# WIFI connection

**DO NOT SIGN INTO THE GUEST NETWORK WITHOUT CREDENTIALS!!!**

- Connect to the network: ‘Harvard University’
- If you are not redirected to the sign-in page, go to http://getonline.harvard.edu
- Click “I am a guest”
- Click “Log in with guest credentials”
    - Guest Username: foss4g2017@gmail.com
    - Guest Password: 7RFQU3rm

# Intro to Spatial Algorithms

FOSS4G 2017

# Chris Barnett

Sr. Geospatial Analyst at Tufts University,
dev Open Geoportal, LOC

## Introductions

## basic outline

- some preliminaries about terminology and data structures
- framework for workshop exercises
- point in polygon: overview, algorithm, exercise, review
- convex hull: overview, algorithm, exercise, review
- triangulations: overview, algorithm, exercise, review
- look at some more advanced algorithms

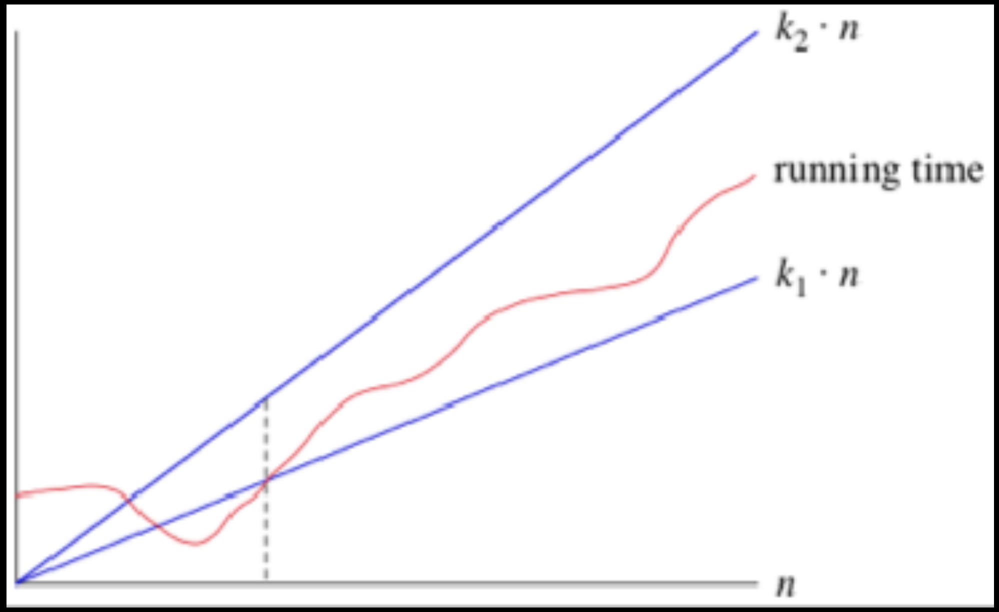
## Preliminaries

## Big-O notation

The basic idea is to express the asymptotic running time of an algorithm as a function of the number of inputs. Theta notation is a measure of rate of growth in running time, so we ignore leading constants and secondary terms. Big-O notation is used to show the upper bound, Big-Ω is used to designate the lower bound, and Big-Θ denotes both upper and lower bounds.

# Big-O Notation

So, if we say that an algorithm runs in linear time, we mean that the graph of number of inputs vs. running time can be bounded by a line on the plane. Note that the line could be very steep or very shallow or be offset by a huge constant value in the y direction.



# Big-O Notation

In practice algorithms with large leading constants or secondary terms may take longer to run than algorithms that are “slower” according to theta notation, depending on your inputs.

If speed is a factor for you, it’s important to know where the running time curves intersect for your inputs!

One approach is to run multiple algorithms simultaneously and take the first result (terminate the others).

# Big-O Notation

1. $\Theta\ (1)$
2. $\Theta(\log n)$
3. $\Theta(n)$
4. $\Theta(n\log n)$
5. $\Theta(n^2)$
6. $\Theta(n^2\log n)$
7. $\Theta(n^3)$
8. $\Theta(2^n)$

Note: exponential functions grow faster than any polynomial function

# geojson

- our exercises will use the geojson format
- this section borrows heavily from Tom MacWright
- https://macwright.org/2015/03/23/geojson-second-bite.html

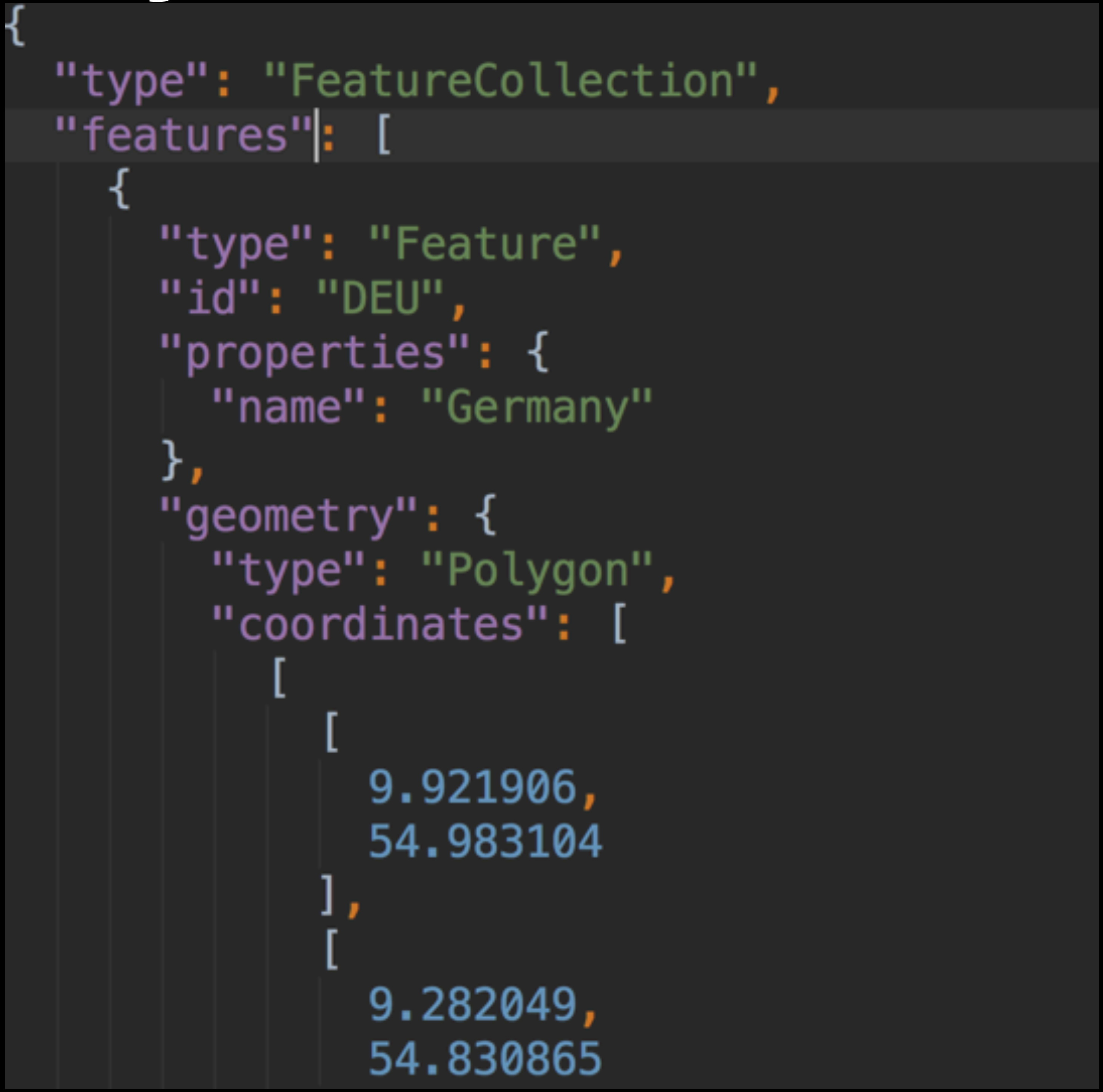
# geojson

Projection: WGS84 (EPSG:4326)

Projections are not well supported in GeoJSON. If your calculations require greater precision, use a different format. Also, you will have a lot to think about and explore in determining what projection is appropriate for your algorithm.

# geojson

This is a FeatureCollection with one feature, with geometry of type Polygon. The polygon is an array of points. The starting and ending points in the array are the same to indicate a closed geometry. The geometry is represented as an array of arrays of arrays to allow for holes.



```
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "id": "DEU",
      "properties": {
        "name": "Germany"
      },
      "geometry": {
        "type": "Polygon",
        "coordinates": [
          [
            [
              9.921906,
              54.983104
            ],
            [
              9.282049,
              54.830865
```

