Arrangement of Planes

Daniel Lederer,
Institute of Theoretical Computer Science

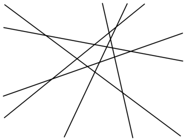
20. Mai 2022

General problem definition

- A finite set $\mathcal{H}$ of hyperplanes in $\mathbb{R}^d$ is given
- We want to compute the hyperplane arrangement $\mathcal{A}(\mathcal{H})$, a subdivision of the *d*-dimensional space induced by $\mathcal{H}$
- We are mainly interested in arrangements in 3-space, so **arrangement of planes**

Line arrangements in $\mathbb{R}^2$

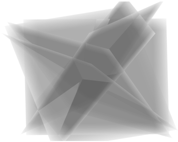
- A finite set $\mathcal{L}$ of lines in the plane is given
- $\mathcal{A}(\mathcal{L})$ is a subdivision of the plane induced by $\mathcal{L}$
- Representation of intersections: vertices, edges, faces
- We only deal with simple arrangements



A simple arrangement of 6 lines

Plane arrangements in $\mathbb{R}^3$

- A finite set $\mathcal{P}$ of planes in the space is given
- $\mathcal{A}(\mathcal{P})$ is a subdivision of the space induced by $\mathcal{P}$
- Representation of intersections: vertices, edges, faces, cells
- We only deal with simple arrangements



A simple arrangement of 6 planes

Combinatorial complexity of simple $\mathcal{A}(\mathcal{P})$

- Let n be the number of given planes
  - Maximum number of vertices = $\frac{n^3-3n^2+2n}{6} = \Theta(n^3)$
  - Maximum number of edges = $\frac{n^3-2n^2+n}{2} = \Theta(n^3)$
  - Maximum number of faces = $\frac{n^3-n^2+2n}{2} = \Theta(n^3)$
  - Maximum number of cells = $\frac{n^3+5n+6}{6} = \Theta(n^3)$
- Simple $\mathcal{A}(\mathcal{P}) \Rightarrow$ maximum number of components
- Overall complexity of $\Theta(n^3)$

Our task

- Developing an algorithm that computes $\mathcal{A}(\mathcal{P})$
- Creating a suitable representation of all arrangement components with all its relationship information
- Algorithm should be implementable and numerically robust
- Software solution using `C++`

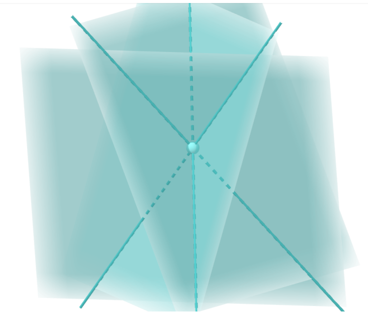
Data structure

Implemented representation of $\mathcal{A}(\mathcal{P})$

Data structure

- A simple $\mathcal{A}(\mathcal{P})$ with $n \geq 3$ necessarily results in 4 non-empty sets:
  - $\mathcal{V}$ contains all vertices
  - $\mathcal{E}$ contains all edges
  - $\mathcal{F}$ contains all faces
  - $\mathcal{C}$ contains all cells
- Each set has size $\Theta(n^3)$

Vertex

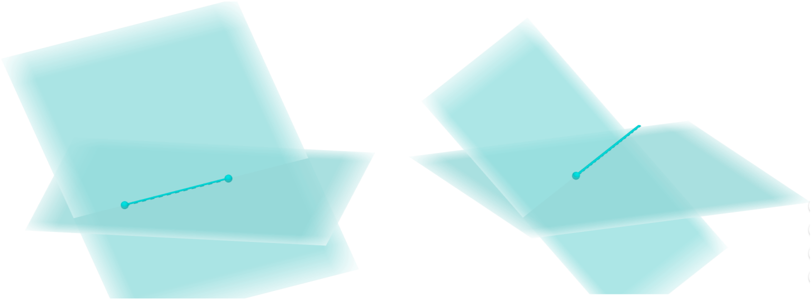


A vertex is created by the intersection of 3 lines, where a line is the intersection of two planes

Vertex

- A vertex holds the following information:
    - Point that holds the coordinates (x, y, z)
    - 6 adjacent edges
    - 12 adjacent faces
    - 8 adjacent cells
- For a vertex there is only a constant number of adjacent components (independent of n)

Edge



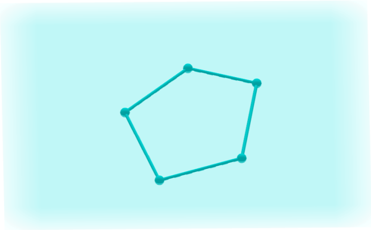
An edge is a portion on an intersection line, bounded by vertices. The edge can be bounded on both sides (line segment) or on one side (ray)

