Multi Party Computation for the Ride Sharing Economy

Sean Smith

December 14th, 2016

Ridesharing, the act of getting a ride in a car not registered as a Taxi or Limousine, has created a new transportation option. The convenience and affordability has convinced many to use the service instead of public transport or traditional taxi cabs. In doing so it has created vast economic opportunity and has displaced incumbent industries, most notably the taxi industry. This has created many political and social issues, sparking protests and legislation around the world.

In New York City, the Taxi and Limousine Commission (TLC), a government body in charge of regulating the taxi industry, has nearly shut down a ridesharing service "Uber" for failure to provide trip data as required by a new law aimed at the ridesharing industry. Uber claims that the trip data is business critical data and that releasing it would violate the privacy of its users and damage its business. The TLC wants the data to get a better picture of transportation (it collects Taxi trip data already) and to prevent gridlock. Uber has agreed to release data but only at the granularity of bases which is not specific enough to satisfy the new legislation.

The Taxi industry provides data to the TLC and subsequently the public, which includes the latitude and longitude of a trip start and end, the time the trip took, and the number of passengers the trip included ("NYC Taxi & Limousine Commission - Trip Record Data"). This data has subsequently been used as a tool for traffic analysis, notably in Dan Work and Brian Donavan's paper on traffic analysis during Hurricane Sandy ("DonovanWorkTR-C2016.pdf").

In contrast, For Hire Vehicles (FHV's), which include companies such as Uber and Lyft, provide very minimal and wildly unstandardized data. In July 2015, Five Thirty Eight filed a freedom of information request to the TLC in order to get access to the newly collected FHV vehicle data. When examining the data I noticed that the file formats differ significantly from company to company, some use street addresses, some use coordinates, some use base codes, all are purposely vague and hard to aggregate. The data is publicly accessible on github (fivethirtyeight).

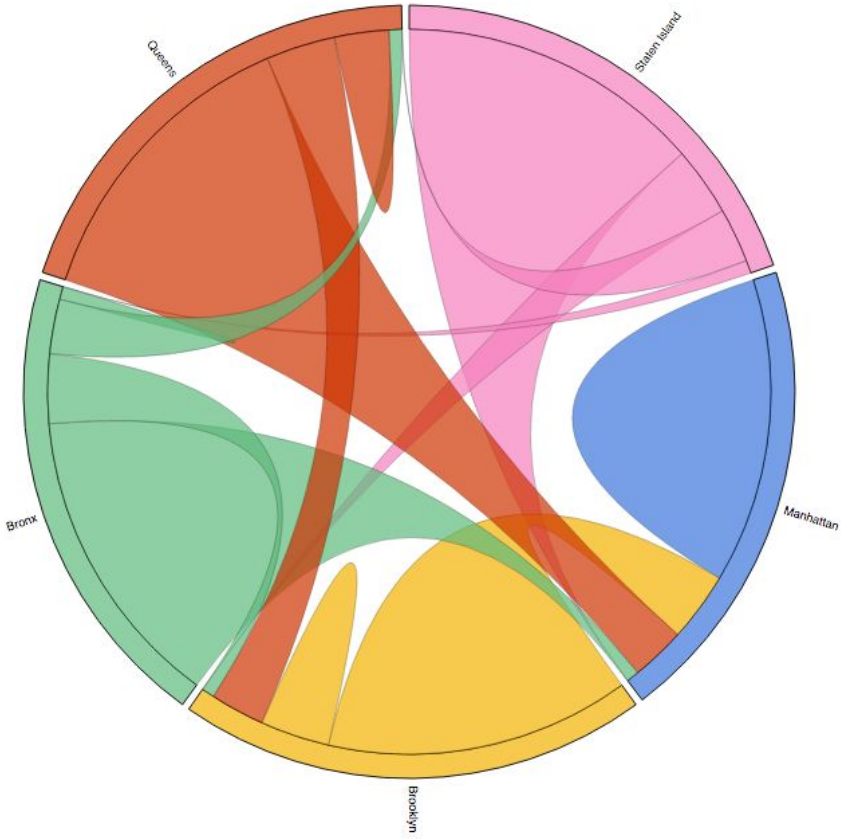
When we consider this from a data sharing standpoint, it seems trivial for ridesharing companies to hand the data over to the TLC and trust the TLC as a "Trusted Third Party", an organization that can be trusted to not share data with anyone. There are several reasons why the TLC cannot be considered a trusted third party. Primarily the TLC is a government organization and thus the data can be requested through a Freedom of Information Act (FOIA) request. Secondly, the TLC has traditionally regulated the incumbent Taxi industry and as a result campaign contributions have gone into creating a favorable legislative environment for the Taxi industry. The TLC enforces laws such as the medallion count, which has barely increased since first being instituted in 1937, creating a small amount (13,347) of medallions that are now worth over