## geojson

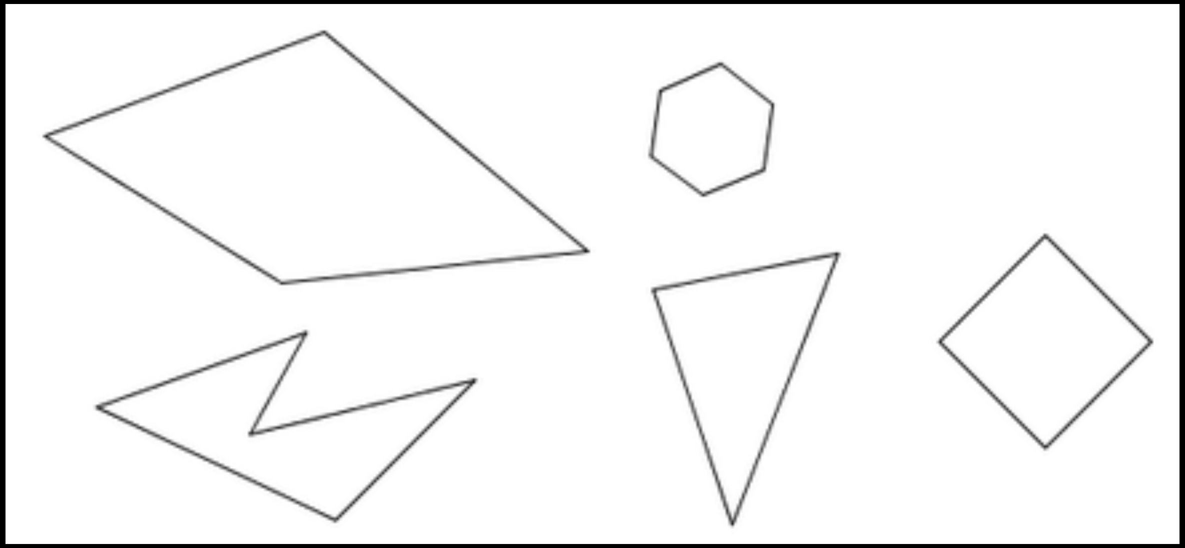
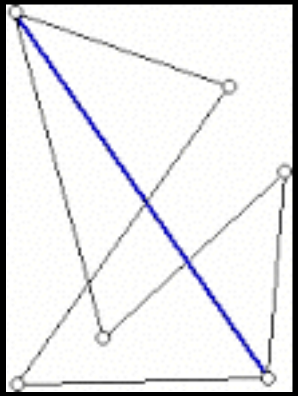
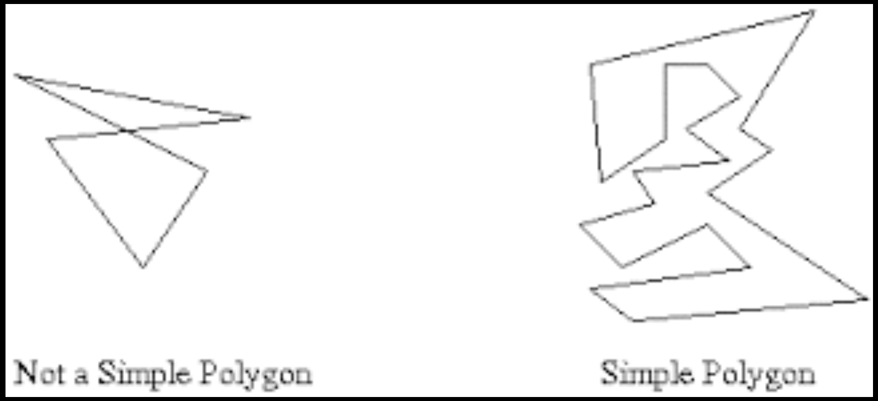
- all of our data will consist of FeatureCollection objects
- some will have single features, others will have multiple
- geometry types: Polygon, Point
- the ‘coordinate’ value is not nested for type ‘Point’

## geojson

- Winding is the term used to describe the direction the segments in a polygon are ordered (clockwise or counter-clockwise).
- Winding is important because many common spatial algorithms depend on a specific winding direction
- the right hand rule is recommended for geojson (CCW outside rings, CW inside rings), but not enforced.

# geojson

- simple polygons!
- non-simple

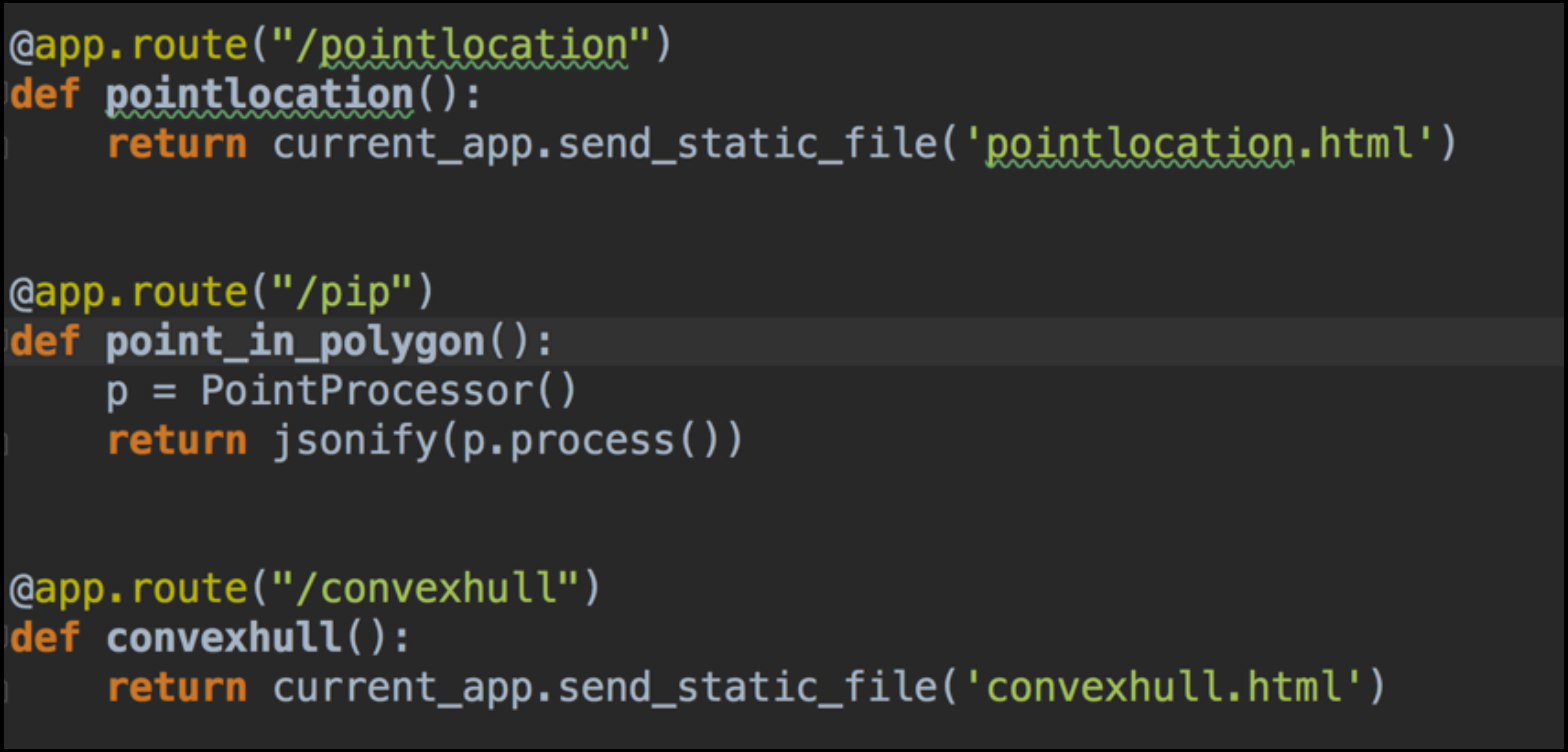
## code

- copy the code from the git repository, or from a USB drive

## Flask

- Flask is a python based web framework
- We will be using Flask to serve some simple Leaflet maps to display our exercise results

# Flask



```python
@app.route("/pointlocation")
def pointlocation():
    return current_app.send_static_file('pointlocation.html')


@app.route("/pip")
def point_in_polygon():
    p = PointProcessor()
    return jsonify(p.process())


@app.route("/convexhull")
def convexhull():
    return current_app.send_static_file('convexhull.html')
```

## Flask

- to install Flask, type ‘pip install flask’. If you use virtual environments, activate your ve first.
- to run Flask
  - navigate to the root of the workshop code folder at a terminal
  - type ‘FLASK_APP=app.py flask run’
- the server will be available in the browser at ‘http://localhost:5000'
- stop and restart Flask after you make changes

## Point in polygon

is a point within a simple 2D polygon?

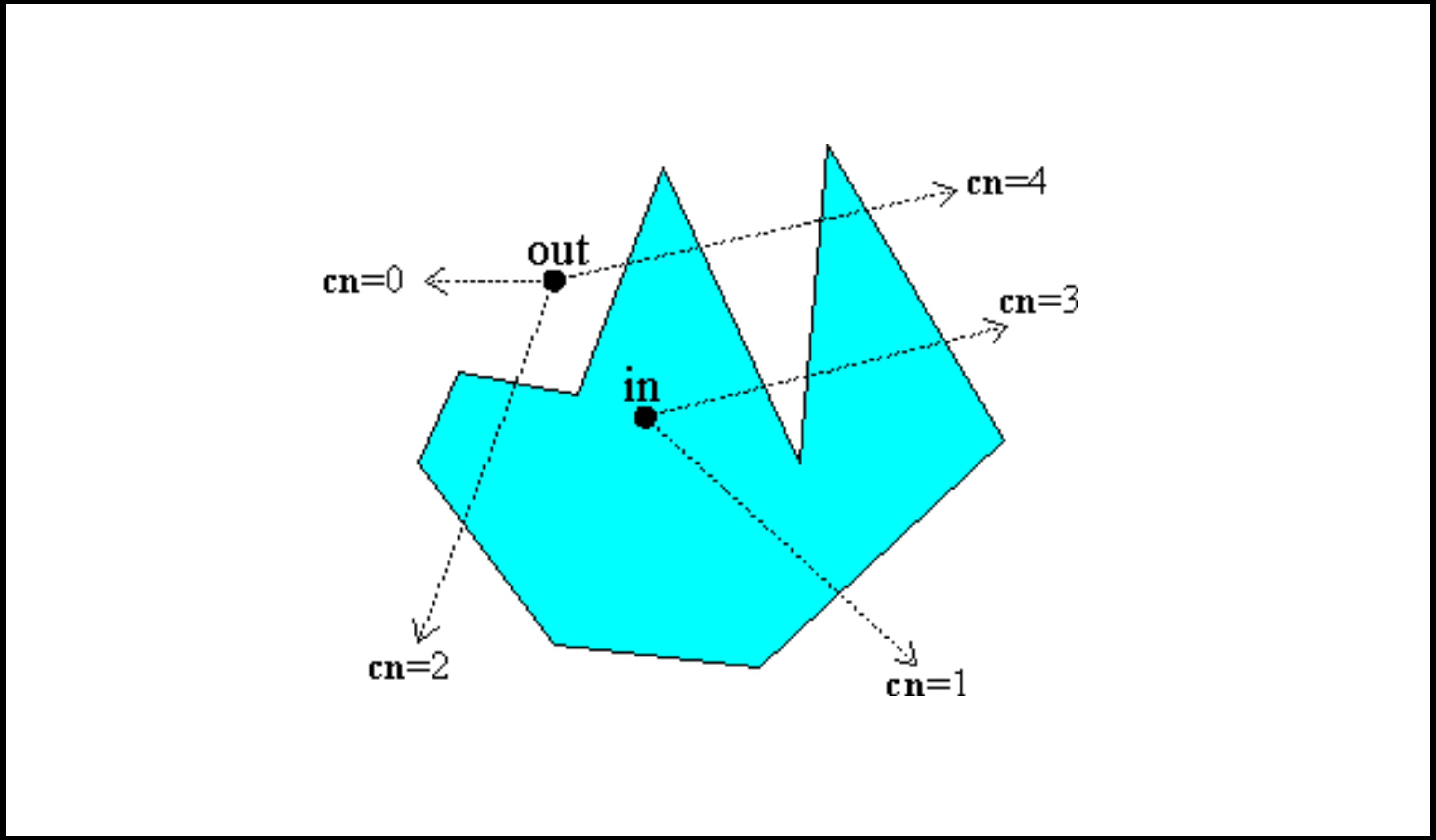
## Crossing Number

# Crossing Number

- A direct consequence of the Jordan Curve Theorem
- A simple closed curve divides the 2D plane into exactly 2 connected components: a bounded "inside" one and an unbounded "outside" one.
- The curve must be simple (without self intersections), otherwise there can be more than 2 components and then there is no guarantee that crossing a boundary changes the in-out parity.