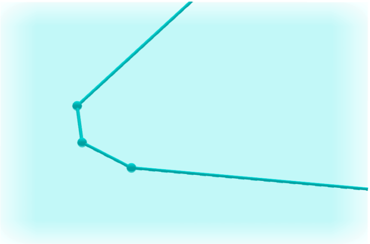
Edge

- An edge holds the following information:
  - Line on which the edge lies
  - Source and destination vertex (if it is bounded)
  - Ray that defines the direction
  - Whether it is bounded or not*
  - 4 adjacent faces
  - 4 adjacent cells
- For an edge there is only a constant number of adjacent components (independent of n)

*We say unbounded if it is not bounded on both sides

Face



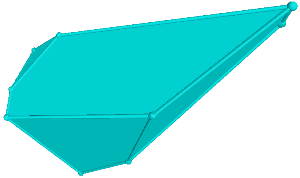


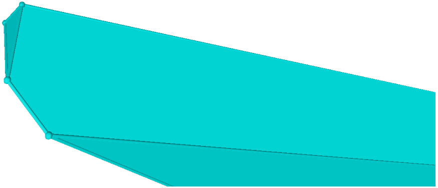
A face is a convex area on a plane, bordered by vertices and edges. The face can be bounded or not (then exactly 2 edges are unbounded)

Face

- A face holds the following information:
  - Plane on which the face lies
  - Set of bordered vertices
  - Set of bordered edges
  - Whether it is bounded or not
  - 2 adjacent cells
- For a face there is only a constant number of adjacent components (independent of n)

Cell





A cell is a convex region in the space, bordered by vertices, edges and faces.

The cell can be bounded or not

Cell

- A cell holds the following information:
  - Set of bordered vertices
  - Set of bordered edges
  - Set of bordered faces
  - Whether it is bounded or not
- A cell has no adjacencies to other (kind of) components

Algorithm

Constructing $\mathcal{A}(\mathcal{P})$

Algorithm

- Input: $\mathcal{P}$
- Output: $\mathcal{A}(\mathcal{P})$, so the filled data structure
- We follow a step-by-step approach:
  - 1. Compute vertices
  - 2. Compute edges
  - 3. Compute faces
  - 4. Compute cells
- Within these steps, relationships between components are created

Computing vertices

- We know, every plane intersects with every other plane
- A vertex is an intersection of 3 planes
- Consider each unique triple of planes and calculate the intersection point, which is the vertex
- Remember: This results in exactly $\frac{n^3-3n^2+2n}{6} = \Theta(n^3)$ vertices in $\mathcal{A}(\mathcal{P})$

Computing lines

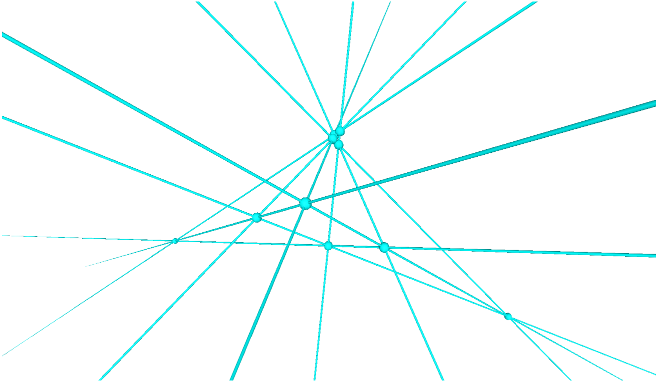
- Later on, we need to know which vertices lie on which line (for creating edges)
- 2 planes must intersect in a line
- Since there are 3 planes that intersect in one point, there must be three lines that intersect in a vertex
- Each vertex lies on 3 lines
- On each line there lie $n-2$ vertices
- In $\mathcal{A}(\mathcal{P})$ there are a total of $\frac{n(n-1)}{2} = \Theta(n^2)$ lines

Pseudocode

Algorithm 1 Computing vertices

```
1: for unique triple (p_a, p_b, p_c) in 𝒫 do
2:     v ← intersection(p_a, p_b, p_c)
3:     𝒱 insert v
4:     for unique tuple (p_r, p_s) in {p_a, p_b, p_c} do
5:         l ← intersection(p_r, p_s)      /* if not computed yet */
6:         l push v
7:         ℒ insert l
8:     end for
9: end for
```

What we have computed so far



In a simple $\mathcal{A}(\mathcal{P})$ with $n = 5$ there are 10 vertices and 10 lines