$1 million dollars each (“Taxicabs of New York City - Wikipedia”). As a result, it is hard for any ridesharing company to trust that the TLC will use the data to benefit everyone and not just the taxi industry.

I am going to examine the issue of data sharing and propose a solution that uses multi-party computation to solve some of the privacy and business industry concerns. The proposed solution will focus on NYC but is applicable to any city or region in the world.

In the first solution I explored, each party provides data in a series of rounds. Each party will secret share their secret information with the other parties and then they will all calculate the region that the other parties coordinates are in without knowing the specific coordinates. The result of this computation is an NxN matrix where the start and end region are denoted by the row and column number, and the data in the matrix represents the number of trips from the start to the end region.

From this matrix that each party computed, a chord diagram similar to the one below can be computed. Each section of the outer circle represents a region, in this case I am using the five boroughs, and the lines between them indicate number of rides between two regions. The lines are proportional to the region so in the case of Staten Island (in pink below), the lines are thick because they represent all the rides departing in Staten Island but then get thin when they arrive in Brooklyn because they are only a tiny percentage of the total rides that arrive. An interactive version of the chart below can be viewed at http://mpc.seanssmith.com/all.



This solution makes both the TLC happy because they are able to run their studies on data with a fine grain of resolution and makes the FHV companies happy because they no longer need to hand over sensitive information.

I am going to define the secret portion of the data as the specific start and end coordinates of the trip. To obscure this data I need some notion of regions that I can map the coordinates into. The New York City Department of Planning provides data helpful for this (“Political and Administrative Districts - Download and Metadata”). On their website they provide a REST API to retrieve GeoJSON data, This data consists of polygons and multipolygons with various levels of resolution. They provide three levels of abstraction, at the most granular level they have a notion of census blocks, each block is a literal block and there are roughly 10,000 of these regions in the five boroughs of NYC. This level of abstraction is still far too specific for my purposes. The next level of abstraction is Census Tracts, each tract is 4-5 blocks and varies in size by population with the intent of having ~4,000 people in each block, there are 2,166 of these regions. See the figure below to see the size of each region:

Census Tracts (~2,166 Total)

Neighborhood Tabulation Areas (195 Total)

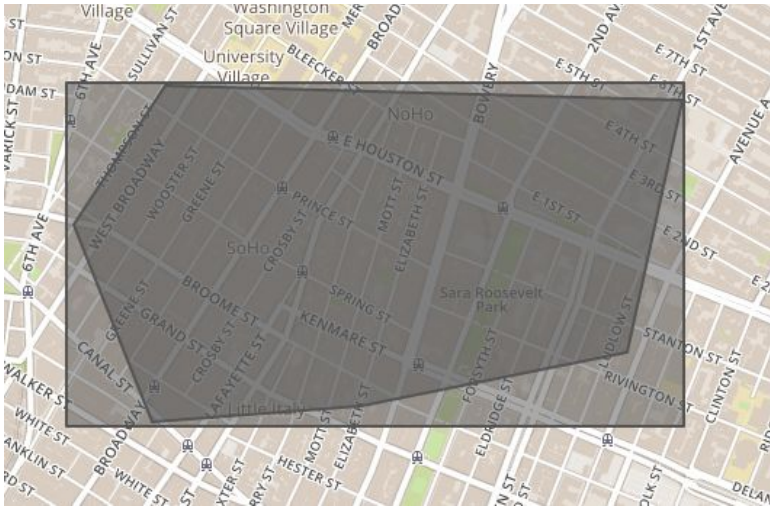
There are still too many regions to show a useful chord diagram. I am going to use the next level of abstraction, Neighborhood Triangulation Areas (NTA's), each NTA is a recognizable section of the city, in the figure above you will see names such as "West Village" or "Gramercy." There are 195 NTA's over the five boroughs.