# Crossing Number

- Count the number of times a ray starting from a point crosses the polygon boundary edges. An odd number of crossings results if the point is inside the polygon.



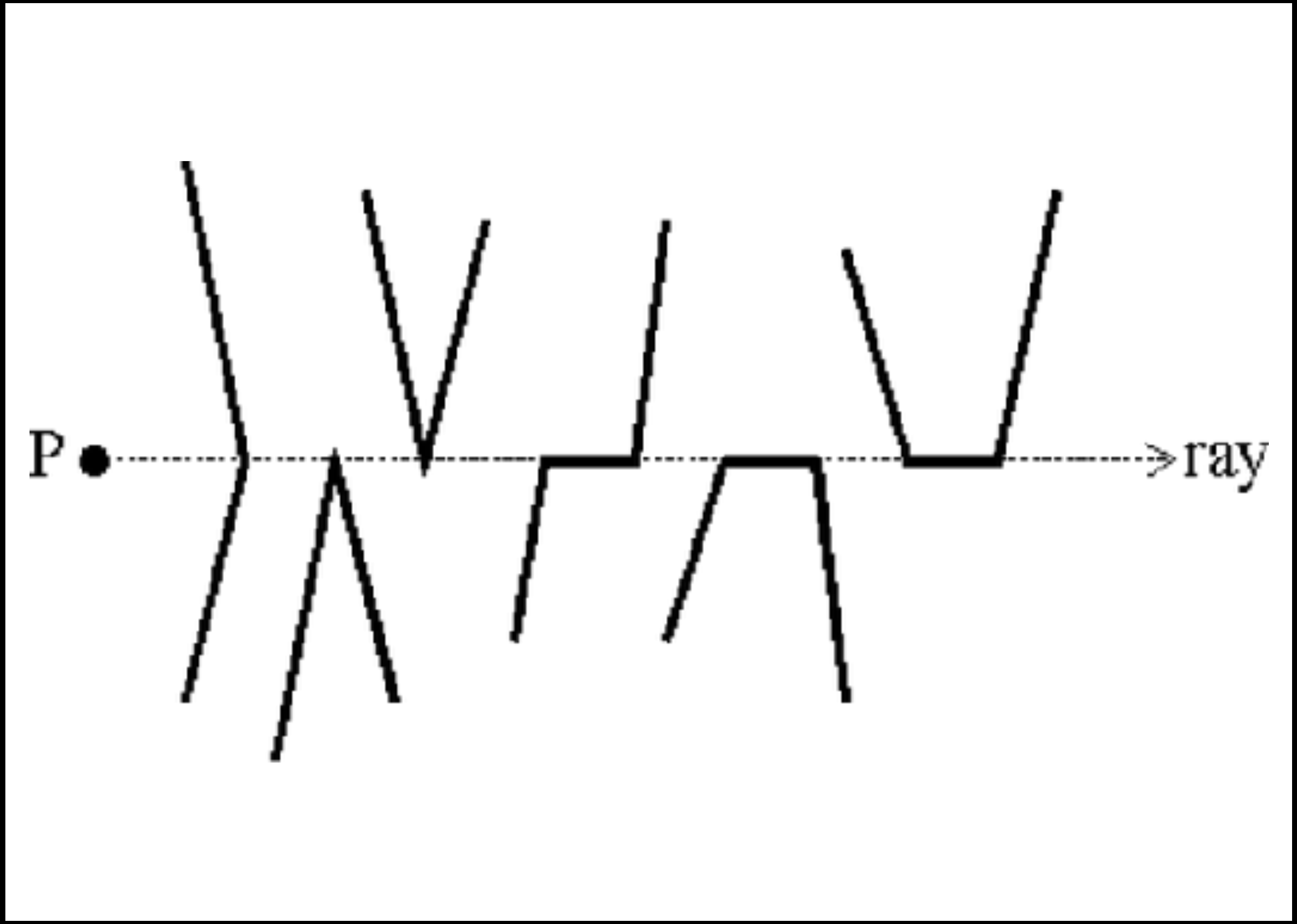
## Crossing Number

**Basic algorithm**

- draw a horizontal ray from the test point to the “right”
- iterate over polygon segments, counting ray crossings
- a ‘ray’ is a “line with one end”; a defined starting point, but continues to infinity

# Crossing Number

- Special cases:
- vertex intersections
    - take care to avoid double counting
- horizontal segments (just ignore)
- point on the boundary



# Crossing Number

Edge Crossing Rules

1. an upward edge includes its starting endpoint, and excludes its final endpoint

2. a downward edge excludes its starting endpoint, and includes its final endpoint

3. horizontal edges are excluded

4. the edge-ray intersection point must be strictly right of the test point

# Crossing Number

- exercise: implement main components of the crossing number algorithm (process_points_cn.py and crossing_number.py)
- end result: show points on a map, indicating whether or not they are in a given country
- goto: ‘http://localhost:5000/pointlocation' to see results

# Crossing Number

- For ‘x_crossing’, you need to know how to get the equation of a line given 2 points:
  - slope = (y2 - y1)/(x2 - x1)
  - substitute slope for ‘m’ in:
    - y - y1 = m(x - x1)

# Crossing Number

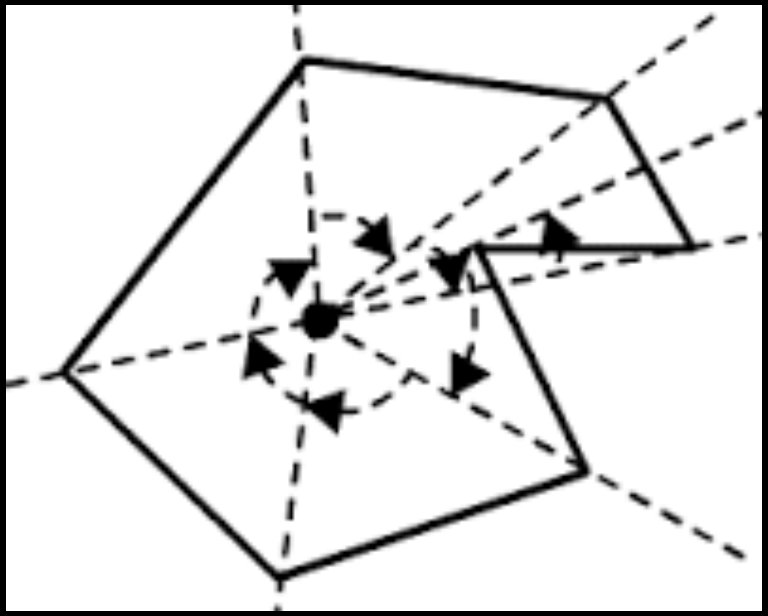
**Limitations**

- Must be a simple closed polygon (no self-intersections)

## Winding Number

# Winding Number

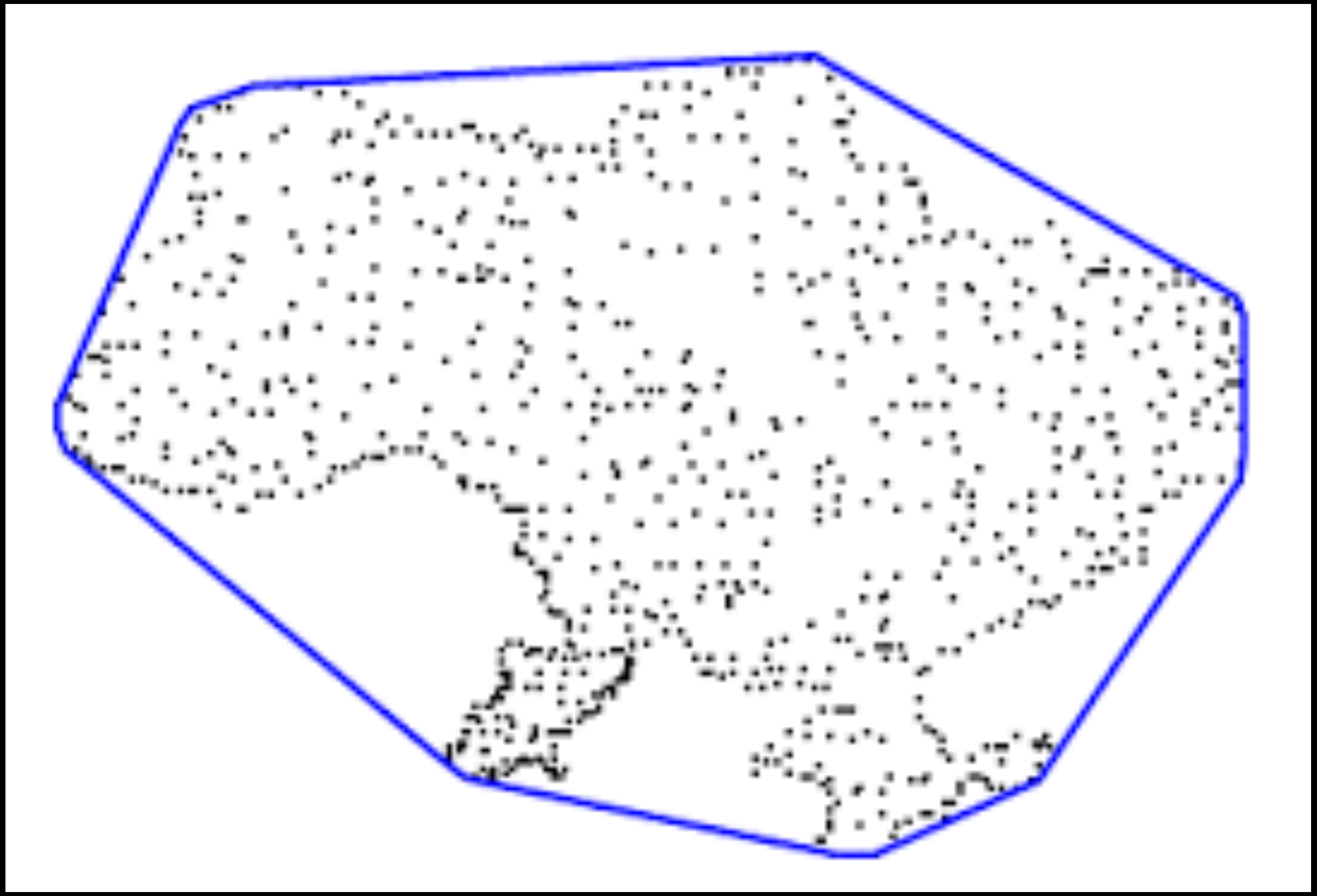
- Another algorithm for determining whether a point is in a polygon.
- add up all the angles subtended by each edge of the polygon (signs matter!)
- a winding number of 0 means that the point is outside the polygon
- handles non-simple polygons