Computing edges

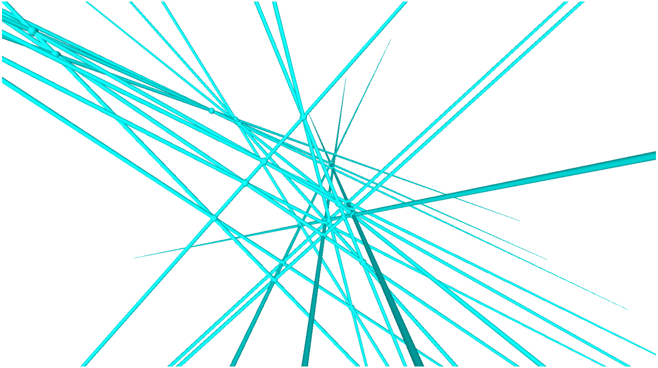
- We know $\mathcal{L}$ and which vertices lie on each $l \in \mathcal{L}$
- Sort all vertices on l from one side to the other side
- Subdivide each l into edges using the vertices
- On each l, $n - 3$ bounded and 2 unbounded edges are created
- Remember: This results in exactly $\frac{n^3 - 2n^2 + n}{2} = \Theta(n^3)$ edges in $\mathcal{A}(\mathcal{P})$

Pseudocode

Algorithm 2 Computing edges

```
for l in ℒ do
    V ← sort vertices in l
    e ← create(first v in V, ray)   /* ray points in the right outward direction */
    connect e with v and l; ℰ insert e
    for v in V do
        if v is last vertex then
            e ← create(v, opposite ray)
        else
            e ← create(v, next v)
        end if
        connect e with v (and next v) and l; ℰ insert e
    end for
end for
```

What we have computed so far



In a simple $\mathcal{A}(\mathcal{P})$ with $n = 7$ there are 35 vertices and 126 edges

Computing faces

- Each edge has 4 adjacent faces (2 on each plane)
- For each edge, where at least 1 adjacent face is still missing, create a new face there
- We have to distinguish on which plane a face is missing
- In a recursive way, gradually collect neighbouring edges that form a face
- Remember: This results in exactly $\frac{n^3-n^2+2n}{2} = \Theta(n^3)$ faces in $\mathcal{A}(\mathcal{P})$

Pseudocode

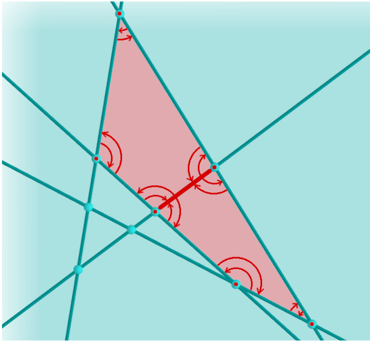
Algorithm 3 Computing faces

```
for e in ℰ do
    while number adjacent faces of e < 4 do
        f ← create()
        p ← suitable plane of e
        connect f with p
        o ← vertices orientation of e on p
        fillFace(f, e, o)
        ℱ insert f
    end while
end for
```

Collecting neighbouring edges

- Recursively inspect the next suitable edge for each vertex
- Suitable edge is the one to which the current edge encloses the smallest angle on the plane
- Calculate angles $\in (0, 2\pi)$ in a specific orientation
- For each side, consider the vertex in the opposite orientation ((counter)clockwise)
- Consider source and destination vertex also in opposite orientation

Choose vertex orientation



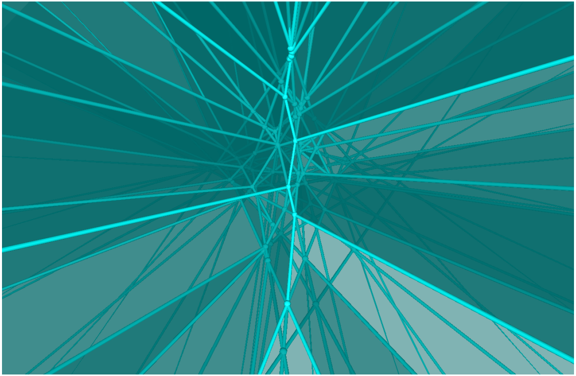
For the respective face on the plane, consider the vertices in the appropriate orientation regarding the plane's normal

Pseudocode

Algorithm 4 Fill face: face f, edge e, orientation o

```
connect f with e
vertices orientation of e on plane ← flip o    /* needed for the future */
update f boundedness
for vertex v of e do
    connect f with v
    e_n ← edge adjacent to v with smallest angle from e with respect to o
    if f is not connected with e_n then
        o_n ← vertices orientation of e_n on plane
        fillFace(f, e_n, o_n)
    end if
end for
```