Now that I have these GeoJSON files, I simply need to map coordinates into their corresponding zones and use that to create a matrix. In a typical large data analysis project, the first step would be to load all the data into a PostGIS database, this would provide a nice abstraction and give the ability to make simple SQL statements that

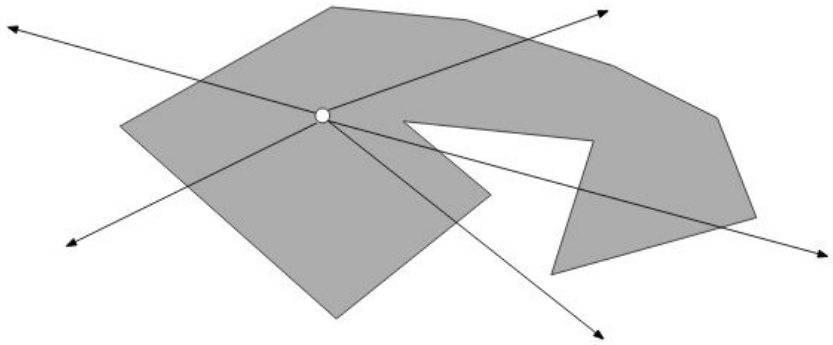
execute built in functions to do such things as determine if a point is in a polygon. Since I am doing this as a multi-party computation I need to write the functionality from scratch. To do this I am going to write a parser that parses the GeoJSON files and runs the point in a polygon algorithm.

A polygon in GeoJSON is a list of longitude, latitude coordinate pairs. The start and end of the list are the same and the polygon is rendered by drawing lines between each coordinate pair. The latitude longitude system is a cartesian coordinate system, with (0, 0) on the equator, just west of West Africa. From now I am going to refer to longitude as X and Latitude as Y. This will make it easier to conceptualize.

Determining if a point is in a square or rectangle is an easy task, you just need to check the X and Y axis of the point and check to see if it is between the X and Y coordinates of each face. This takes 4 comparisons and can be done quickly. When you have more than 4 sides it becomes trickier and takes significantly longer. The GeoJSON standard provides an optional bounding box, which is a rectangle that encloses a polygon completely and can be quickly checked to determine if the point is not in the polygon without going through all the work to determine if it definitely is or is not. Example below:



To extend this beyond 4 faces, we can use Jordan's Curve Theorem which states that a continuous loop in a plane, divides the plane into two regions, an interior region and an exterior region and so that any continuous path connecting the regions must intersect the curve ("Jordan Curve Theorem - Wikipedia"). From this theorem we can develop a parity test, given a point, that point is contained by a polygon if and only if a ray emanating from that point crosses an odd number of faces in any direction. In the figure below you can see that regardless of the direction, the ray intersects the curve an odd number of times.



To implement this for a polygon with straight edges we can make a couple of assumptions that simplify things. First we only need to consider rays that lie on the x axis. Secondly, instead of considering an infinite number of rays extending in every direction, we only need to consider one ray.

For each edge, we need to check if the ray intersects that edge. To do this we parameterize the x and y components of the ray and edge. The ray can be written as:

$$x(t) = P_x + t$$
$$y(t) = P_y$$

Where P_x, P_y is the point that we are testing for.

The edge can be written as:

$$x(u) = (V_{2x} - V_{1x})\,u + V_{1x}$$
$$y(u) = (V_{2y} - V_{1y})u + V_{2y}$$

Where (V_{1x}, V_{1y}) and (V_{2x}, V_{2y}) are the start and end coordinates of the edge that we are testing for. If the ray intersects the edge, it means that following equations were satisfied:

$$x(t) = x(u)$$
$$y(t) = y(u)$$
$$t > 0$$
$$0 \le u \le 1$$

The equation is only satisfied if the point is within the range of the Y coordinates of the edge, so we can check if one of the following equations is satisfied first:

$$V_{1y} \leq P_y \leq V_{2y} \text{ or}$$
$$V_{2y} \leq P_y \leq V_{1y}$$

Now, this creates a problem if the edge intersects one of the vertices. Since computers only store a finite precision, we can have the following two cases:

