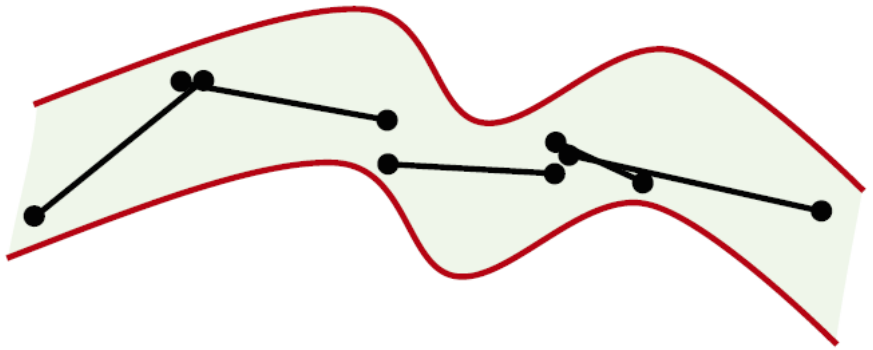
TetWild

- Approximating the original input within $\mathcal{E}$ bound



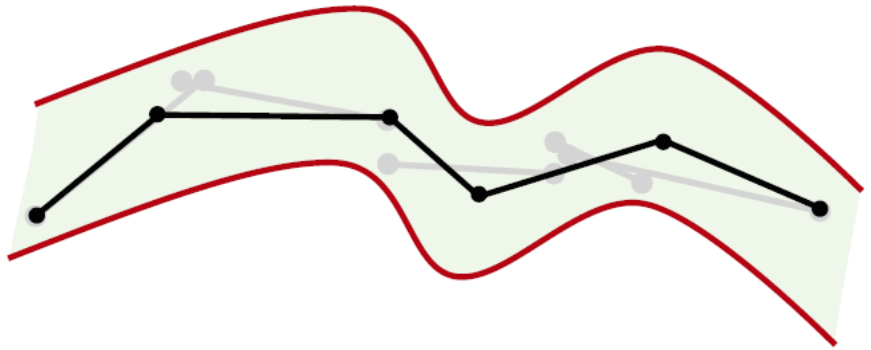
TetWild

- Approximating the original input within $\mathcal{E}$ bound



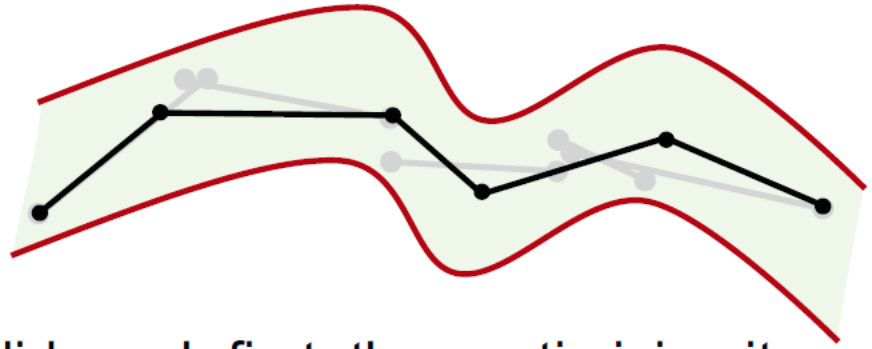
TetWild

- Approximating the original input within $\mathcal{E}$ bound

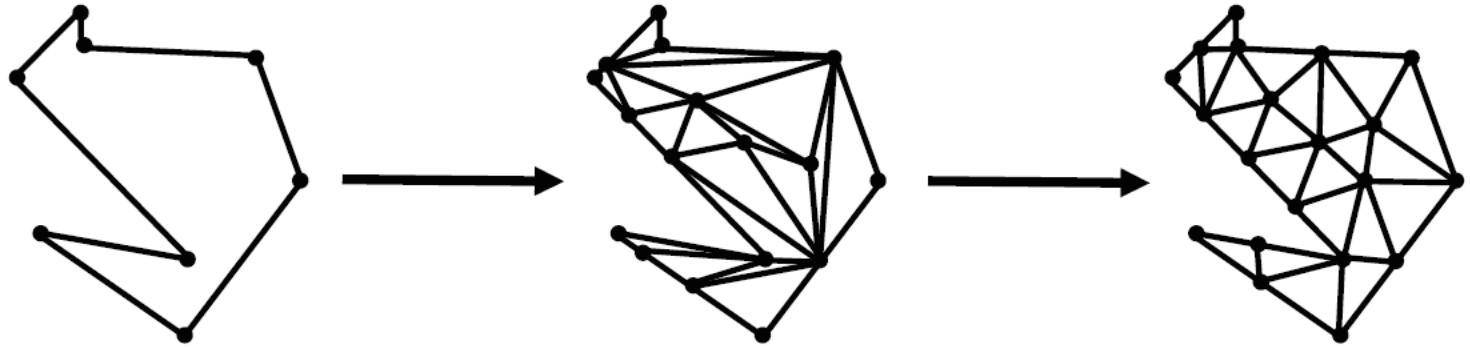


TetWild

- Approximating the original input within $\mathcal{E}$ bound



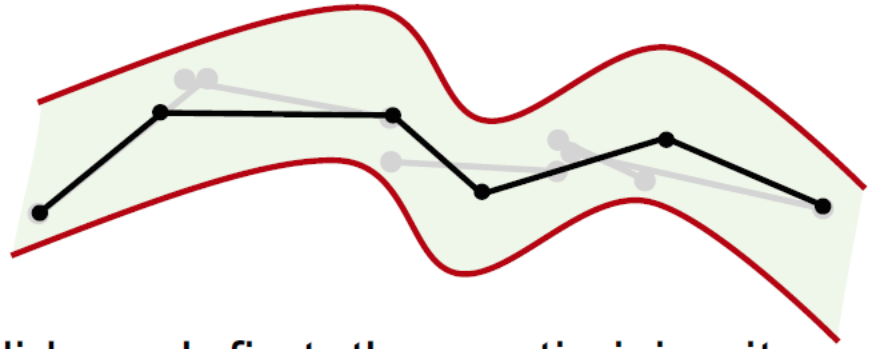
- Generating a valid mesh first, then optimizing its quality as much as possible



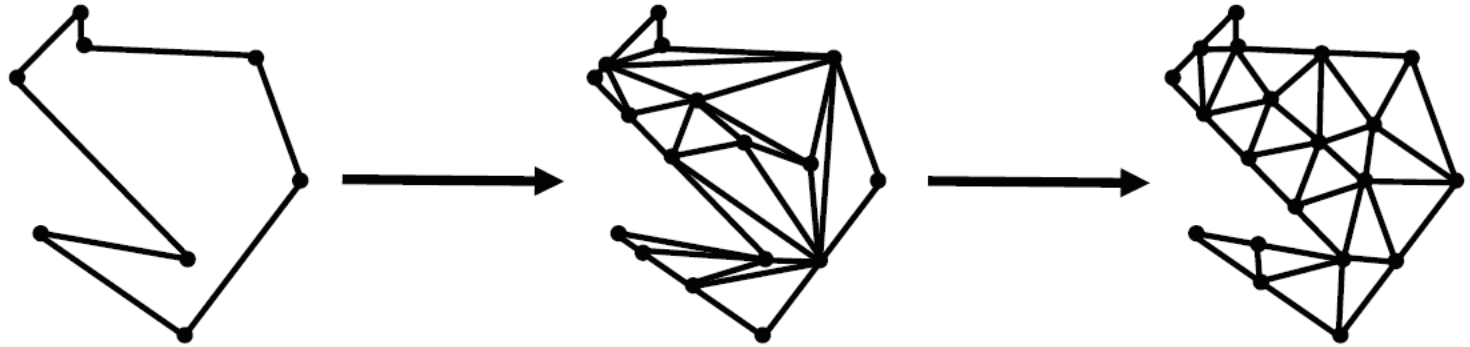
- Exact representation instead of floating precision