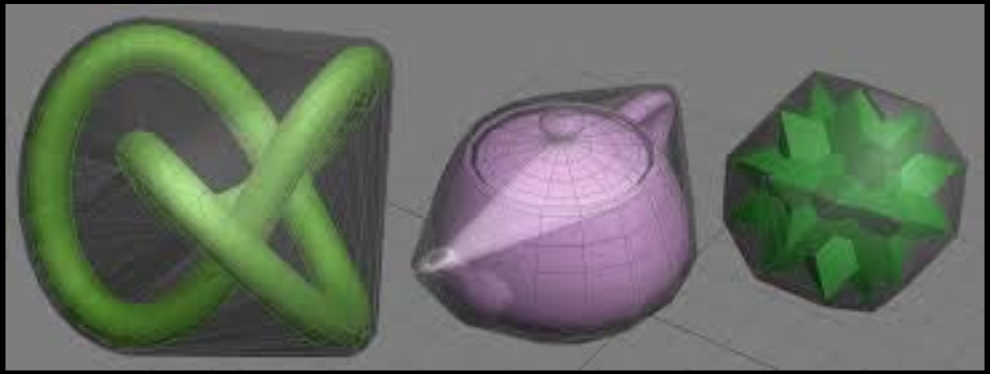


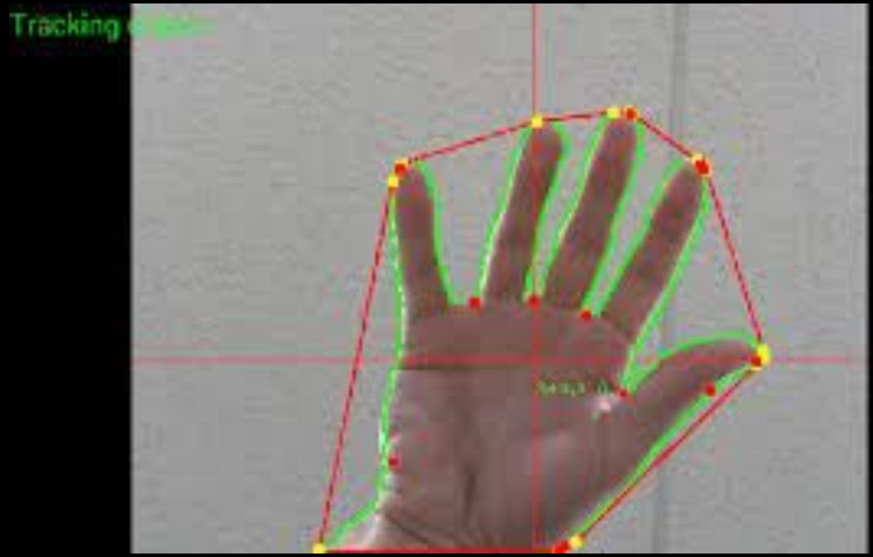
# Convex Hulls

- The smallest convex polygon that encloses a polygon or point set
- Applications include pattern recognition, image processing, statistics, geographic information system, game theory, construction of phase diagrams, and static code analysis by abstract interpretation. It also serves as a tool, a building block for a number of other computational-geometric algorithms such as the rotating calipers method for computing the width and diameter of a point set.
- brute force algorithms give you O($n^4$) run time!

# Convex Hulls

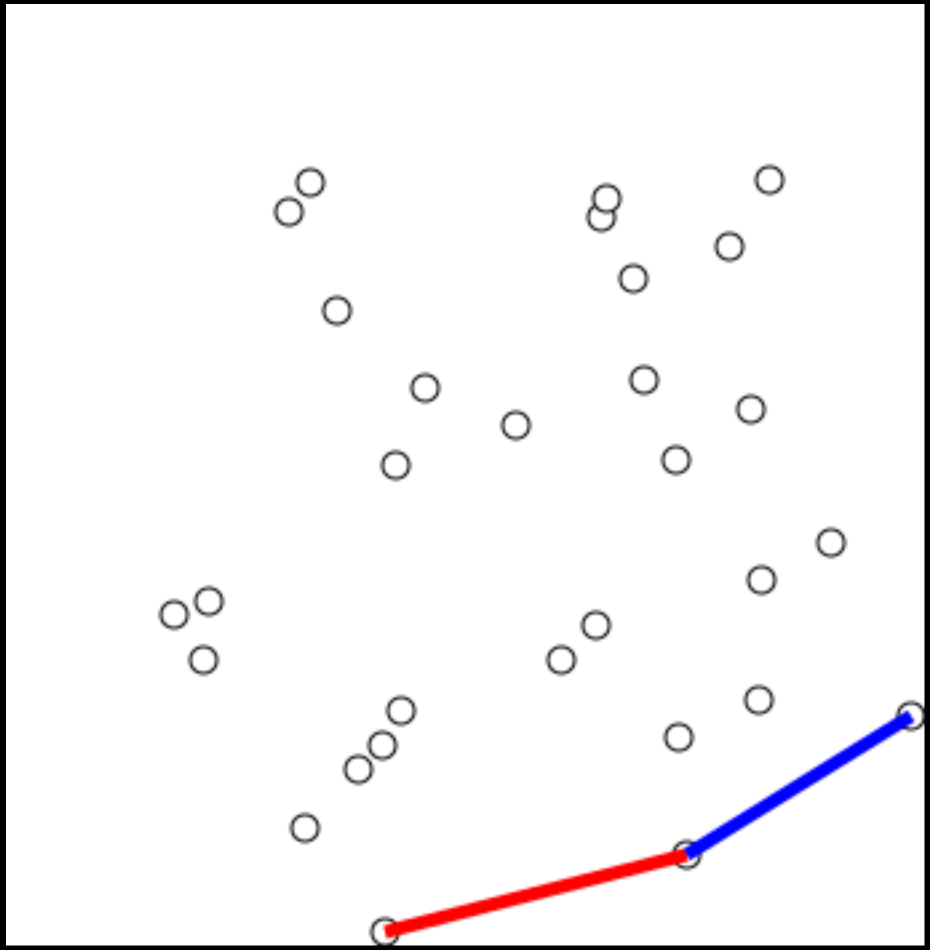






# Graham Scan

- O(n log n) algorithm for computing the convex hull of a 2d point set



# Graham Scan

- Need to start with a point on the convex hull. What points can we guarantee are on the hull?

## Graham Scan

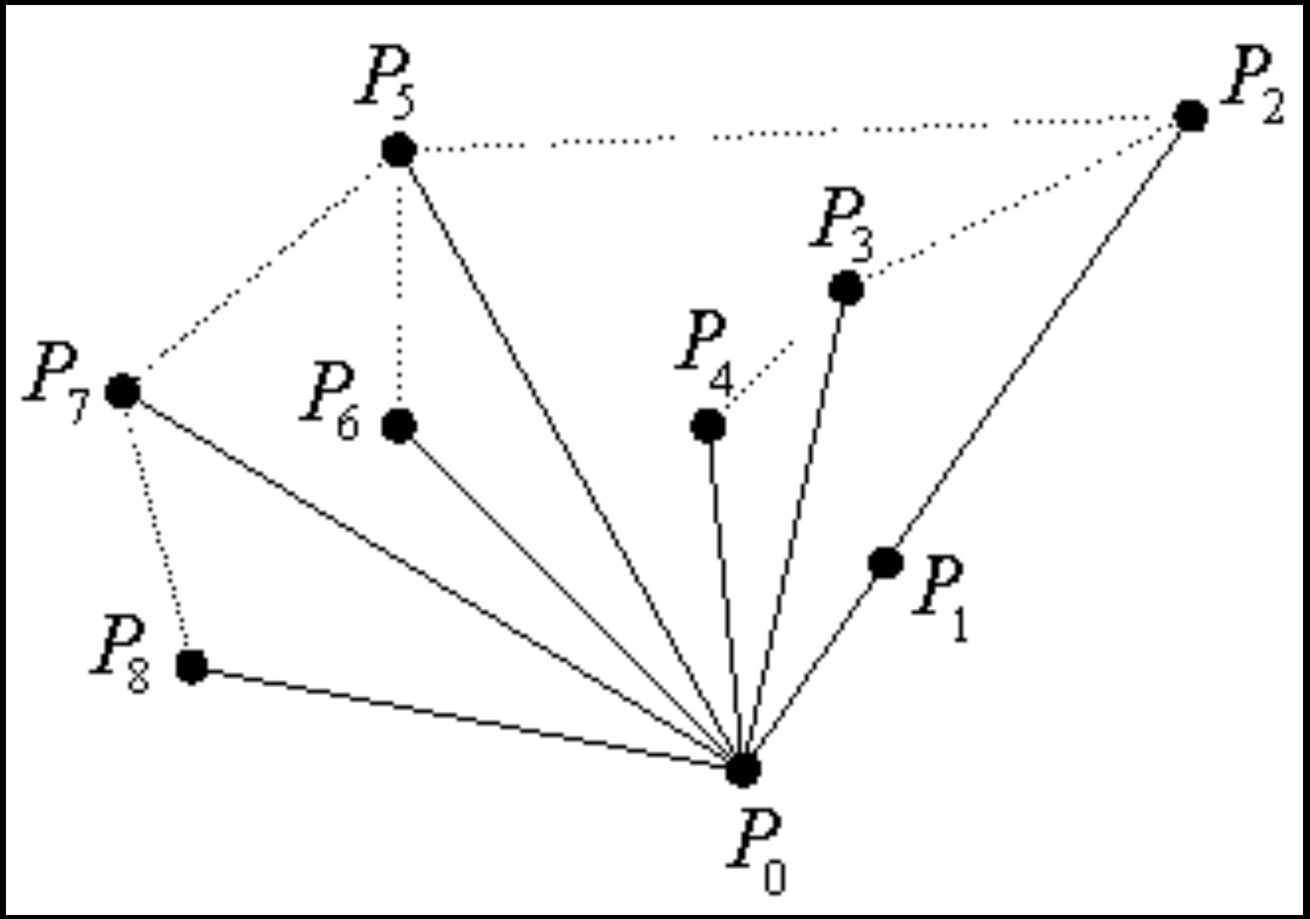
- Need to start with a point on the convex hull. What points can we guarantee are on the hull?
    - Find an extreme point (min x, max x, min y, or max y)
    - how fast can we get this?