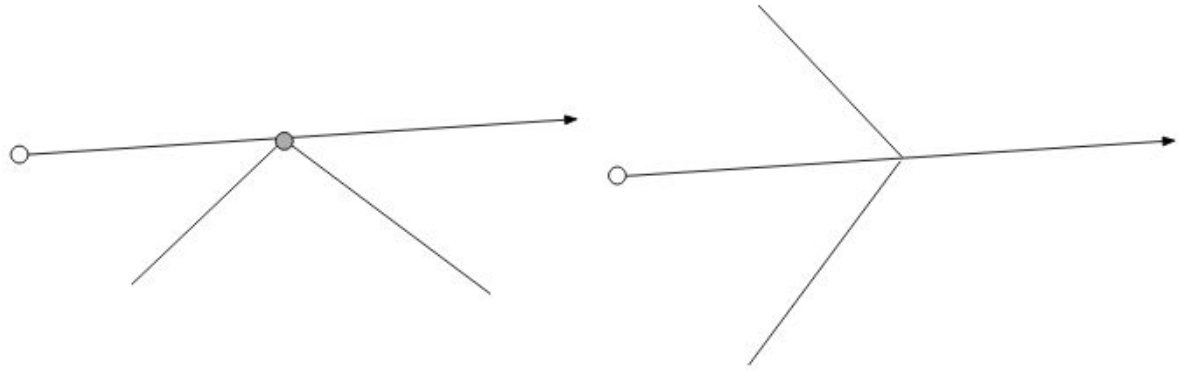


Figure 1
(Point is outside of Polygon)

Figure 2
(Point in Polygon)

In figure 1, because the ray intersects a vertex, the intersection is counted as a single intersection. This would return the wrong result that the point is contained in the polygon. Now let's say we decide to count all vertex as two intersections. This creates a problem in figure 2, now the ray crosses the boundary twice and is counted as outside the polygon.

In the paper Simulation of Simplicity, Edelsbrunner and Mucke came up with a simple solution to solve this problem ("Simulation of Simplicity"). This problem can be solved by adding a small offset to the vertices in the polygon. As you see below, this results in the correct number of crossings for each ray.

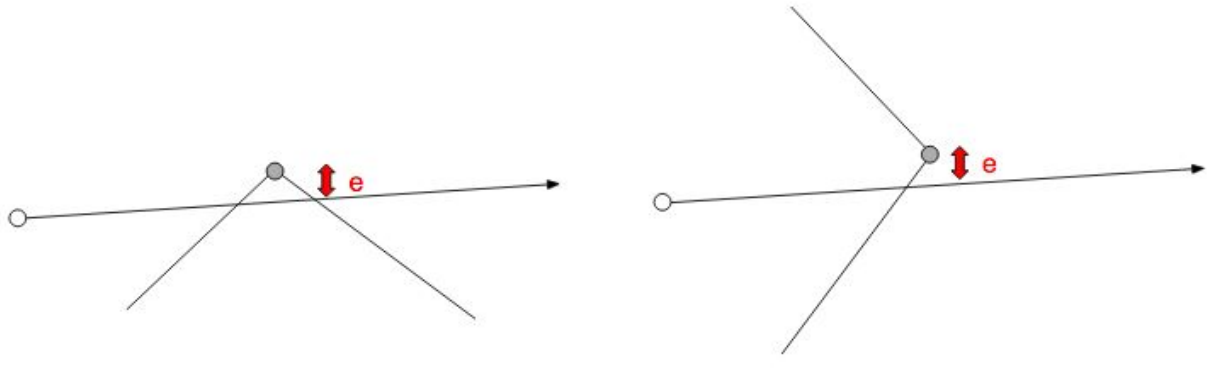


Figure 1

Figure 2

The following equation can be updated with e:

$$(V_{1y} + e) \leq P_y \leq (V_{2y} + e) \text{ or}$$
$$(V_{2y} + e) \leq P_y \leq (V_{1y} + e)$$

Since $e > 0$ and $e \to 0$, we can factor out e and we are left with:

$$V_{1y} \leq P_y < V_{2y} \text{ or}$$
$$V_{2y} \leq P_y < V_{1y}$$

The set of relations that determine if the ray intersects a given edge can be combined into the following relation:

$$P_x < \frac{(V_{2x} - V_{1x})(P_y - V_{1y})}{V_{2y} - V_{1y}} + V_{1x}$$

This can be fairly easily translated into code. Here is my implementation in python:

```
def in_polygon(lat, lng, polygon) :
    for coord in polygon :
        c = False
        j = len(coord) - 1;
        for i in range(len(coord)) :
            if ( ((y(coord[i]) > lat) != (y(coord[j]) > lat))
            and (long < (x(coord[j]) - x(coord[i])) * (lat - y(coord[i])) / (y(coord[j]) - y(coord[i])) +
        x(coord[i]))
                c = not c
                j = i
        if c :
            return True
    return
        False
```

Note that `coord` is not a coordinate but a set of coordinates. This is because multipolygons in GeoJSON can be a list of polygons so I need to test each one.