TetWild

- Approximating the original input within $\mathcal{E}$ bound

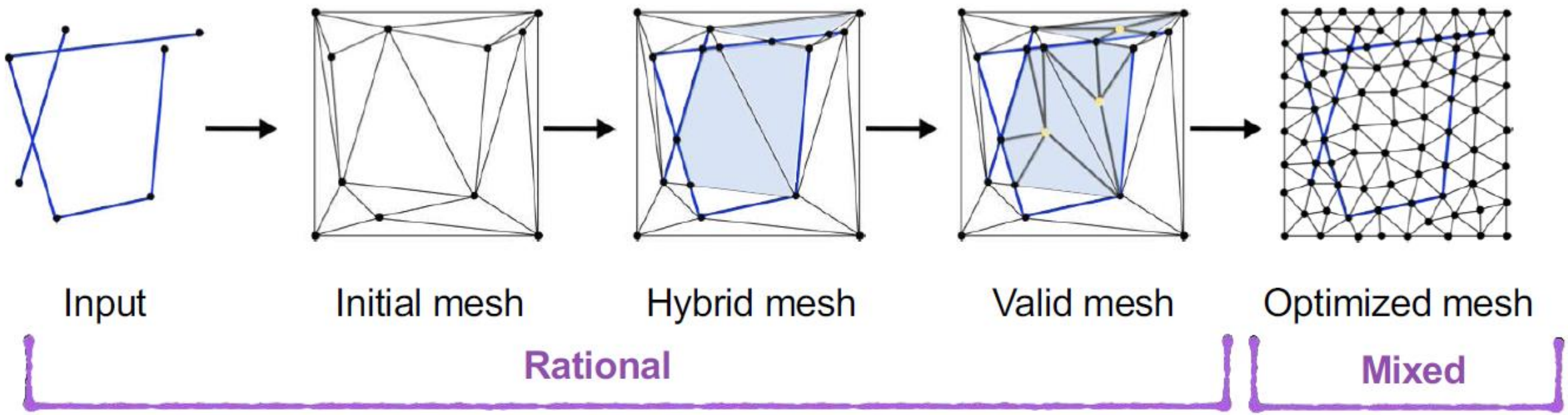


- Generating a valid mesh first, then optimizing its quality as much as possible

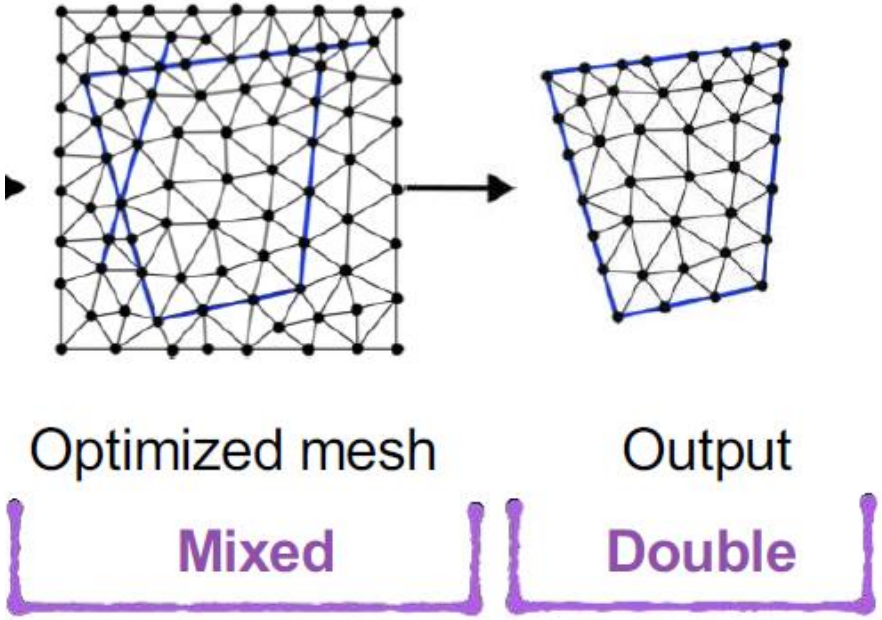


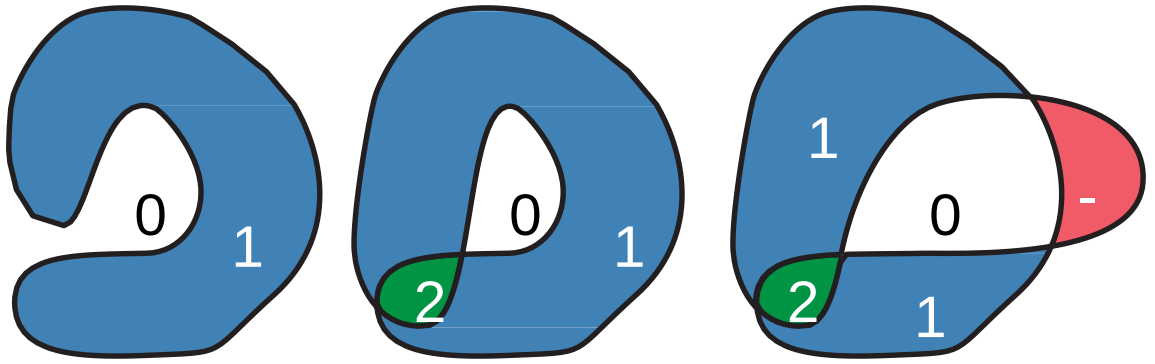
- Exact representation instead of floating precision

Mesh the approximated input



Inside-outside segmentation

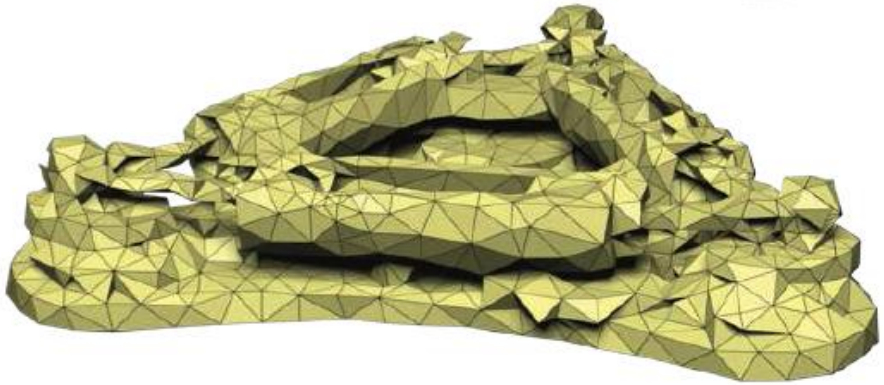


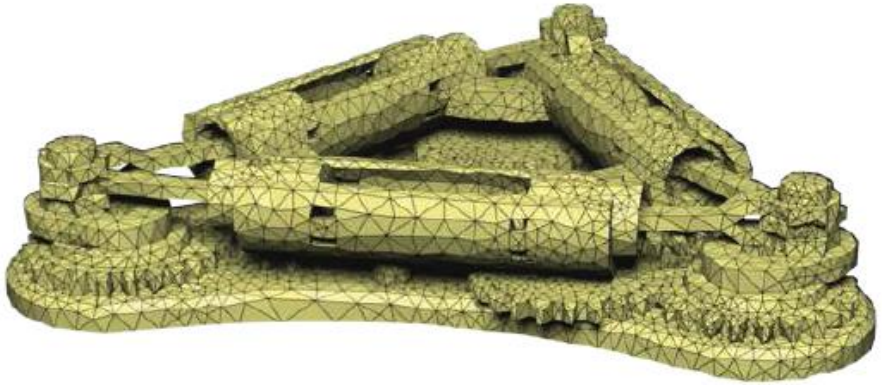


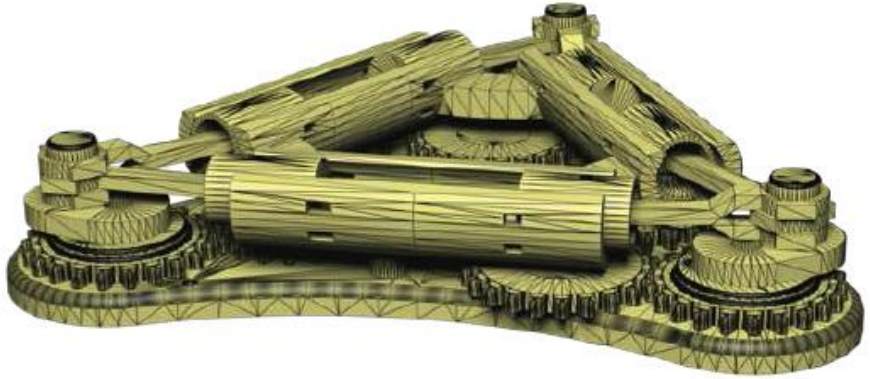
A. Jacobson, L. Kavan, O. Sorkine-Hornung (2013). **Robust Inside-Outside Segmentation using Generalized Winding Numbers**. ACM TOG (Siggraph 2013)

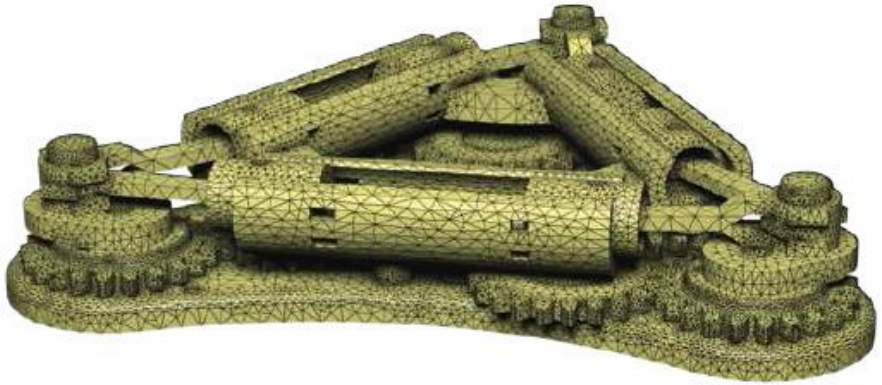
TetWild – Concluding remarks

- TetWild
  - Extremely robust
  - Any mesh
  - Slow
  - Distortion everywhere
    - Inaccuracy vs. triangle count