## Graham Scan

- Need to start with a point on the convex hull. What points can we guarantee are on the hull?
  - Find an extreme point (min x, max x, min y, or max y)
  - how fast can we get this?
  - linear time

# Graham Scan

- Next, sort the points radially around the first point.
- how fast can we do this?



# Graham Scan

- Next, sort the points radially around the first point.
- how fast can we do this?
  - O(n log n). This is the lower bounds for sorting. This is also the only non linear time step. however…

# Graham Scan

- Next, sort the points radially around the first point.
- how fast can we do this?
  - O(n log n). This is the lower bounds for sorting. This is also the only non linear time step. however…
  - we might be able to sort in linear time if we use fixed precision and alter the algorithm to use x-coordinate sort (using radix sort, splitting into top and bottom hulls that are merged)

# Graham Scan

- Take the first 3 sorted points. This will be a correct convex hull of three points.
- Find the rest of the hull incrementally
    - Add the next point from the sorted list.
    - Check the last 3 points to see if they make a CCW turn (we'll see how to do this later)
    - If they do not, remove previous points until a CCW turn is restored
    - Iterate over the entire point list

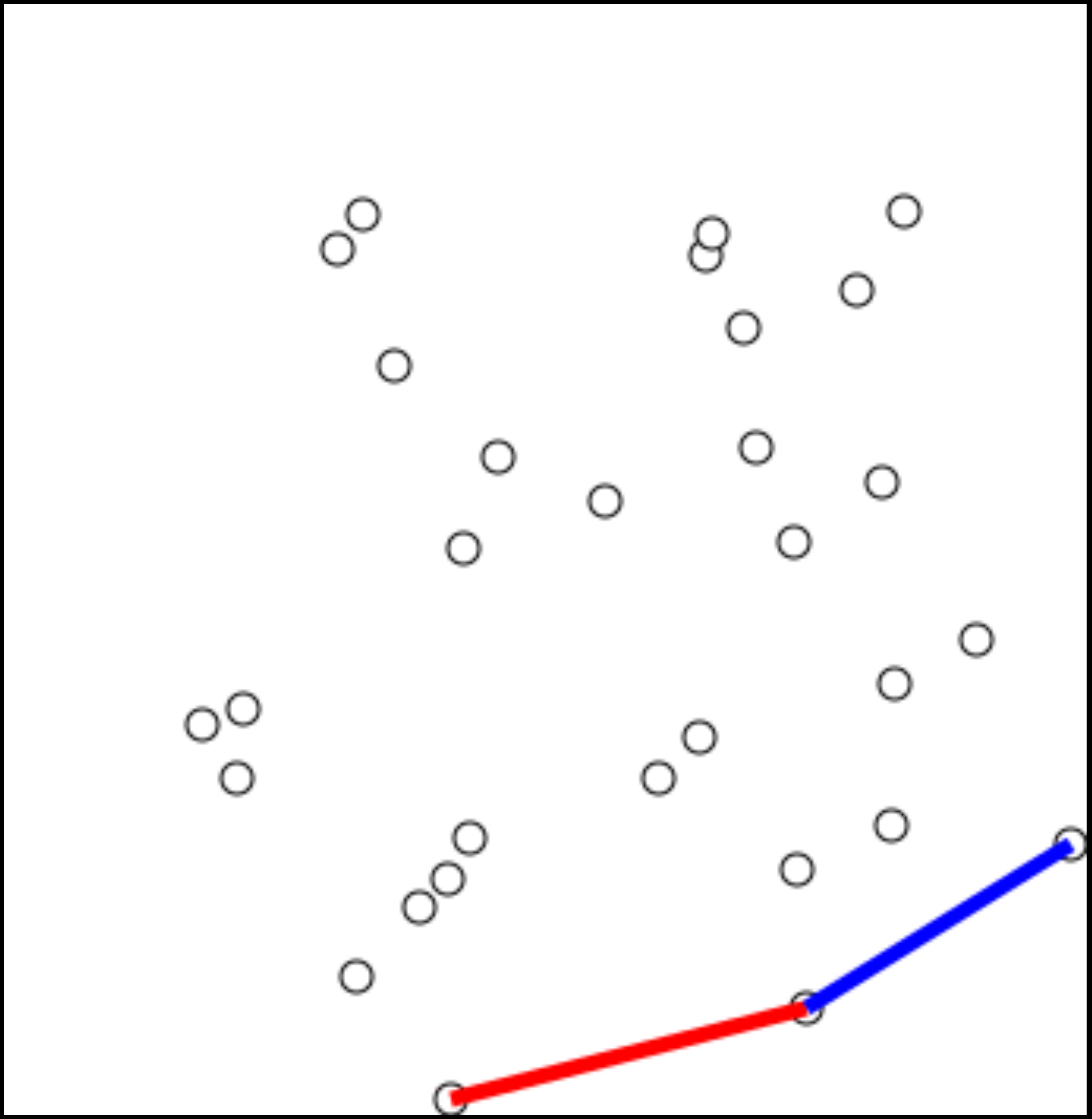
## Graham Scan

- How fast can we iterate over the point list to complete the convex hull?

## Graham Scan

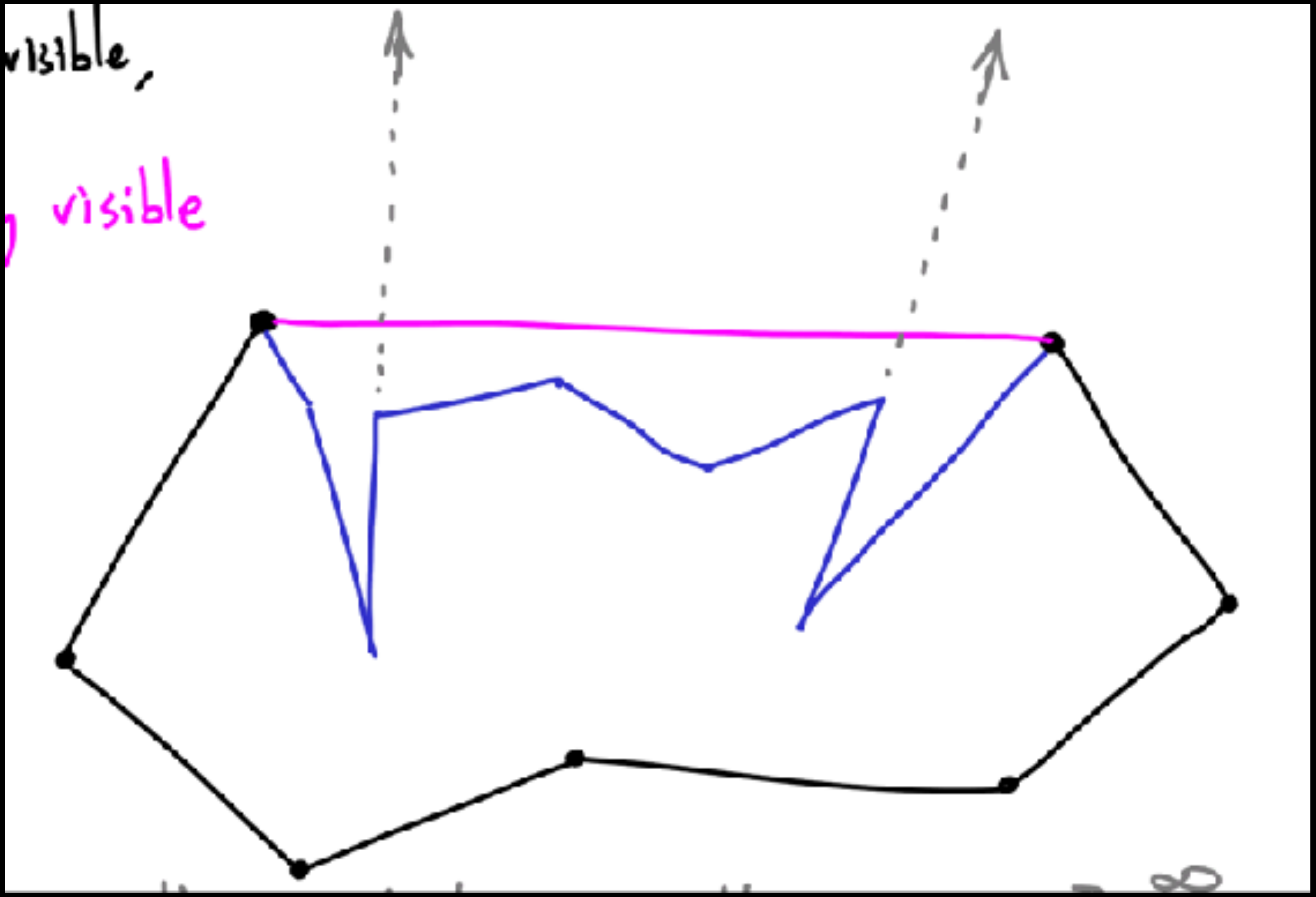
- How fast can we iterate over the point list to complete the convex hull?
    - Once a point is removed, it is never considered again, so 2n, which is O(n). Sorting is the bottleneck for the algorithm.