To test out this algorithm, I wrote a python script polygon.py that parsed the GeoJSON file for Census Tracts and Boroughs in New York and using the Google Maps

API, I created a web app to test my implementation of this point in polygon algorithm. You can try it out here: http://mpc.seanssmith.com

When you click on a point in the map, a POST request is made to my server which runs the point in polygon algorithm and asynchronously returns the result as a popup above the pin. You can select different overlays from the drop down menu, when selected, the overlay will asynchronously load onto the map. Note this takes a few seconds and does not work for Census Blocks since that GeoJSON file is > 100MB and is beyond the file size that Google Maps will render. If you click submit, it runs the same algorithm synchronously and reloads the page with the result.

So now with my point in polygon algorithm done, I needed to write a parser for the input data. The taxi cab data from the TLC is the highest quality data available and I used it, as it represents what the FHV vehicle companies should be providing the TLC. The taxi data can be obtained from the TLC website (NYC Taxi & Limousine Commission - Tri...). It is distributed in large CSV files for each month. I wrote a parser to take in this data, feed it into my point in polygon algorithm, and from the result create a matrix that represents the graph of the trips.

To test out this and my Chord Diagrams, I made a few test pages, http://mpc.seanssmith.com/green, http://mpc.seanssmith.com/yellow. I ingested 1,000 trips from Green and Yellow Taxi's, created a matrix that represents the trips between the five boroughs with the data obtained, and rendered it using the javascript library D3.

Let's say you want a more detailed view of trips. To do that I ran my algorithm on 10,000 taxi trips and generated the following graph between the 195 Neighborhood Tabulation Areas colored by borough http://mpc.seanssmith.com/nta (It takes a few seconds to render). This turned out to be far more connected than I thought and was ultimately not that useful.

Now I have the algorithm working in a non multi-party computation version. Next I implemented the same point in polygon algorithm using some of the multi party computation primitives provided by Viff.

Viff is a python framework that implements Shamir Secret Sharing and computation on those shares across a minimum of three different parties. Viff handles synchronizing communication, and overwrites the primitives provided by Python to do addition, multiplication, subtraction, and relationational operators. Division is not baked in but is included as an example and implements a protocol by Sigurd Meldgaard.

I decided to implement a simple test classification program to test the effectiveness of a multi-party based point in polygon algorithm. Each party starts with the same GeoJSON file which defines the zones that points are classified into. At the end of the computation, all parties are informed of the start zone, end zone of the other parties. With this information they can construct the matrix necessary to display the Chord Diagram. The algorithm works as follows:

1) Each party reads in their start latitude, start longitude, end latitude, and end longitude from a json file provided by the user.
2) The latitude and longitude values are converted into integers for easier computation. I only preserved 10^{13} places of precision which is plenty for the algorithm to work.
3) Each party then secret shares their coordinate pairs to the other parties. With three parties, this creates 12 shares.
4) For each starting and ending point for each party, I run point my point in polygon algorithm.
5) In the algorithm, I read the GeoJSON zone file and create a share out of each of the

This is implemented in point_in_polygon.py.

I ran this on a few small test examples and found that it was wildly inefficient, taking upwards of 10 minutes to classify a point in a simple polygon. Larger zone files, like the ones I discussed previously would fill my memory up and cause the OS to kill the process.

In addition, because the parties provide their own zone files, an adversary could use an extremely granular zone file to get the coordinate pairs with close precision to what the actual coordinates are. Because we only care about aggregate values in each zone, I am proposing an alternate algorithm.

Instead of computing the zone that contains each coordinate pair, I am only going to combine the final output matrix of each party. This simplies things significantly. For N zones and P parties, there are $N^2 * P$ number of shares. In the previous approach there would be $4P + 2C$ shares, where C is the number of coordinate pairs in the zone file. In a real zone file there are thousands of coordinates.