Y. Hu, Q. Zhou, X. Gao, A. Jacobson, D. Zorin, D. Panozzo. **Tetrahedral Meshing in the Wild**. ACM TOG (SIGGRAPH 2018).

Robust geometric programming

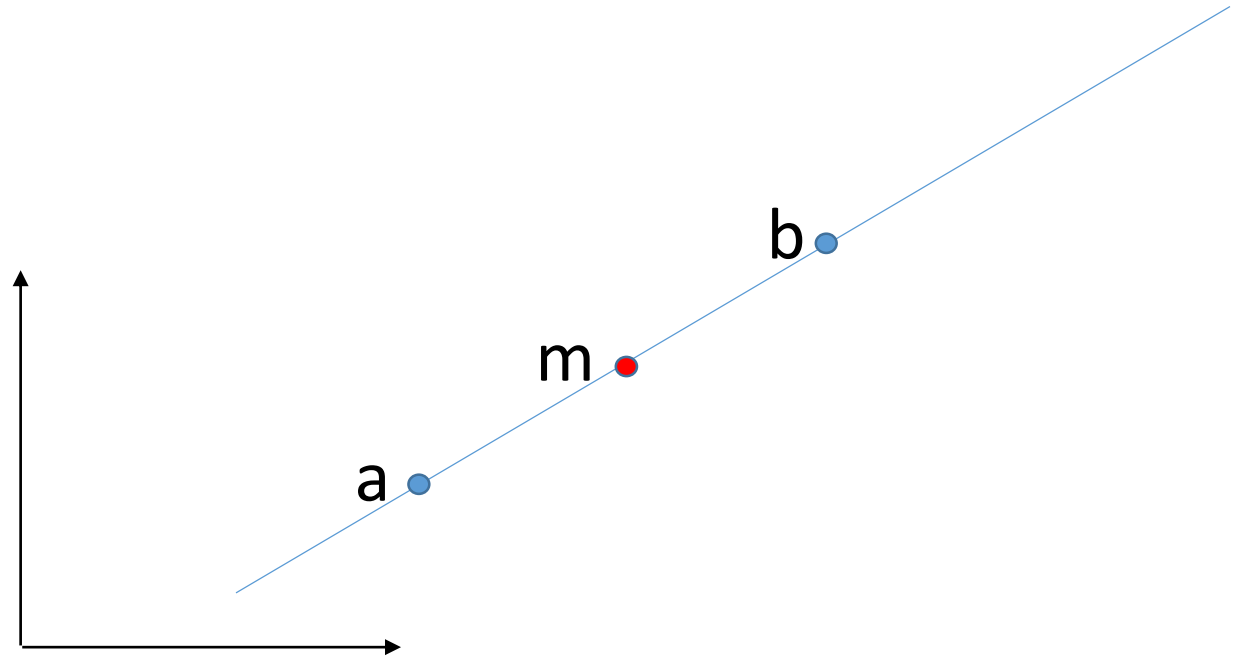
Robust geometric programming

Euclidean space = infinitely many points

YOUR PAPER

m=(**a**+**b**)/2

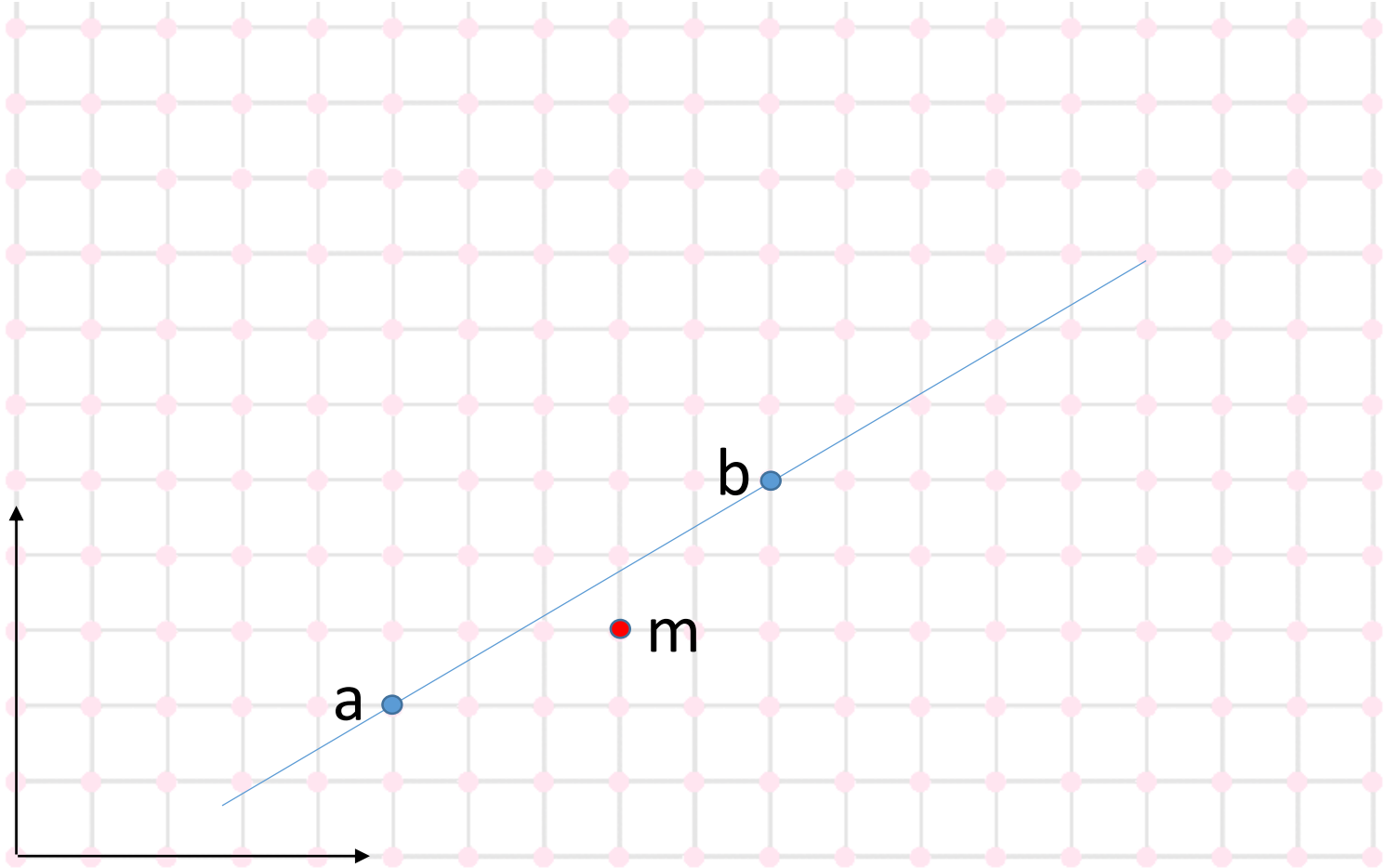
$\mathbb{R}^2$



Finitely many points

m=(**a**+**b**)/2

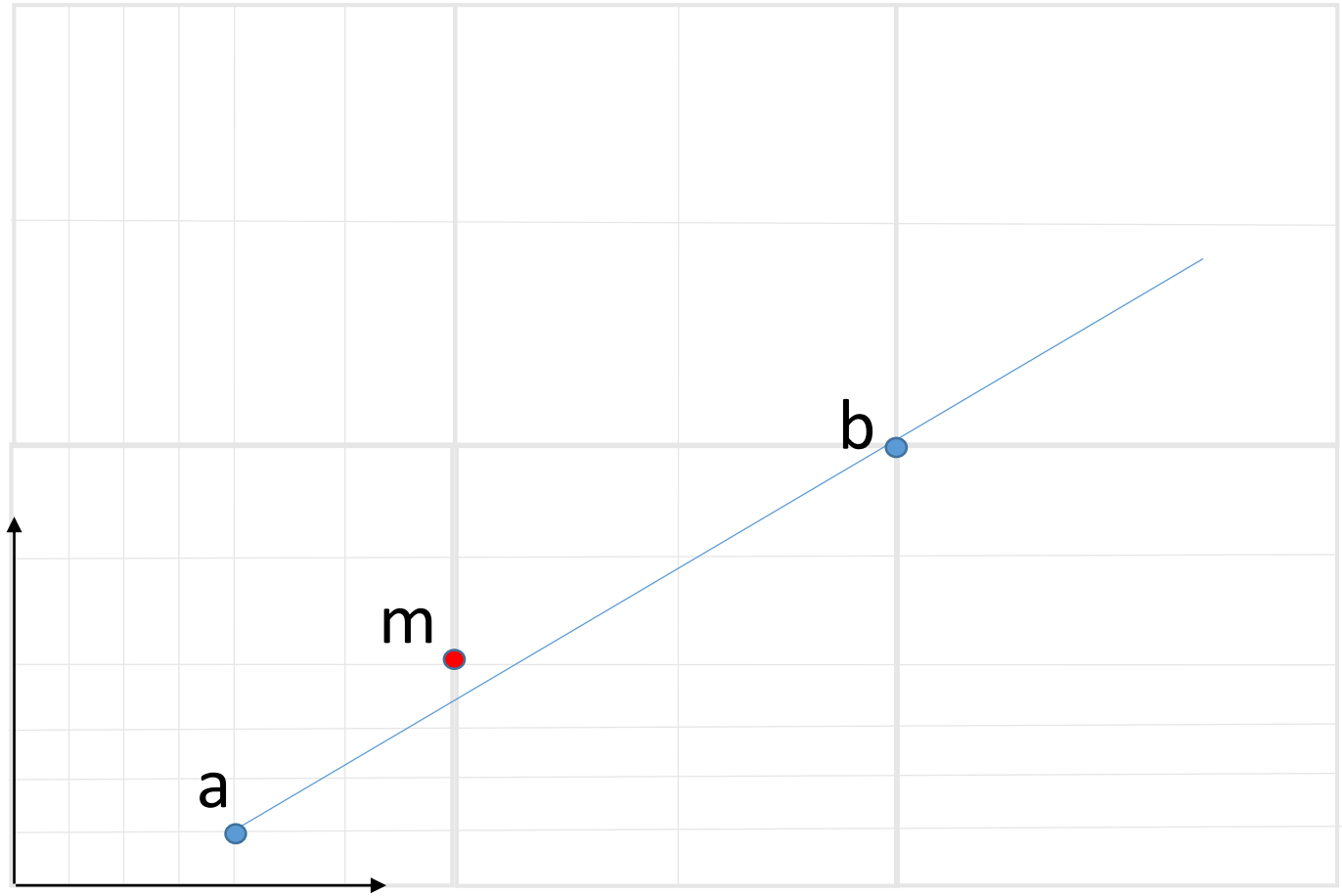
$\mathbb{N}^2$



Finitely many points

m=(**a**+**b**)/2

$\mathbb{F}^2$

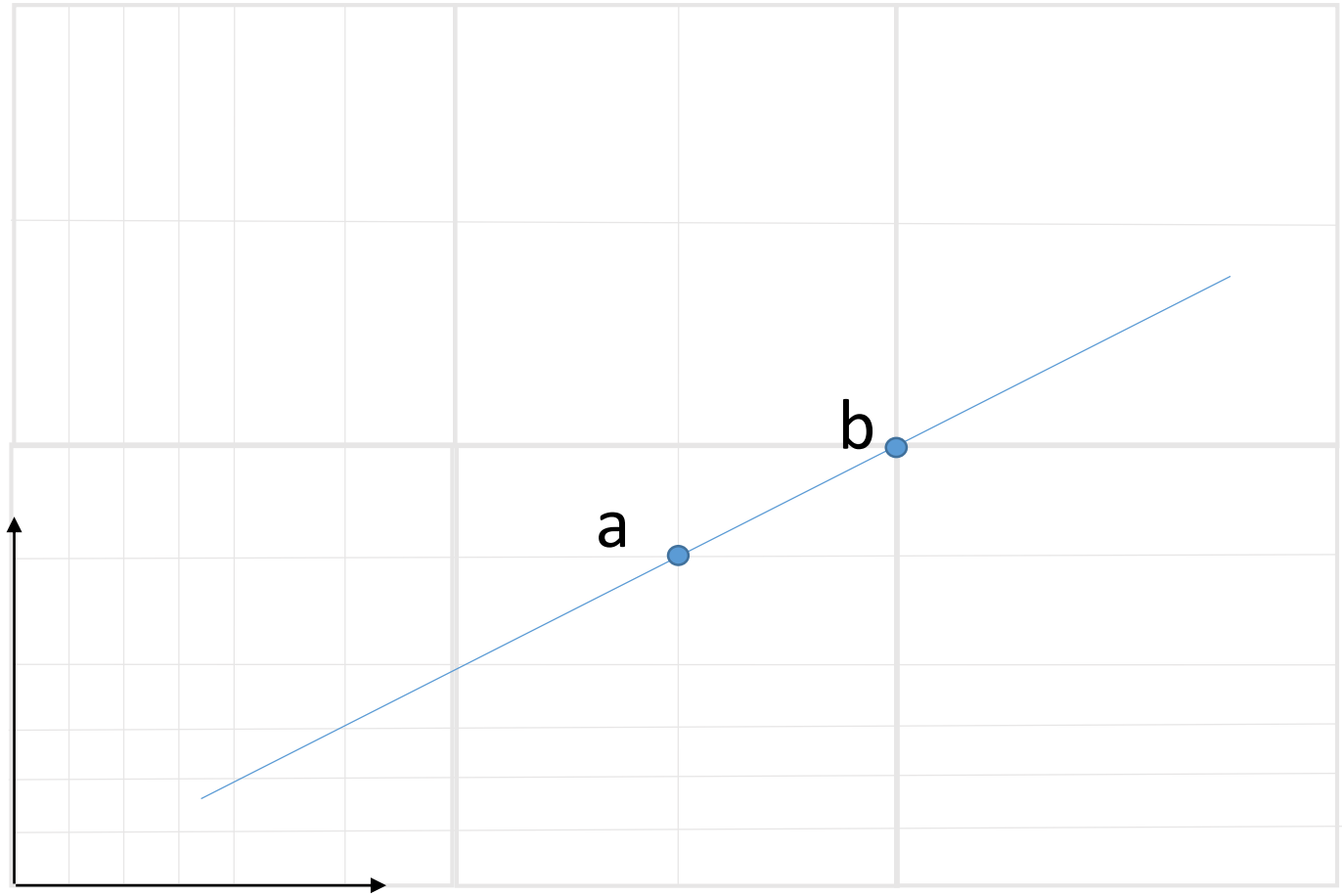


YOUR CODE

Finitely many points

m=(**a**+**b**)/2

$\mathbb{F}^2$

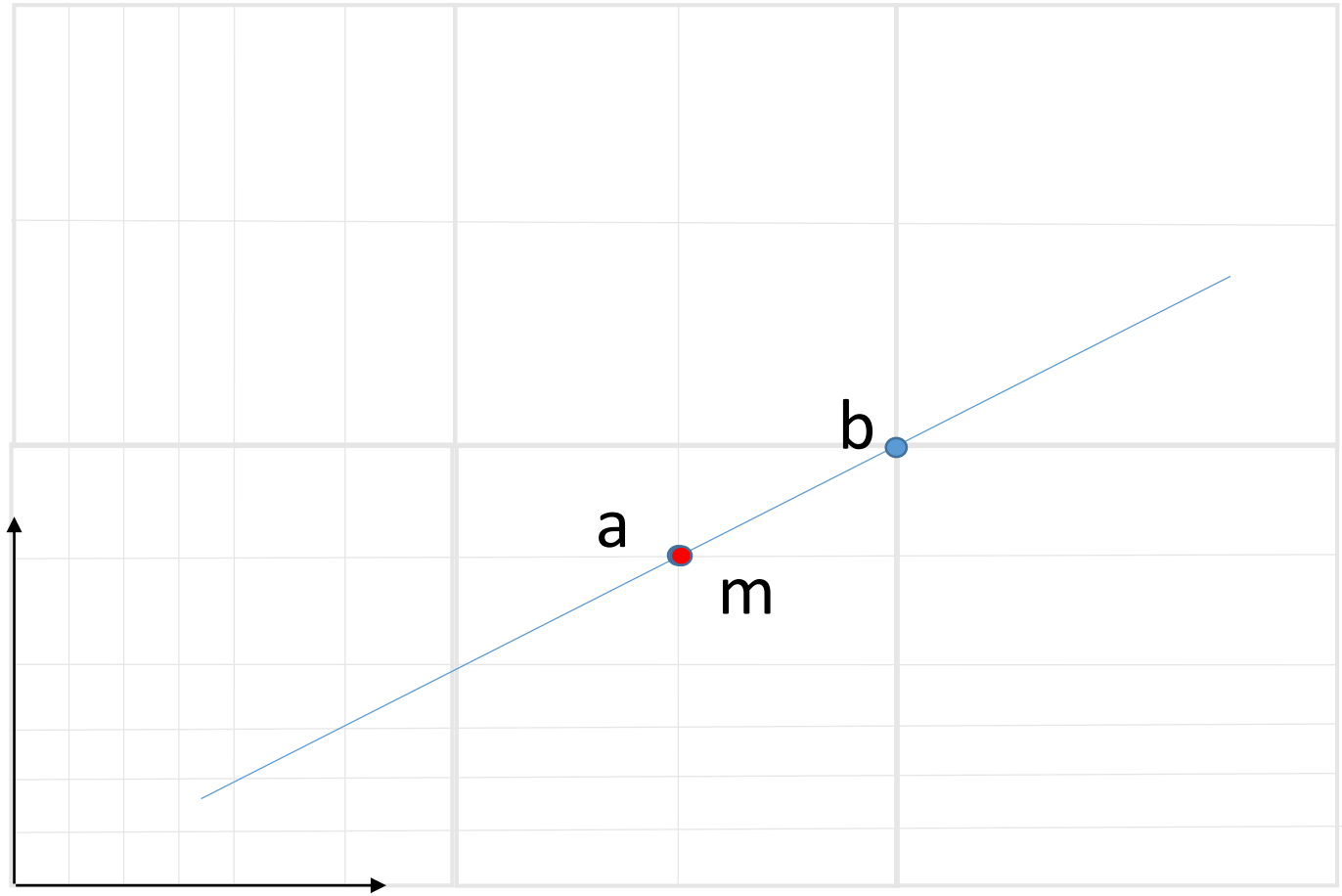


YOUR CODE

Finitely many points

m=(**a**+**b**)/2

$\mathbb{F}^2$



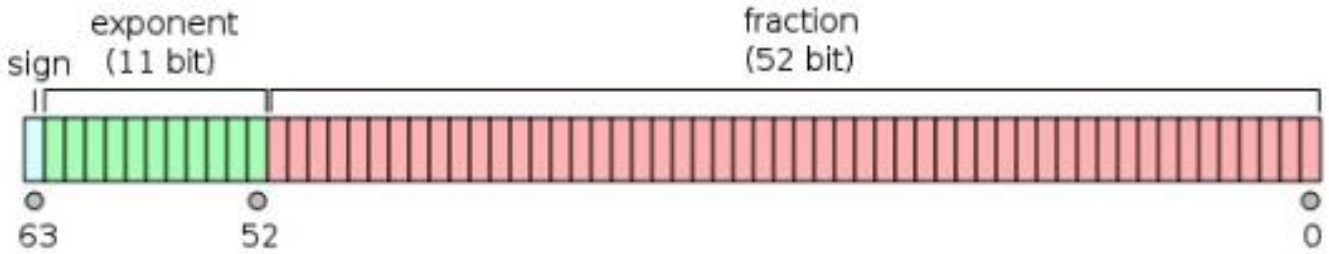
YOUR CODE

Machine precision (IEEE-754)

Sure, but that is a pathological case! Points are not that close to each other in normal applications

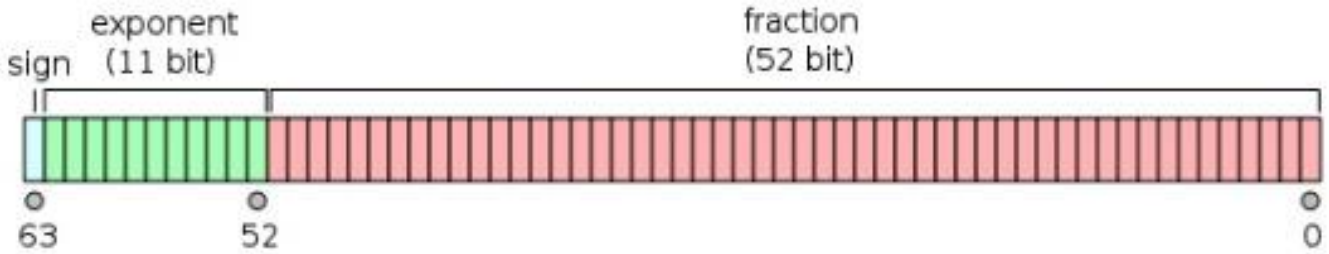
Machine precision (IEEE-754)