# Graham Scan



# Graham Scan

- a limitation: only applies to Weakly Externally Visible (WEV) polygons.



## Melkman

- Considered the best CH algorithm for simple polygons. (a poly line suffices)
- time is O(n)
- An online algorithm. At any step a correct convex hull for the point set is maintained. Also means that the initial point need not be on the final hull.
- Logic is very similar to the Graham scan.

# Melkman

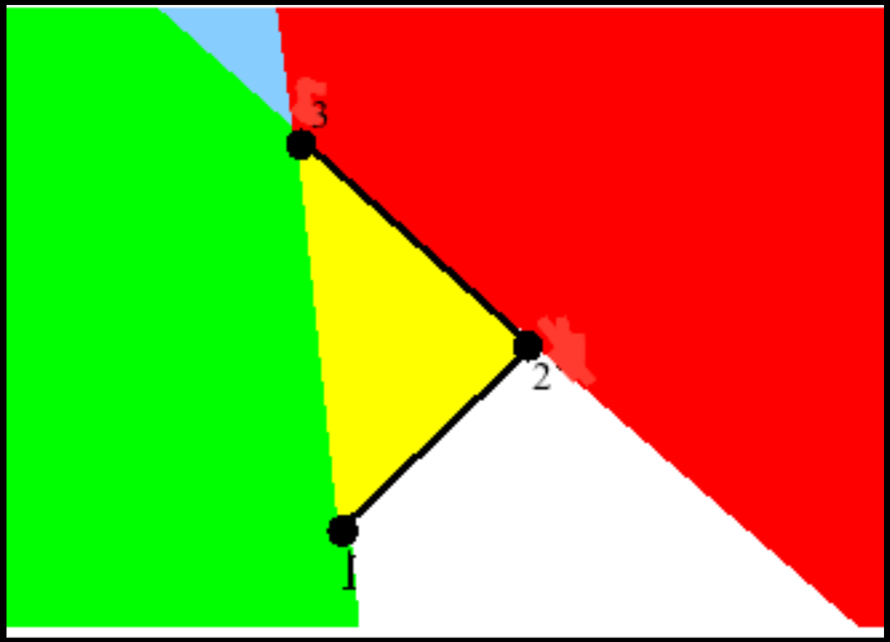
- Points are stored in a deque (double ended queue). This means that points can be added or removed from each end.

- To start, load the first 3 points into the deque:

    - (deque) : (bottom) 3-1-2-3 (top)

# Melkman

Notice that if we take the last three vertices of the bottom in the order 2,1,3 we get a right turn. If we take the last three vertices from the top, in the order 1,2,3 we get a left turn. This is a property that we wish to maintain: as we read the deque from bottom to top, we get the hull in CCW order, and as we read from top to bottom we get a CW order.

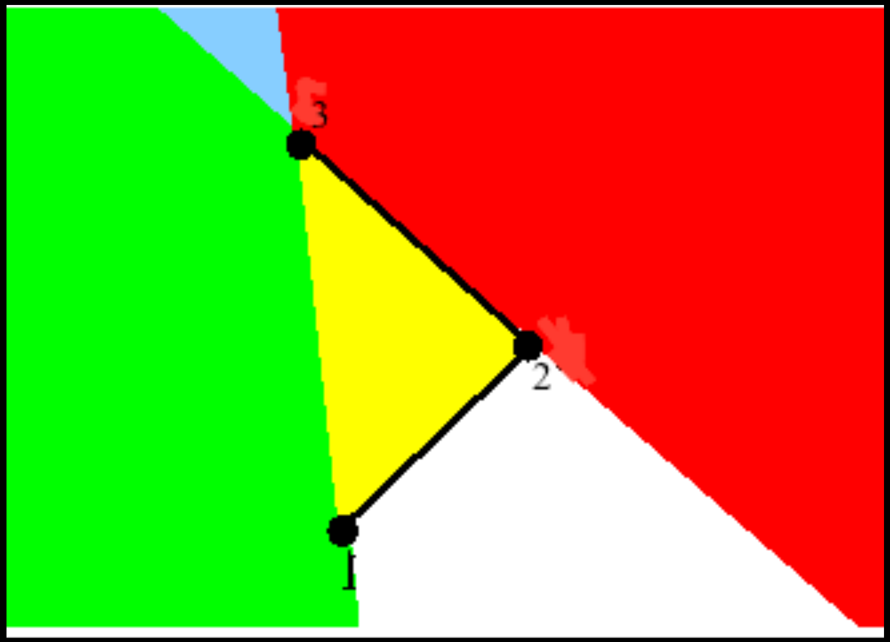


# Melkman

1. The next vertex, V, could be in the red/green/blue/yellow regions.

2. If V is in the yellow region, ignore it and all following vertices until one emerges into the other regions.

3. If V is not yellow, we must add it to the deque on both sides, because it will be on the current hull.



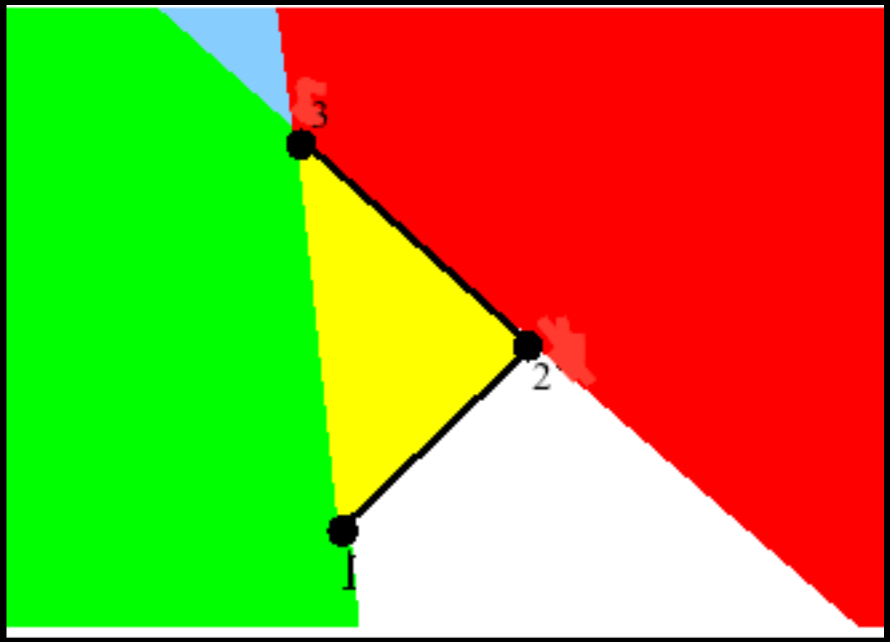
# Melkman

1. If V is in the red region, then 2,3,V form a right turn. Delete vertices from the top of the deque (i.e. 3, then maybe 2, etc), until a left turn is formed by the last 3 vertices.

2. If V is in the green region, then 1,3,V form a left turn. Delete vertices from the bottom of the deque (i.e. 3, then maybe 1, etc), until a right turn is formed by the last 3 vertices (symmetric to 1)

3. If blue, follow the instructions for both red and green.

4. iterate

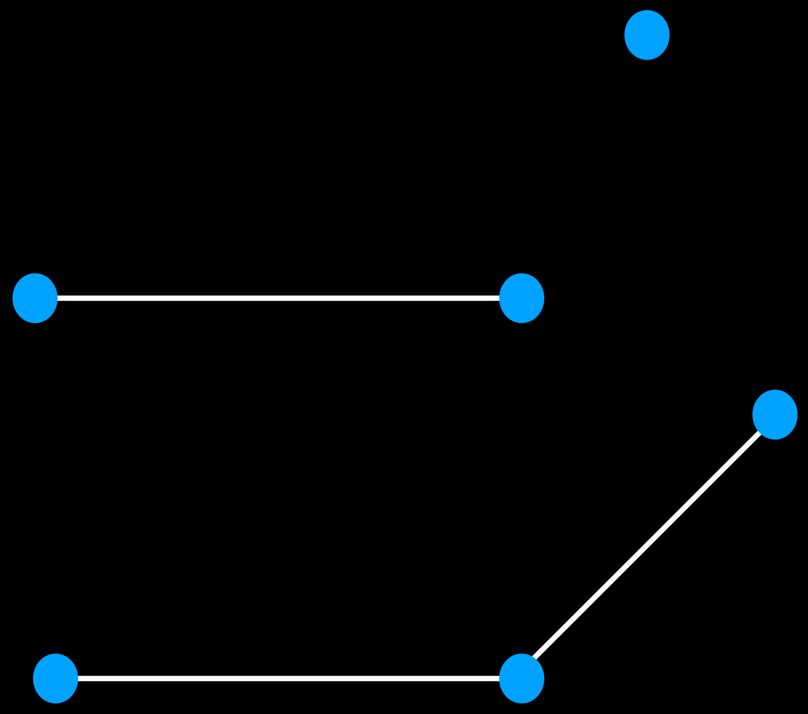


## Melkman

- Interactive visualization of Melkman:
- https://maxgoldste.in/melkman/

# Cross Product

- the cross product is used to determine the sign of the acute angle defined by three points
- so, we can use it to determine if a point is above or below a line, or if 3 points turn left (CCW) or right (CW)
- $P = (x2 - x1)(y3 - y1) - (y2 - y1)(x3 - x1)$
- $P=0$ : colinear
- $P > 0$ : left turn
- $P < 0$ : right turn
- use ‘test_ccw’ in ‘utilities.py’