I wrote a script sum_matrix.py that implements this for three parties. I ran this on the matrix I generated with 1,000 Green Taxi trips, 1,000 Yellow Taxi trips, and fake Uber data. The resulting Chord Diagram can be viewed here:
http://mpc.seanssmith.com/all

By doing the zone computation within each party, I was able to run this script in less than 30 seconds as opposed the the 10 minutes per coordinate pair that I was experiencing before. This runtime makes it reasonable to for companies to do frequently and give the TLC a time varying view of transportation throughout the city.

The code I wrote, I wrote to be as modular as possible, this allowed me to easily test out the algorithms with different zone files and render the results. I hope going forward, other people will use and extend my code. It's publicly accessible on github here: https://github.com/sean-smith/boroughs

For future work I'd like to visualize the time varying nature of the rides, so you can see the effect of rush hour and traffic anomalies. I think creating a chord diagram for how trips change would be a compelling visualization.

Sources

"DonovanWorkTR-C2016.pdf." *Dropbox*. N.p., n.d. Web. 13 Dec. 2016.

fivethirtyeight. "Fivethirtyeight/uber-Tlc-Foil-Response." *GitHub*. N.p., n.d. Web. 13 Dec. 2016.

"Jordan Curve Theorem - Wikipedia." N.p., n.d. Web. 14 Dec. 2016.

"[No Title]." N.p., n.d. Web. 14 Dec. 2016.

"NYC Taxi & Limousine Commission - Trip Record Data." N.p., n.d. Web. 13 Dec. 2016.

"Political and Administrative Districts - Download and Metadata." N.p., n.d. Web. 14 Dec. 2016.

"Taxicabs of New York City - Wikipedia." N.p., n.d. Web. 13 Dec. 2016.

Information Referenced

1) 538 Article on “Uber Is Taking Millions Of Manhattan Rides Away From Taxis” http://fivethirtyeight.com/features/uber-is-taking-millions-of-manhattan-rides-away-from-taxis/
2) Uber and NY Taxi data: https://github.com/toddwschneider/nyc-taxi-data
3) Forbes article on why Uber is handing over data to cities http://www.forbes.com/sites/ellenhuet/2015/01/13/uber-boston-city-data/#349a04ba7ae
4) http://observer.com/2014/10/the-tlc-meeting-that-ended-in-screams/#ixzz3O53IojNZ
5) http://www.newsweek.com/new-york-city-suspends-uber-over-trip-records-297139 Actually suspended 5 out of 6 uber bases in NY
6) Rules on why uber has to hand over data to TLC: http://www.nyc.gov/html/tlc/downloads/pdf/newly_passed_rules_fhv_dispatch_rules.pdf
7) Medallion data
8) http://daim.idi.ntnu.no/masteroppgaver/004/4559/masteroppgave.pdf Overview of Shamir Secret Sharing and implementation for a simple voting scheme
9) Dan Work Traffic Modelling nyc
10) http://engineering.illinois.edu/news/article/9717
    a) https://publish.illinois.edu/dbwork/research/
    b) http://www.radicalcartography.net/
11) https://newsroom.uber.com/us-california/uberdata-san-franciscomics/ Uber rides in SF
    a) Breaks down data in crazy unheard of ways, ie by race, gender, crime rate ect
12) I really liked the visualization here: https://bost.ocks.org/mike/uberdata/
13) I found code to create that visualization here: https://github.com/sghall/d3-chord-diagrams.git
    a) Another Example: http://projects.delimited.io/experiments/chord-transitions/demos/trade.html
14) I got data from here: http://www1.nyc.gov/site/planning/data-maps/open-data/districts-download-metadata.page
15) I really liked the analysis done by Todd Schneider:
    a) http://toddwschneider.com/posts/analyzing-1-1-billion-nyc-taxi-and-uber-trips-with-a-vengeance/
16) Geojson is fucking awesome: https://github.com/tmcw/awesome-geojson
17) Point inclusion in a polygon: https://www.ecse.rpi.edu/Homepages/wrf/Research/Short_Notes/pnpoly.html
18) Jordan Curve Theorem problem with vertices http://wiki.cizmar.org/doku.php?id=physics:point-in-polygon_problem_with_simulation_of_simplicity
19) https://arxiv.org/pdf/math/9410209.pdf Simulation of Simplicity, a Technique to Cope with Degenerative cases in Geometric Algorithms