Sure, but that is a pathological case! Points are not that close to each other in normal applications



Machine precision (IEEE-754)

Sure, but that is a pathological case! Points are not that close to each other in normal applications



Unit in the last place (or «of least precision»)

$\mathrm{ULP}(x) = b-a, \quad \text{s.t. } a \le x \le b,\ a \ne b,\ x \in \mathbb{R},\ a,b \in \mathbb{F}$

$\mathrm{ULP}(1) = 2^{-52}$, $\mathrm{ULP}(2^{52}) = 1$, …

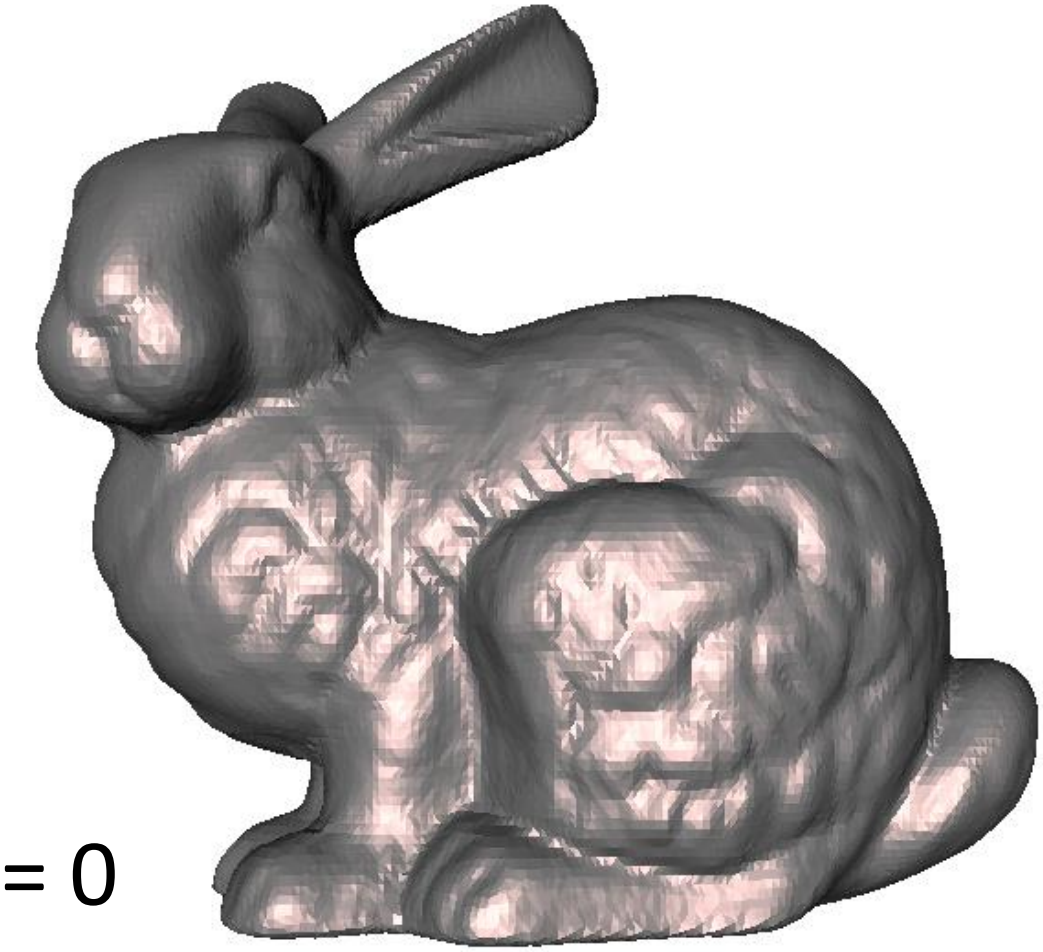
$\mathrm{ULP}(2^{1023}) = 2^{971} = 1.99584e{+}292$

Impact on geometry

```
// Using double precision
Load_bunny(vertices, faces);
double k = pow(2, e);
Point3 d(k, k, k);
for (Point3 v : vertices) v += d;
Save_bunny(vertices, faces);
```

Impact on geometry

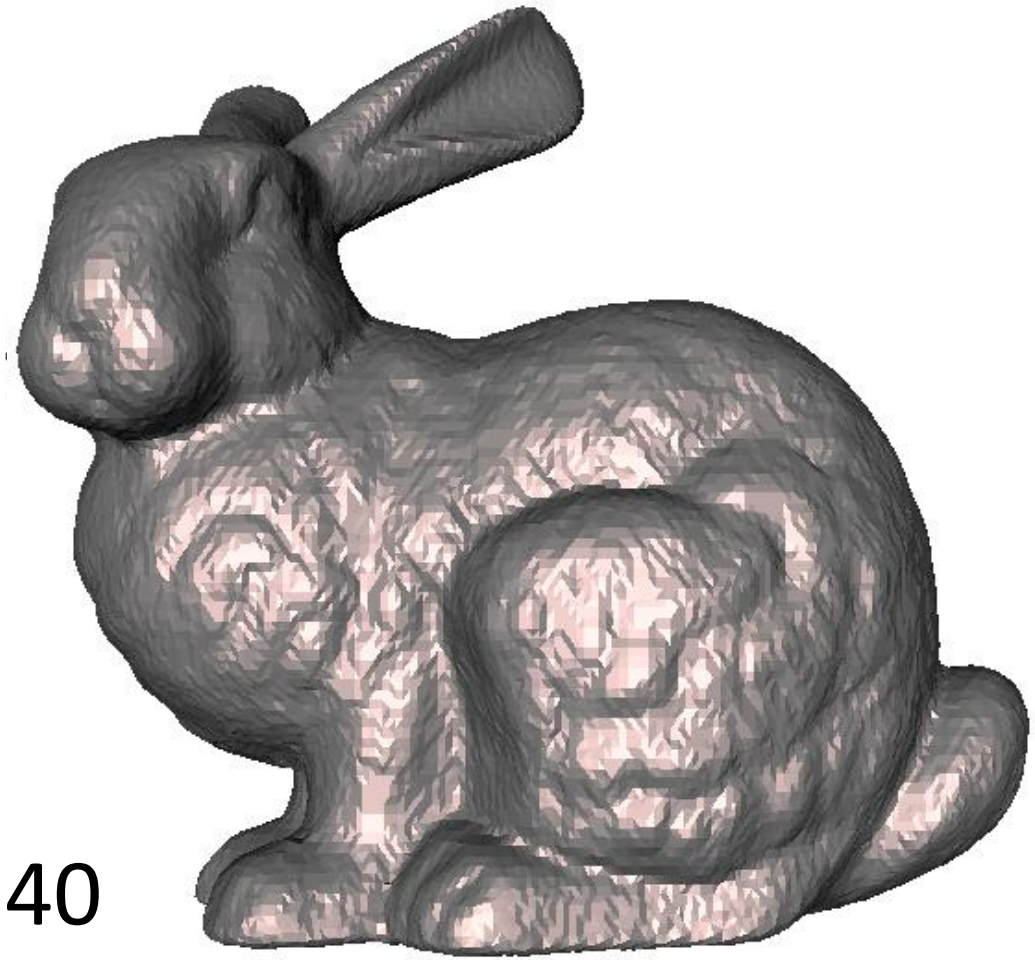
```
// Using double precision
Load_bunny(vertices, faces);
double k = pow(2, e);
Point3 d(k, k, k);
for (Point3 v : vertices) v += d;
Save_bunny(vertices, faces);
```