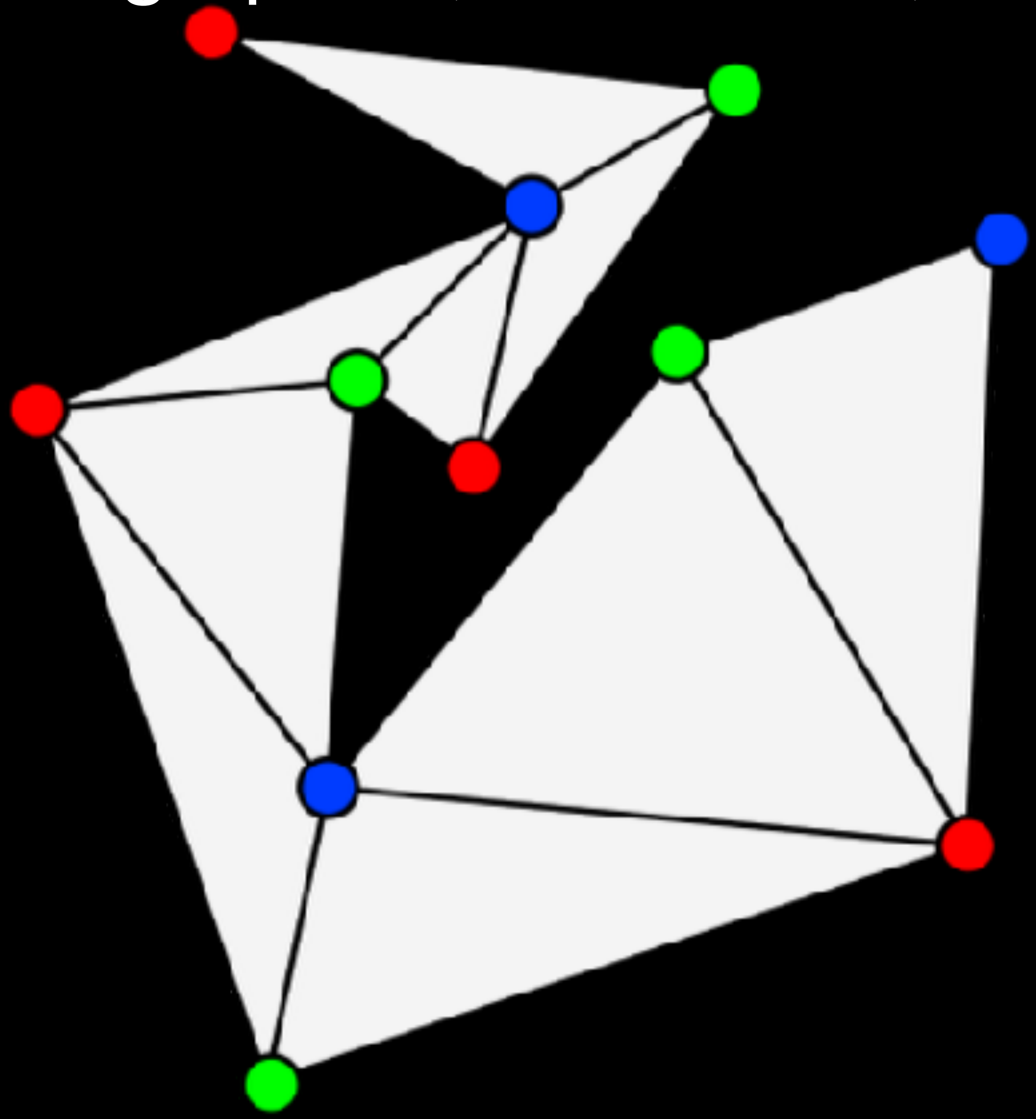


# Melkman

- exercise: complete implementation of Melkman
- see: ‘melkman.py’
- goto: ‘http://localhost:5000/convexhull' to see results

# Polygon Triangulation

- Decomposition of polygons into triangles
- used for interpolation, computer graphics, calculation, etc.



# Polygon Triangulation

**Meisters Two-ear theorem**

- The two ears theorem states that every simple polygon (with more than three vertices) has at least two ears.
- Guarantees that there is a complete triangulation for any polygon.

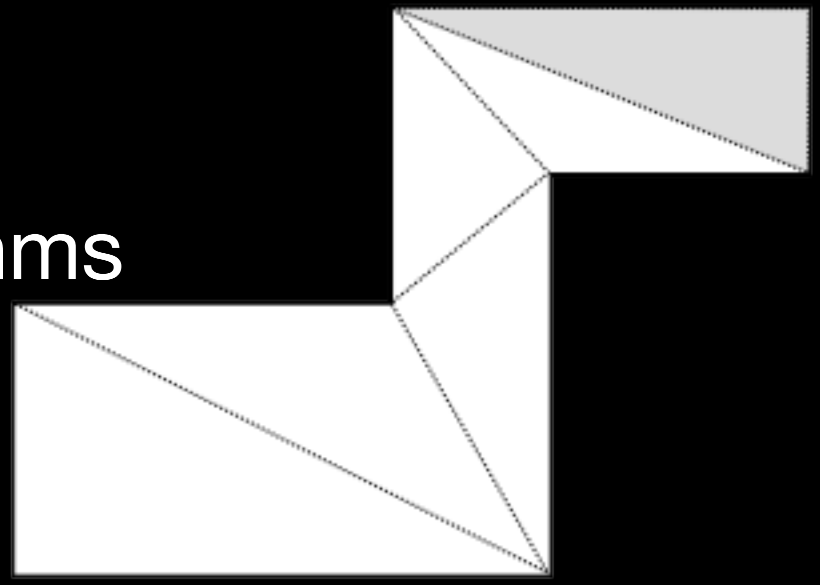
# Polygon Triangulation

- Triangulation by ear-clipping (recursively finding ears) takes O($n^2$)

- decomposing first into monotone polygons, then triangulating can be done in O(n log n)

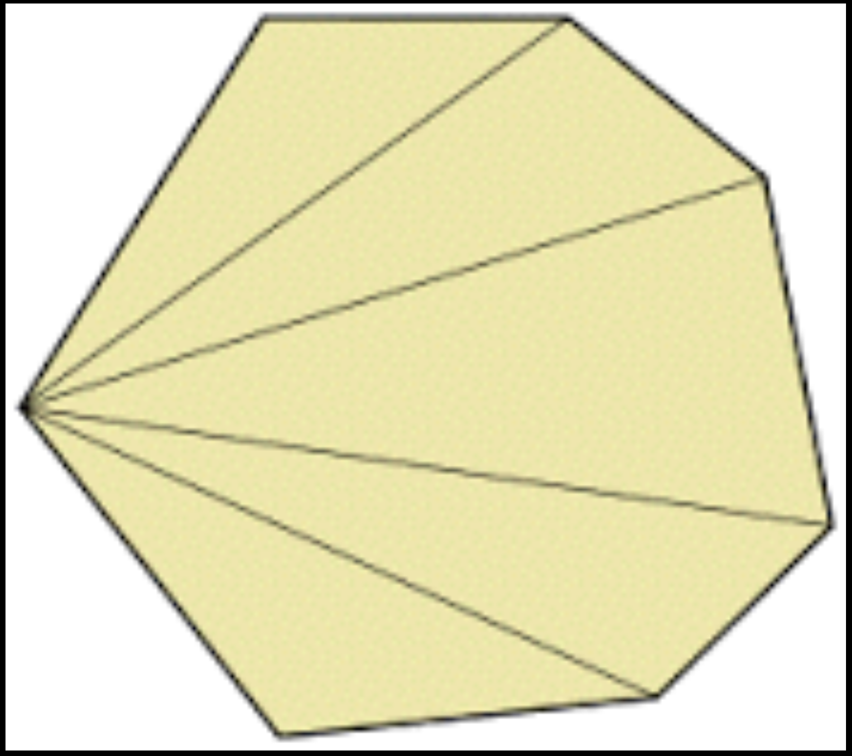
- Chazelle wrote a paper in 1991 showing that any polygon can be triangulated in O(n), but it is very complicated and not well understood.

- There are linear time randomized algorithms



# Polygon Triangulation

- A special case: triangulation of convex polygons.
- A fan triangulation can be done in O(n)
- also monotone polygons
- also star shaped polygons