What we have computed so far



In a simple $\mathcal{A}(\mathcal{P})$ with $n = 10$ there are 120 vertices, 405 edges and 460 faces

Computing cells

- Each face has 2 adjacent cells
- For each face, where at least 1 adjacent cell is still missing, create a new cell there
- In a recursive way, gradually collect neighbouring faces that form a cell
- Remember: This results in exactly $\frac{n^3+5n+6}{6} = \Theta(n^3)$ cells in $\mathcal{A}(\mathcal{P})$

Pseudocode

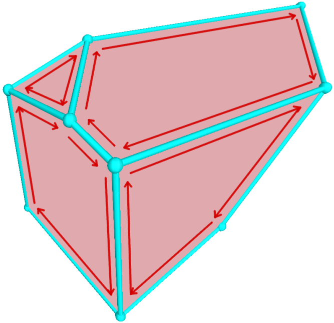
Algorithm 5 Computing cells

for f in $\mathcal{F}$ **do**
    while number adjacent cells of $f < 2$ **do**
        $c \leftarrow$ create()
        $o \leftarrow$ edges orientation on f
        fillCell(c, f, o)
        $\mathcal{C}$ insert c
    end while
end for

Collecting neighbouring faces

- Recursively inspect the next suitable face for each edge
- Suitable face is the one to which the current face encloses the smallest angle
- Calculate dihedral angles $\in (0, 2\pi)$ in a specific orientation
- For each side, consider the edges in the opposite orientation ((counter)clockwise)

Choose edges orientation



Consider the edges in the appropriate orientation

Pseudocode

Algorithm 6 Fill cell: cell *c*, face *f*, orientation *o*

```
connect c with f
connect c with vertices of f
edges orientation on f ← flip o    /* for next adjacent cell */
update c boundedness
for edge e of f do
    if c is not connected with e then
        connect c with e
        f_n ← face adjacent to e with smallest angle from f with respect to o
        if c is not connected with f_n then
            o_n ← edges orientation on f_n
            fillCell(c, f_n, o_n)
        end if
    end if
end for
```

We are done!

- The presented algorithm constructs the simple $\mathcal{A}(\mathcal{P})$
- Data structure is filled
- Representation of all occuring vertices, edges, faces and cells in the arrangement
- Relationship information between arrangement components
- Most operations are combinatorial only, except the vertex position and angle calculations

Complexity

Runtime and space consumption

Runtime

- Since we follow a step-by-step approach, the runtime can be determined quite easily
- Our algorithm takes advantage of the fact that there are only $\mathcal{O}(1)$ adjacency components
- Asymptotically, computing edges needs the most time, since we need to presort vertices
- Each face and cell consists of $\Omega(1)$ and $\mathcal{O}(n)$ components
- But overall, there are only $\Theta(n^3)$ components

Runtime

- Computing vertices: obviously $\Theta(n^3)$
- Computing edges: On each of the $\Theta(n^2)$ lines we need to sort $\Theta(n)$ vertices $\Rightarrow \Theta(n^3 \log n)$
- Computing faces: Iterating over $\Theta(n^3)$ vertices and edges $\Rightarrow \Theta(n^3)$
- Computing cells: Iterating over $\Theta(n^3)$ vertices, edges and faces $\Rightarrow \Theta(n^3)$

$$\Rightarrow \text{Overall runtime of } \Theta(n^3 \log n)$$

Space consumption

- For each vertex in $\mathcal{V}$, $\mathcal{O}(1)$ memory is required
- For each edge in $\mathcal{E}$, $\mathcal{O}(1)$ memory is required
- For each face in $\mathcal{F}$, we store $\Omega(1)$ and $\mathcal{O}(n)$ vertices and edges
- For each cell in $\mathcal{C}$, we store $\Omega(1)$ and $\mathcal{O}(n)$ vertices, edges and faces
- Nevertheless, there are only $\Theta(n^3)$ components

$\Rightarrow$ Overall space consumption of $\Theta(n^3)$

Conclusion

Summary

- Our algorithm constructs the simple $\mathcal{A}(\mathcal{P})$ of n planes in $\mathcal{P}$ in $\Theta(n^3 \log n)$ time and $\Theta(n^3)$ space
- Asymptotically, we do not need more space than $\mathcal{A}(\mathcal{P})$ itself
- Algorithm and data structure are easy to implement and fast
- Strategy is numerically quite robust, since we do not need any additional intersection checks (on which a decision depends)

Time for demo!