e = 0

Impact on geometry

```
// Using double precision
Load_bunny(vertices, faces);
double k = pow(2, e);
Point3 d(k, k, k);
for (Point3 v : vertices) v += d;
Save_bunny(vertices, faces);
```



e = 40

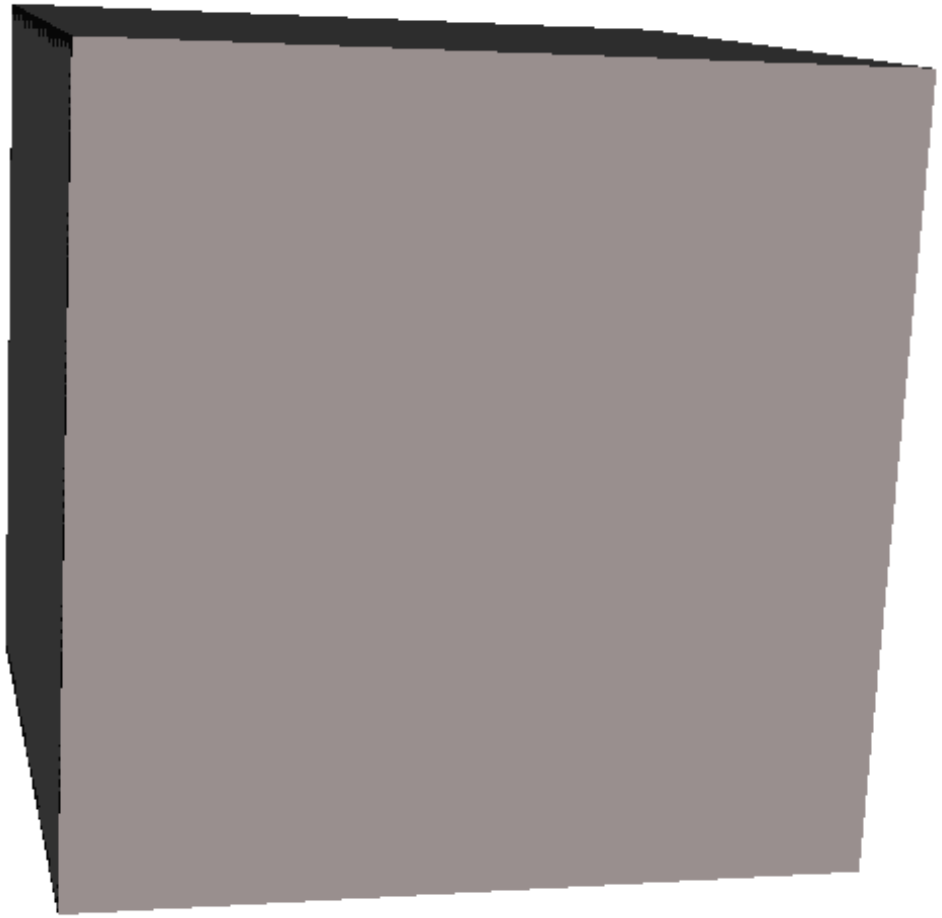
Impact on geometry

```
// Using double precision
Load_bunny(vertices, faces);
double k = pow(2, e);
Point3 d(k, k, k);
for (Point3 v : vertices) v += d;
Save_bunny(vertices, faces);
```

e = 45

Impact on geometry

```
// Using double precision
Load_bunny(vertices, faces);
double k = pow(2, e);
Point3 d(k, k, k);
for (Point3 v : vertices) v += d;
Save_bunny(vertices, faces);
```



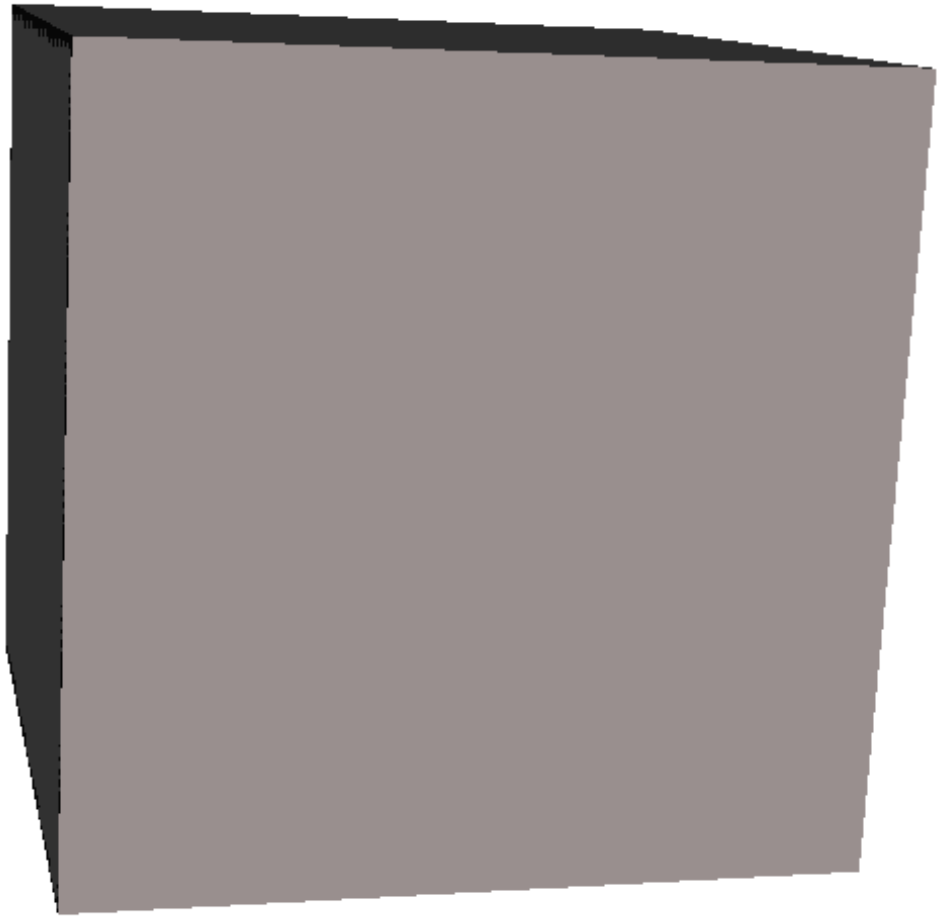
e = 49

Impact on geometry

```
// Using double precision
Load_bunny(vertices, faces);
double k = pow(2, e);
Point3 d(k, k, k);
for (Point3 v : vertices) v += d;
Save_bunny(vertices, faces);
```

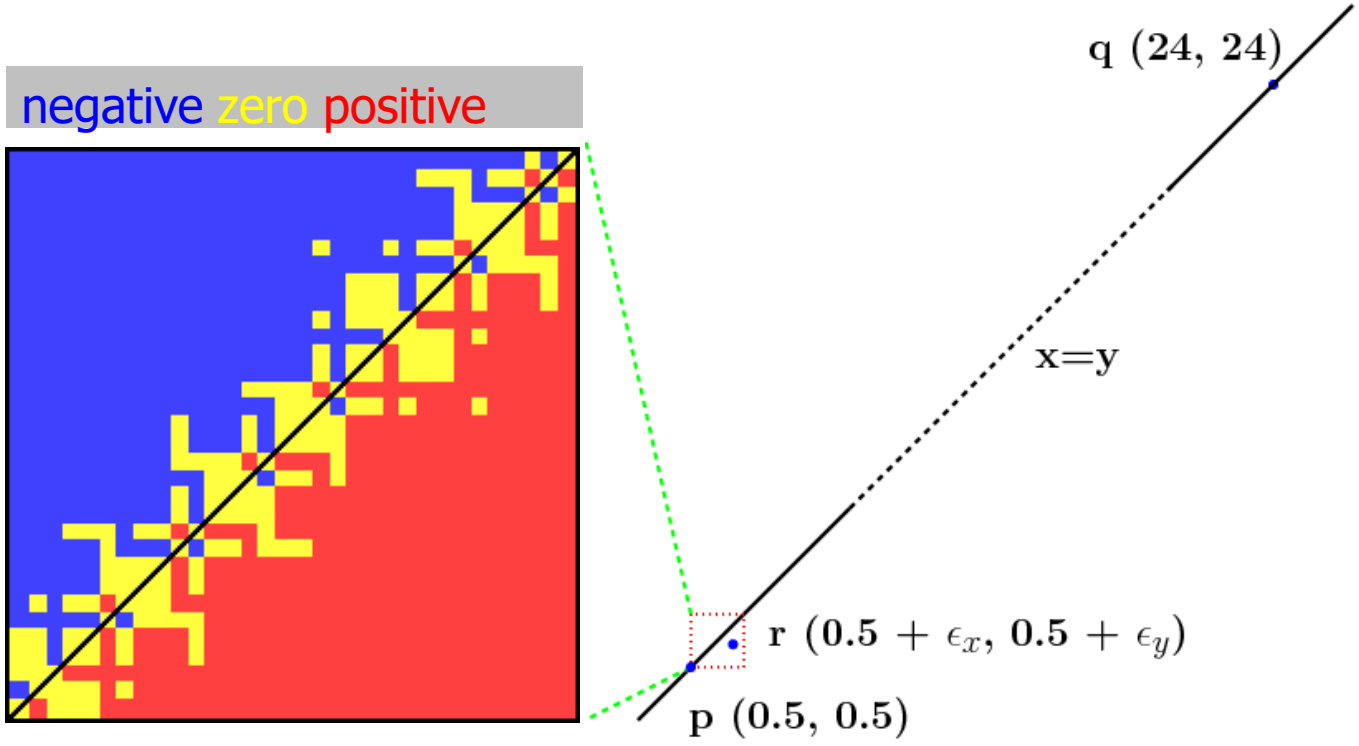
Only a += operation here
No error accumulation!!!