## Polygon Triangulation

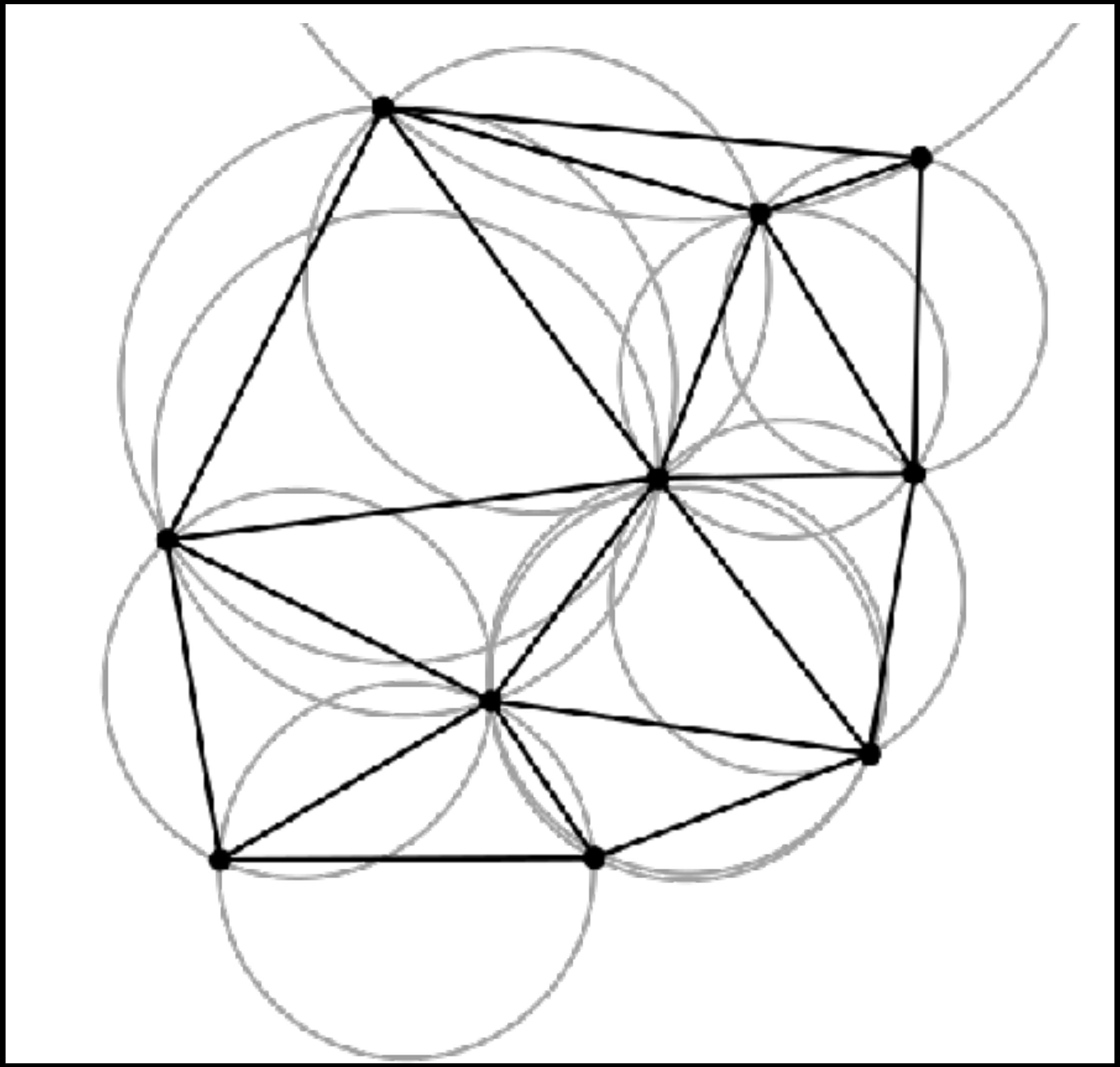
- exercise: implement fan triangulation of our convex hull
- see: ‘triangulate_polygon.py’
- goto: ‘http://localhost:5000/fan' to see results

# Point set Triangulation

- triangulate a point set.
- Delauney triangulations are particularly interesting.
  - super set of Gabriel graph, nearest neighbor graph, MST of the point set
  - avoids “slivers”, making it ideal for surfaces and interpolations (TIN)
  - dual of the Voronoi diagram (Theissen Polygons)

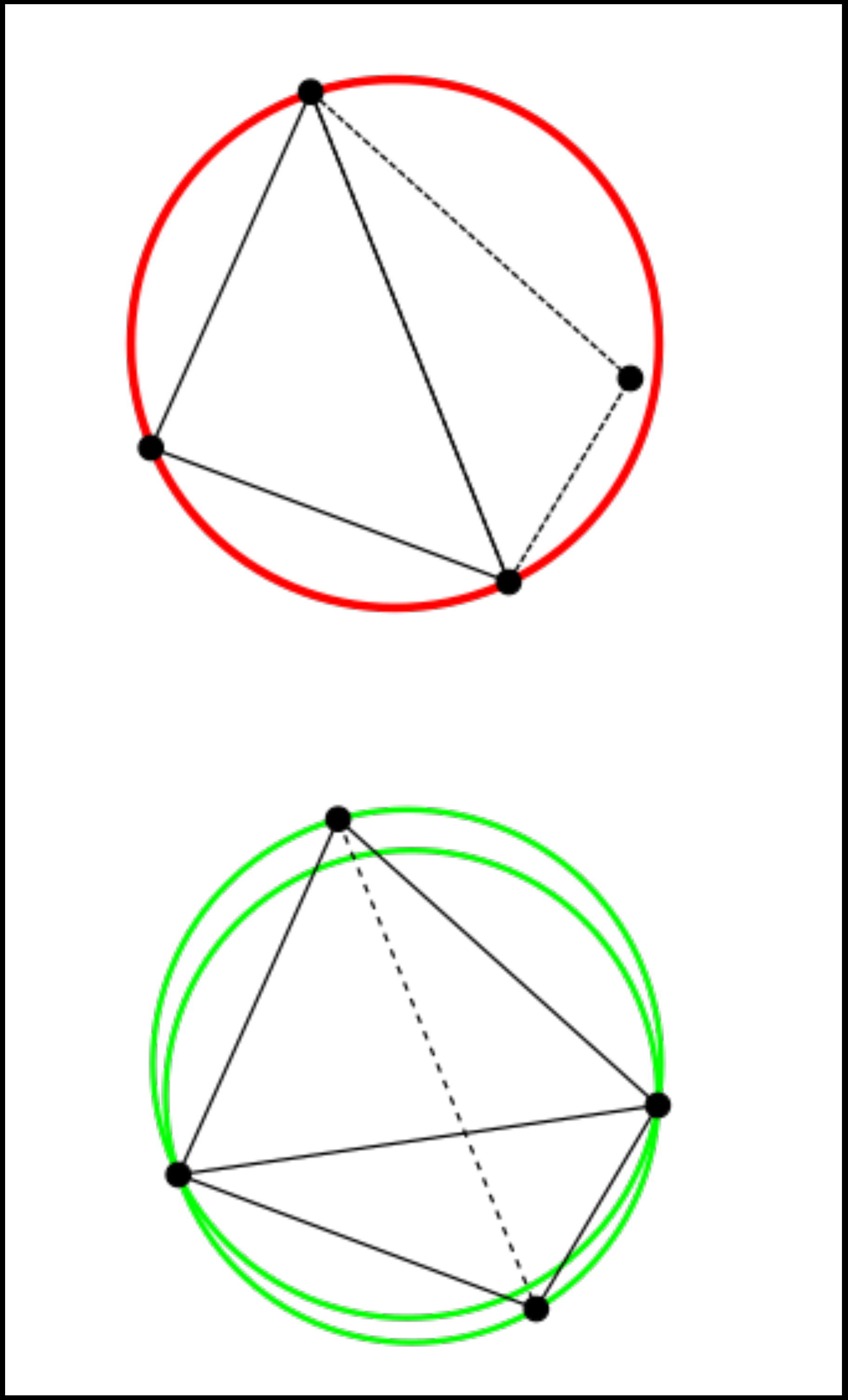
## Point set Triangulation

- Delaunay triangulations maximize the minimum angle of all the angles of the triangles in the triangulation



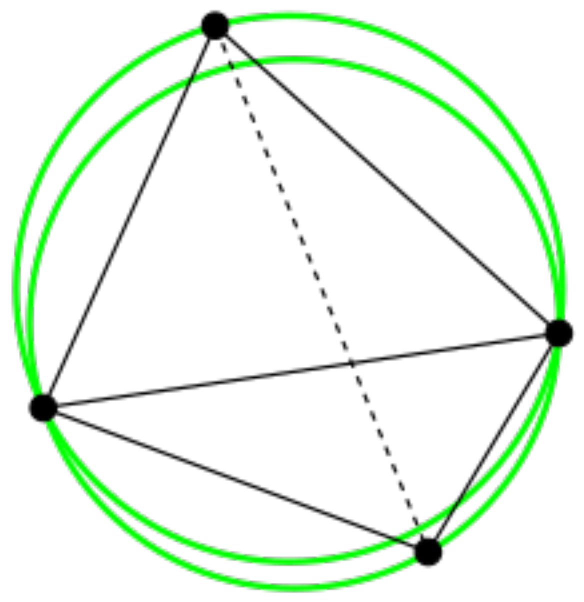
# Point set Triangulation

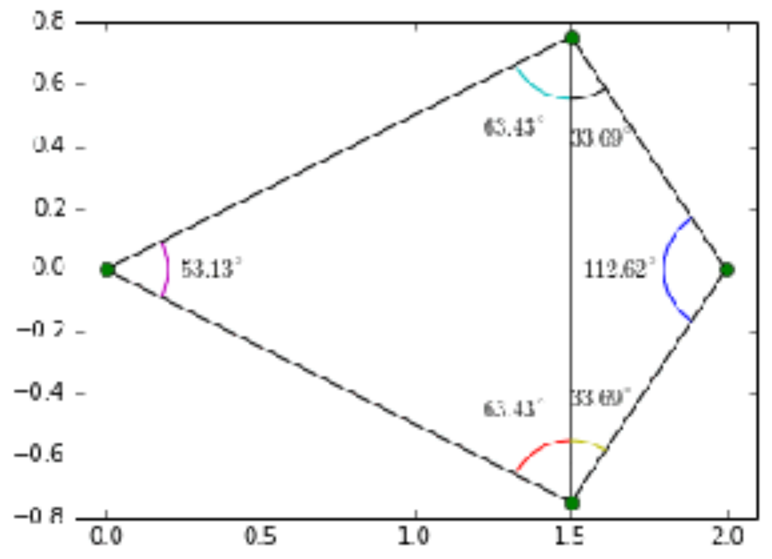
- Delauney condition:
- adjacent triangles in the triangulation form a convex quadrilateral
- the circle containing the 3 points of one triangle must not contain the 4th



# Point set Triangulation

- Delauney condition (cont'd)
- if the Delauney condition fails, flipping the interior edge produces a Delauney edge.





## Point set Triangulation

- One algorithm (inefficient) to calculate the Delauney triangulation of a point set is to calculate any triangulation of the point set, then iterate through edges, test for the Delauney condition, and flip edges where necessary.

## Point set Triangulation

- We won't calculate the Delauney triangulation of a point set, but we will calculate a triangulation of a point set.
- We’ll use the Triangle Splitting algorithm.

# Triangle Splitting Algorithm

- Find the convex hull of the point set.
- Triangulate this hull.
- iterate through interior points
  - if a point is inside a triangle, split the triangle into 3 new triangles using the current point.
  - continue until all interior points are exhausted

## Triangle Splitting Algorithm

- exercise: implement triangle splitting algorithm
- see: ‘triangulate_points.py’
- goto: ‘http://localhost:5000/triangulation' to see results

## A more elegant Delauney solution

- https://compgeo.github.io/voronoi-3d/src/index.html

# A more elegant Delauney solution

- also Fortune’s algorithm: O(n log n)