e = 49



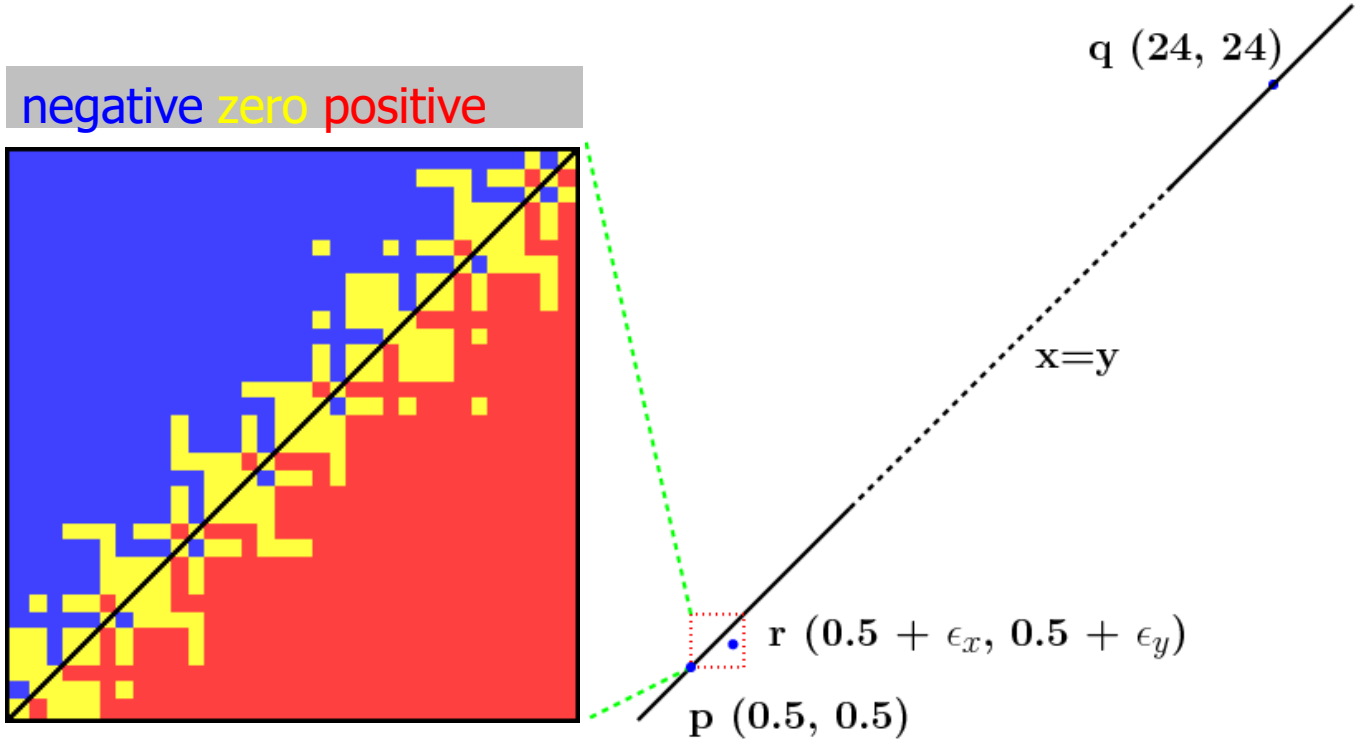
Impact on program flow

$$orientation(p, q, r) = sign\big((q_x - p_x)(r_y - p_y) - (q_y - p_y)(r_x - p_x)\big)$$



Impact on program flow

$$orientation(p, q, r) = sign\big((q_x - p_x)(r_y - p_y) - (q_y - p_y)(r_x - p_x)\big)$$



E.g. broken invariant in incremental insertion

Tolerances

Ok, so let's consider p,q,r to be aligned if
|*orientation(p,q,r)*| < epsilon

Tolerances

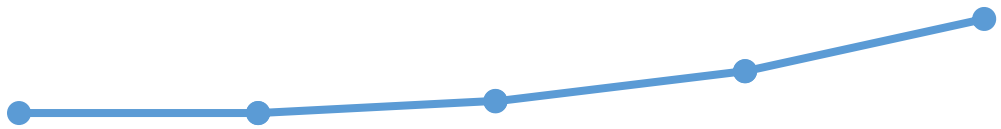
Ok, so let’s consider p,q,r to be aligned if

|*orientation(p,q,r)*| < epsilon

Tolerances

Ok, so let’s consider p,q,r to be aligned if

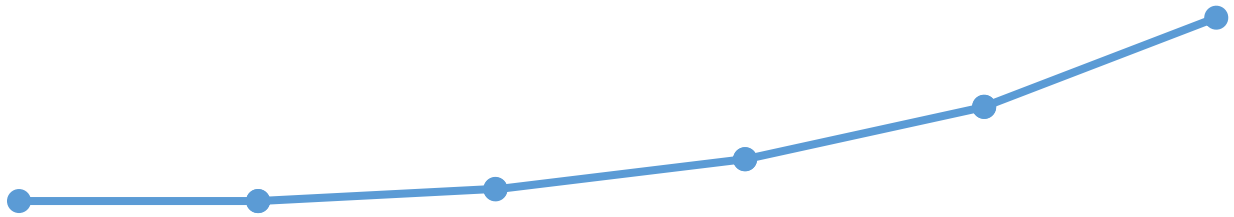
|*orientation(p,q,r)*| < epsilon



Tolerances

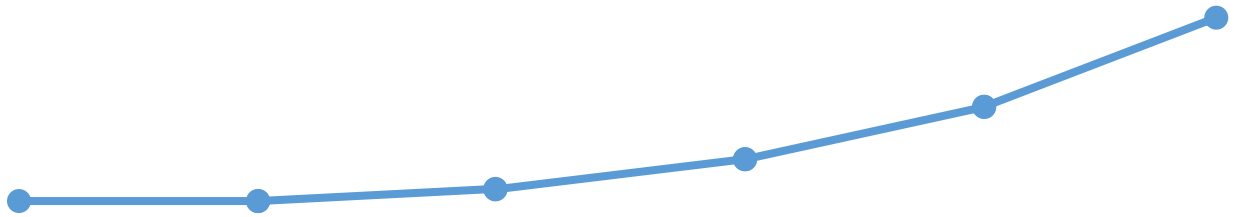
Ok, so let’s consider p,q,r to be aligned if

|*orientation(p,q,r)*| < epsilon



Tolerances

Ok, so let's consider p,q,r to be aligned if
$|orientation(p,q,r)| < epsilon$

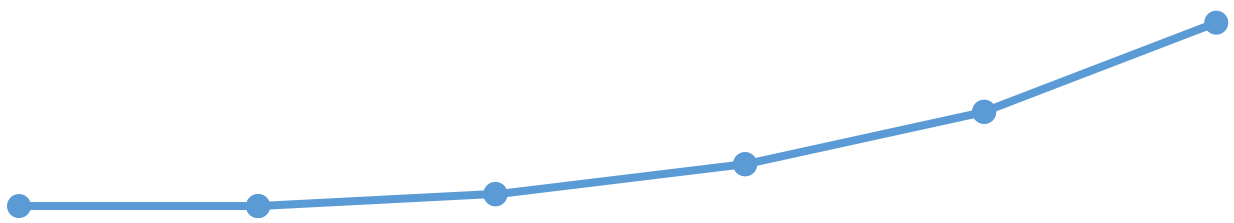


No longer transitive!

collinear(a,b,c) && collinear(b,c,d) $\Rightarrow$ collinear(a,b,d)

Tolerances

Ok, so let's consider p,q,r to be aligned if
$|orientation(p,q,r)| < epsilon$



- Must predict non-trivial behaviour (hard coding)
- Loose convergence guarantees
- Which epsilon?
  - Depends on coordinates

No longer transitive!

collinear(a,b,c) && collinear(b,c,d) $\Rightarrow$ collinear(a,b,d)

Exact Geometric Computing

- Use exact numbers instead of floats (e.g. GNU GMP, MPIR, LEDA, …)

Chee-Keng Yap
Towards exact geometric computation
Computational Geometry 7(1-2), 1997

Exact Geometric Computing

- Use exact numbers instead of floats (e.g. GNU GMP, MPIR, LEDA, …)
  - 20 times slower (in controlled «test» conditions)

Chee-Keng Yap
Towards exact geometric computation
Computational Geometry 7(1-2), 1997

Exact Geometric Computing

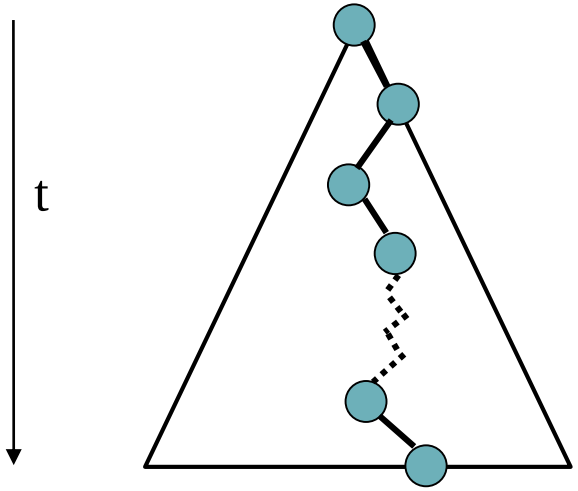
- Use exact numbers instead of floats (e.g. GNU GMP, MPIR, LEDA, …)
  - 20 times slower (in controlled «test» conditions)
  - **100 times** slower (in practice, if naively used)

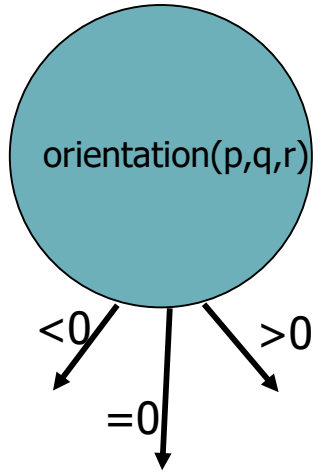
Chee-Keng Yap
Towards exact geometric computation
Computational Geometry 7(1-2), 1997

Exact Geometric Computing

- Use exact numbers instead of floats (e.g. GNU GMP, MPIR, LEDA, …)
  - 20 times slower (in controlled «test» conditions)
  - **100 times** slower (in practice, if naively used)
- Do we really need exactness everywhere?
  - Floating point approximations can be tolerated…
    - as long as they do not change the expected program flow

Chee-Keng Yap
Towards exact geometric computation
Computational Geometry 7(1-2), 1997





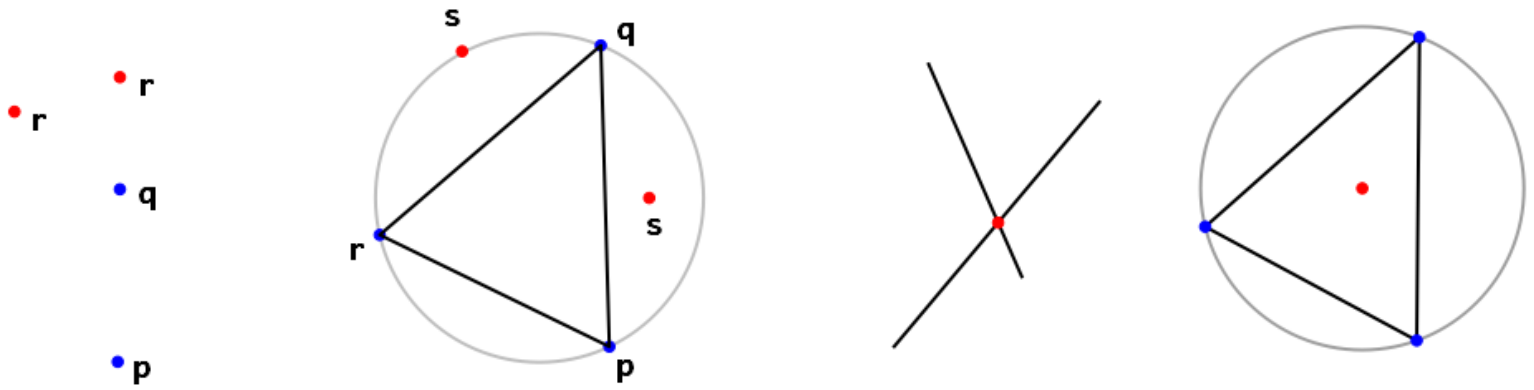
Predicates and constructions

Predicates and constructions

- Need exact constructions?
  - **NO** = The program flow is exactly determined by input values only
    - E.g. Delaunay triangulation
  - **YES** = The program flow depends on «intermediate» values
    - E.g. Mesh booleans

Predicates and constructions

- Need exact constructions?
  - **NO** = The program flow is exactly determined by input values only
    - E.g. Delaunay triangulation
  - **YES** = The program flow depends on «intermediate» values
    - E.g. Mesh booleans



orientation in_circle intersection circumcenter

Approaches to geometric robustness

- IF no exact constructions needed
  - «A la Shewchuk» predicates (1.6 times slower. Only algebraic functions)
  - Interval arithmetic filters (3-8 times slower. More flexible)
- ELSE
  - Lazy exact evaluation
    - CGAL (20 times slower for reasonable construction depths)
  - Hybrid arithmetic
    - TMesh (3 times slower if constructions are sparse)
    - Example app: TetWild

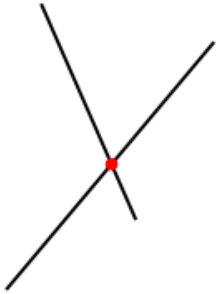
Arbitrary precision

- GNU GMP / MPIR
  - int = type for integer numbers in the range [-INT_MAX, INT_MAX]
  - Gmpz = type for **arbitrarily <u>large</u>** integer numbers
    - A vector of int

$$N = \sum_{k=0}^{n-1} a[k] * 2^{32k}$$

Arbitrary precision

- GNU GMP / MPIR
  - int = type for integer numbers in the range [-INT_MAX, INT_MAX]
  - Gmpz = type for **arbitrarily <u>large</u>** integer numbers
    - A vector of int
  - Gmpq = type for rational numbers
    - A pair of Gmpz = numerator/denominator
    - **Arbitrarily <u>precise</u>**



$$N = \sum_{k=0}^{n-1} a[k] * 2^{32k}$$

Floating point expansions

- Used in Shewchuk's predicates and in Levy's Geogram
  - double = type for FP numbers in the range [-DBL_MAX, DBL_MAX]
  - expansion = an **arbitrarily <u>precise</u>** floating point number
    - A vector of double
    - Not arbitrarily large. Overflows can still occur!

$$N = \sum_{k=0}^{n-1} f[k]$$

Floating point expansions

- Used in Shewchuk's predicates and in Levy's Geogram
  - double = type for FP numbers in the range [-DBL_MAX, DBL_MAX]
  - expansion = an **arbitrarily <u>precise</u>** floating point number
    - A vector of double
    - Not arbitrarily large. Overflows can still occur!

$$N = \sum_{k=0}^{n-1} f[k]$$

No need to multiply by 2^x. Exponent is already in FP representation

Arithmetic on expansions

- Expansions can be summed, subtracted or multiplied
  - Without any approximation error!
- Let $|e|$ denote the «length» of an expansion e
  - the number of double «components» to be summed to form N
- A double is an expansion of length 1
- The result of arithmetic operations has bounded length

$$N = \sum_{k=0}^{n-1} f[k]$$

$$|e_1 + e_2| \leq |e_1| + |e_2|$$

$$|e_1 - e_2| \leq |e_1| + |e_2|$$

$$|e_1 * e_2| \leq 2|e_1||e_2|$$

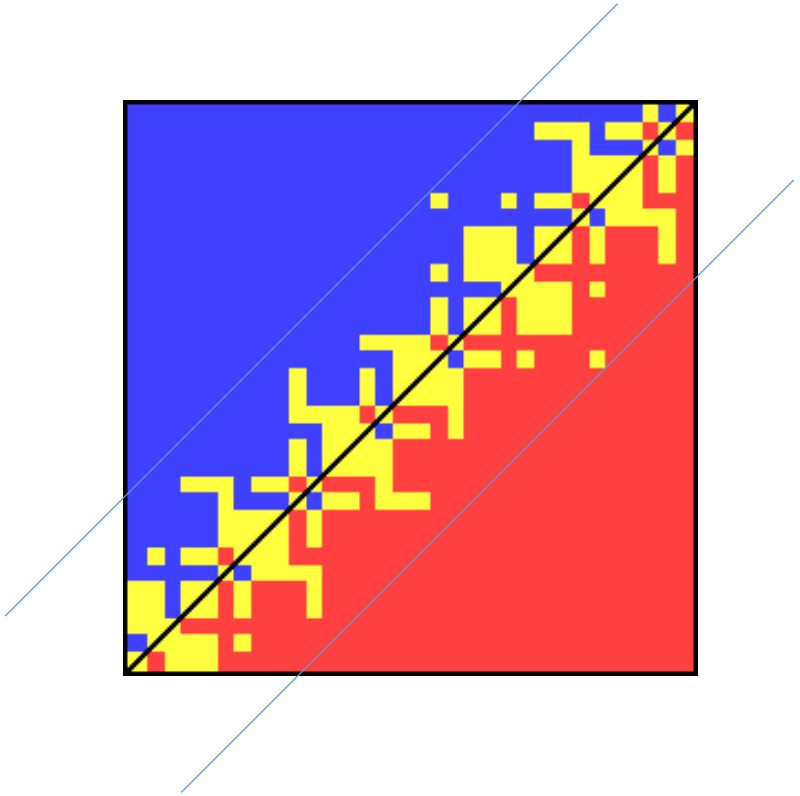
Geometric predicates with expansions

- Orient2d, orient3d, orient4d, incircle, insphere, …
- Each of them is the sign of a (low degree) polynomial
  - Thus involving only +, -, * operations
- The sign of an expansion is the sign of its dominant component

- Still too slow
  - We do not need the polynomial to be exactly evaluated…
    - …as long as its sign is correct
  - Calculate using standard floating point arithmetic
    - Switch to expansion arithmetic only if unsure

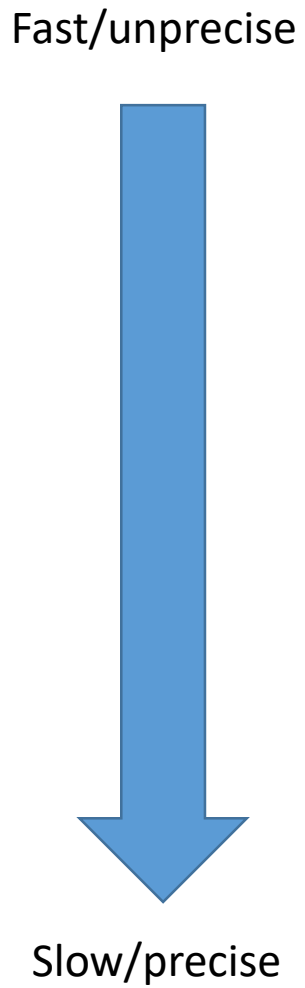
Filters

- Let D be the value of the polynomial calculated using plain floating point arithmetic
- D has a correct sign if it is far enough from zero
  - How far?

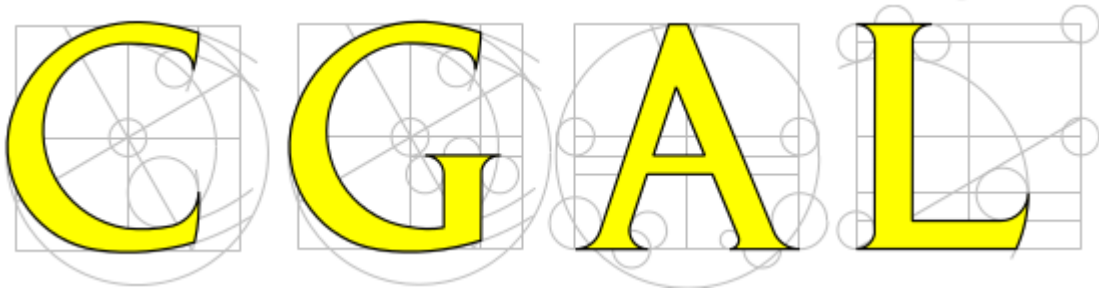


Filters

- Sign(D) is correct if $|D| > \varepsilon$
- Various filtering approaches
  - Static filter:
    - ε is constant and depends on the polynomial only
    - + Can be precomputed once for all at compile time. Very efficient
    - - Too pessimistic -> too many switches to exact computation
    - - Assumption on the range of input values
  - Almost static filter:
    - ε is initialized based on optimistic assumptions
    - It is adjusted if necessary
  - Dynamic filter
    - ε depends on the actual accumulated rounding error based on the specific input values
    - + Extremely precise -> few switches to exact computation
    - - Must be calculated at each call
- Multi-stage filters



Predicates in CGAL



- Extremely flexible and generic
- Precompute static filters
- Compute predicate with floating point arithmetic
- If result is uncertain (static filter fails):
  Compute predicate with interval arithmetic (dynamic filter)

- If result is uncertain (dynamic filter fails):
  Compute predicate with arbitrary precision arithmetic

Shewchuk's predicates

- One of the fastest approaches
- Uses adaptivity
  1. Evaluate polynomial using floating point arith
  2. Use filter to check if precision is sufficient. If so, return
  3. Increase precision (intermediate expansions) and re-use partial results from (1)
  4. Use 2nd stage filter. If precision sufficient, return
  5. Increase precision and re-use partial results from (1) and (3)
  6. …
  7. If even last filter fails, switch to full expansion arithmetic and return
- Currently used in state-of-the-art 2D and 3D meshing software
  - Triangle (J. R. Shewchuk)
  - TetGEN (H. Si)

Expansions - summary

- Extremely fast
- Fully exploit FPU acceleration
- Pure C code - do not require external libraries

- Difficult to code
  - Shewchuk's predicates (orient2d, incircle, orient3d, insphere) > 4200 lines of C code
- Set of expansions is closed under +, -, * operations only
  - Proposals to support division (Priest), but limitation is intrinsic due to possible infinite representations (e.g. 1/3)
- Suitable only if exact constructions are not necessary

Interval Arithmetic

- The smallest floating point interval containing a real number *x* (e.g. the result of an operation)
  - Can be a single value (if $x \in \mathbb{F}$)
- The sign of a polynomial computed using intervals is certainly correct if the interval does not contain the zero
- FPU rounding mode can be set to calculate tight intervals
- Provably efficient dynamic filter

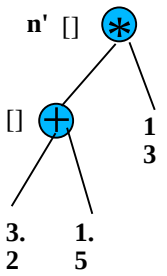
Let $a \in \mathbb{R}$, and let $a,b \in \mathbb{F}$. $[a,b] = \{x \in \mathbb{R} : a \leq x \leq b\}$

Exact constructions

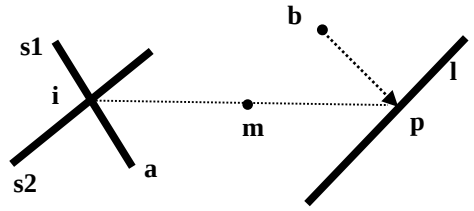
- Naive approach
  - Use exact arithmetic
- Lazy evaluation
  - Arithmetic expressions
  - Geometric expressions

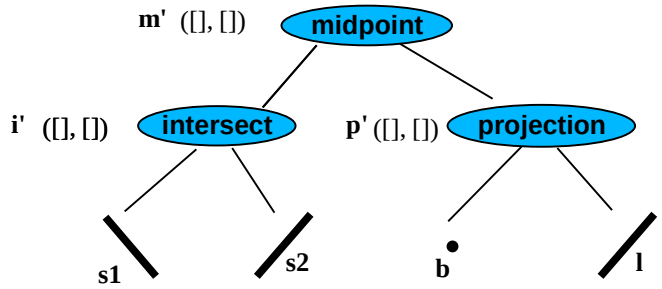
Lazy number = interval and arithmetic expression tree

$$(3.2 + 1.5) * 13$$



Lazy object = approximated object and geometric operation tree





Test that may trigger an exact re-evaluation:

```
if ( n' < m' )
```

```
if (collinear(a',m',b'))
```

`CGAL::Lazy_kernel<NT>`

TMesh hybrid kernel – basic type

- What if exact constructions are needed at sparse spots only?

- Polymorphic number type PM_Rational (alias coord)
  - Internally, a PM_Rational can be either a double or an exact rational number
  - This underlying representation is called the «subtype» of the PM_Rational
  - Interoperability is guaranteed and transparent
- Example:
  - a, b and c are all PM_Rational
  - The subtypes of ‘a’ and ‘b’ are double and rational respectively
  - The expression c = a + b is valid
  - The subtype of ‘c’ depends on the current «precision level»
  - Precision level can be changed by the program at runtime

TMesh hybrid kernel

- Precision level
  - Determines the subtype of newly created PM_Rationals
  - Can be either «rational» or «floating point»
  - It is a program/thread state
- Subtype (double or rational)
  - Existing PM_Rational objects maintain their subtype
  - The subtype of new objects depends on current precision level
- Result of a comparison/predicate
  - Always exact, independently of the subtype of operands
  - Speed depends on subtype of operands
- Result of a construction
  - Constructions are arithmetic only
  - Exact if current precision level is «rational»

Using TMesh in your program

- Download and compile TMesh
  - https://github.com/MarcoAttene/TMesh_Kernel
- Configure your program code to use TMesh
  - paths to headers, static lib, program initialization

```
#include "tmesh_kernel.h"

using namespace T_MESH;

int main()
{
    TMesh::init(); // This is mandatory to correctly set up the FPU for exact computation
```

Using TMesh in your program

```
// The type 'coord' represents the basic hybrid number in TMesh.
// It can be handled as an IEEE 754 standard double in most cases.
coord q_x = 3.0;
coord q_y = 4.0;
coord q_z = 5.0;

// for example...
q_x += (q_y - q_z);

// When necessary, the TMESH_TO_DOUBLE macro approximates a coord to an actual double
double sqrtex = sqrt( TMESH_TO_DOUBLE(q_y) );

// And a double can be seamlessly assigned to a coord
q_z = sqrtex;
```

Basic 3D geometry in TMesh

```
// A Point3c in TMesh is a triplet of coordinates, each having type 'coord'
// The following example shows how to manipulate points as vectors
// Here we calculate the projection of 'p' on the plane by A, B and C
Point3c q(q_x, q_y, q_z);
Point3c A(0, 0, 0);
Point3c B(2, 0, 0);
Point3c C(0, 2, 0);

Point3c BA = B - A;
Point3c CA = C - A;
Point3c plane_vec = BA & CA; // Operator '&' represents the cross product

// The previous three lines can be compacted into a single line as follows:
// Point3c plane_vec = (B - A) & (C - A);

// Here we calculate the projection as the intersection of a straight line by
// q and orthogonal to the plane.
Point3c lifted_q = q + plane_vec;
Point3c projected_point = Point3c::linePlaneIntersection(q, lifted_q, A, B, C);
```

Precision level and predicates

```
// By default, TMesh works in floating point mode

// Declare four coplanar points
Point3c p1(5, 1, 0);
Point3c p2(4, 2, 0);
Point3c p3(8, 1, 1);
Point3c p4(7, 2, 1);

// Predicates in TMesh are always exact - the following reports coplanarity as expected
// even if we work with floating point coordinates
if (p1.exactOrientation(p2, p3, p4) == 0) std::cout << "Points pi are coplanar\n";
else std::cout << "Points pi are not coplanar\n";
```

```
Points pi are coplanar
```

Precision level and predicates

```
// We now create scaled copies of the pi's
Point3c q1 = p1 / 3;
Point3c q2 = p2 / 3;
Point3c q3 = p3 / 3;
Point3c q4 = p4 / 3;

// Coordinates in the qi's are not exact scales of those in the pi's because we are
// approximating them using floating point numbers
if (q1.exactOrientation(q2, q3, q4) == 0) std::cout << "Points qi are coplanar\n";
else std::cout << "Points qi are not coplanar\n";
```

```
Points pi are coplanar
Points qi are not coplanar
```

Precision level and predicates

```
// Now we switch to rational mode
TMesh::useRationals();

// We create other scaled copies of the pi's
Point3c r1 = p1 / 3;
Point3c r2 = p2 / 3;
Point3c r3 = p3 / 3;
Point3c r4 = p4 / 3;

// Coordinates in the ri's are exact scales of those in the pi's because we are
// representing them using rational numbers
if (r1.exactOrientation(r2, r3, r4) == 0) std::cout << "Points ri are coplanar\n";
else std::cout << "Points ri are not coplanar\n";

// Now we switch back to floating point mode
TMesh::useRationals(false);

// Even if we switched back to floating point, coordinates in the ri's are still
// rational numbers. That is why the following still reports coplanarity.
if (r1.exactOrientation(r2, r3, r4) == 0) std::cout << "Points ri are coplanar\n";
else std::cout << "Points ri are not coplanar\n";
```

```
Points pi are coplanar
Points qi are not coplanar
Points ri are coplanar
Points ri are coplanar
```

Behind the scenes

- Multiple technologies are integrated in Tmesh
  - Arithmetic Filtering
  - Floating Point Expansions
  - «a la Shewchuk» adaptive predicates
  - Intervals
  - Lazy evaluation (thread safe)
- Kernel may need to dynamically change FPU rounding mode

Predicate evaluation in TMesh

1. Check the subtype of all the parameters
2. IF all of them are double
    1. Use «a la Shewchuk» adaptive evaluation
3. ELSE
    1. Convert all parameters to intervals
    2. Evaluate predicate using intervals
    3. IF resulting interval contains zero
        1. Convert all parameters to rational numbers
        2. Evaluate predicate exactly

M. Attene (2017). **ImatiSTL - Fast and Reliable Mesh Processing with a Hybrid Kernel**. LNCS Transactions on Computational Science XXIX, Vol. 10220, pp. 86-96 (Springer)

Conclusions

To successfully implement a geometric algorithm

Carefully assess assumptions on input and numerical processes

- Invalid/unexpected input
  - Can I reasonably repair the input to make it valid?
  - Can I tolerate a small distortion everywhere?
  - What class of models is my algorithm designed for?
- Inaccurate process
  - If there are numerical errors, are they catastrophic?
  - If so, do I need intermediate constructions to determine the program flow?
  - If so, are these constructions sparse wrt the overall data to be processed?

Open Positions at IMATI-CNR

- **WHAT**: 3D Shape Design and Analysis for Digital Fabrication
- **WHO**: Myself and a vibrant research group
- **WHEN**: Application deadline: Sept 13. Start: ~ mid October. Duration: 1-2 years
- **WHY**: Because you love geometry processing and 3D printing
- **WHERE**: Genova (Intl. Airport C. Colombo)

More at: www.imati.cnr.it -> opportunities

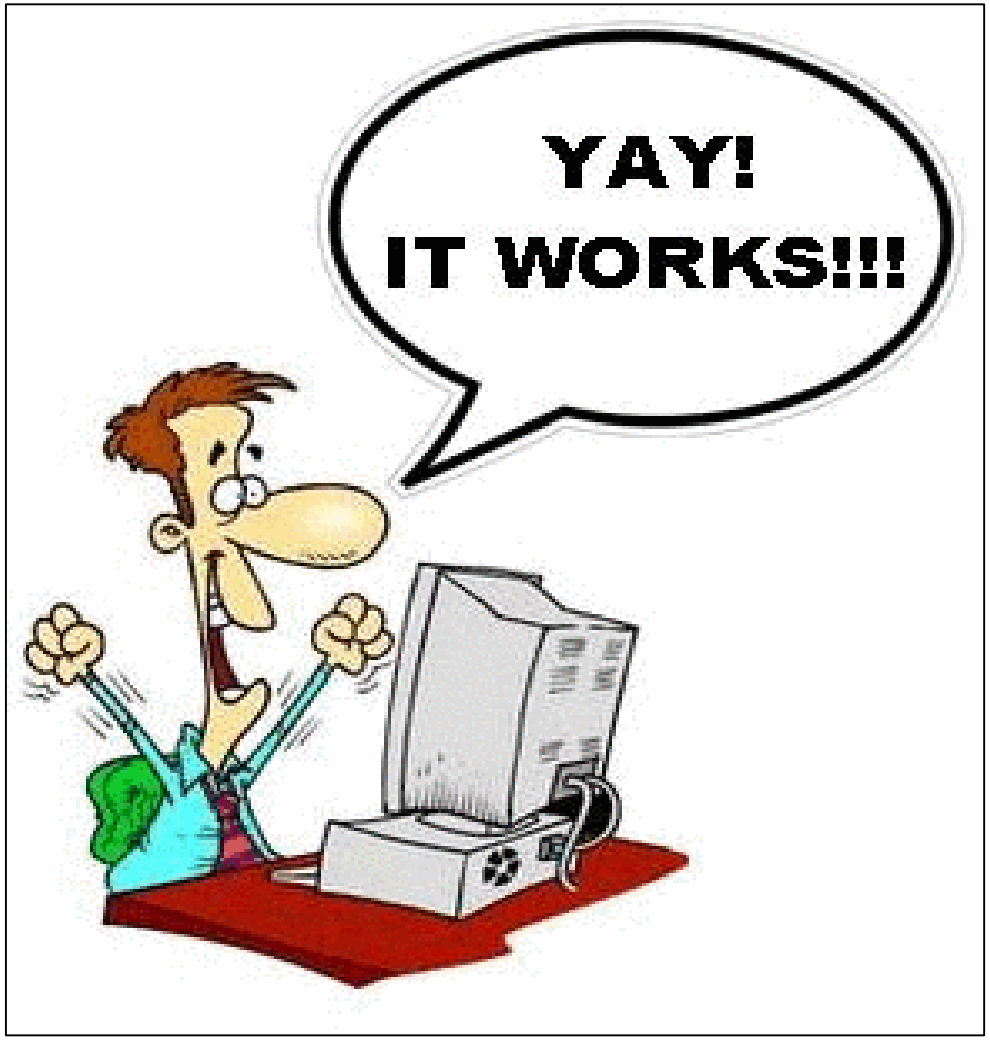
Thank you

Time for questions

More at: www.imati.cnr.it -> opportunities

Thank you

Time for questions



More at: www.imati.cnr.it -> opportunities