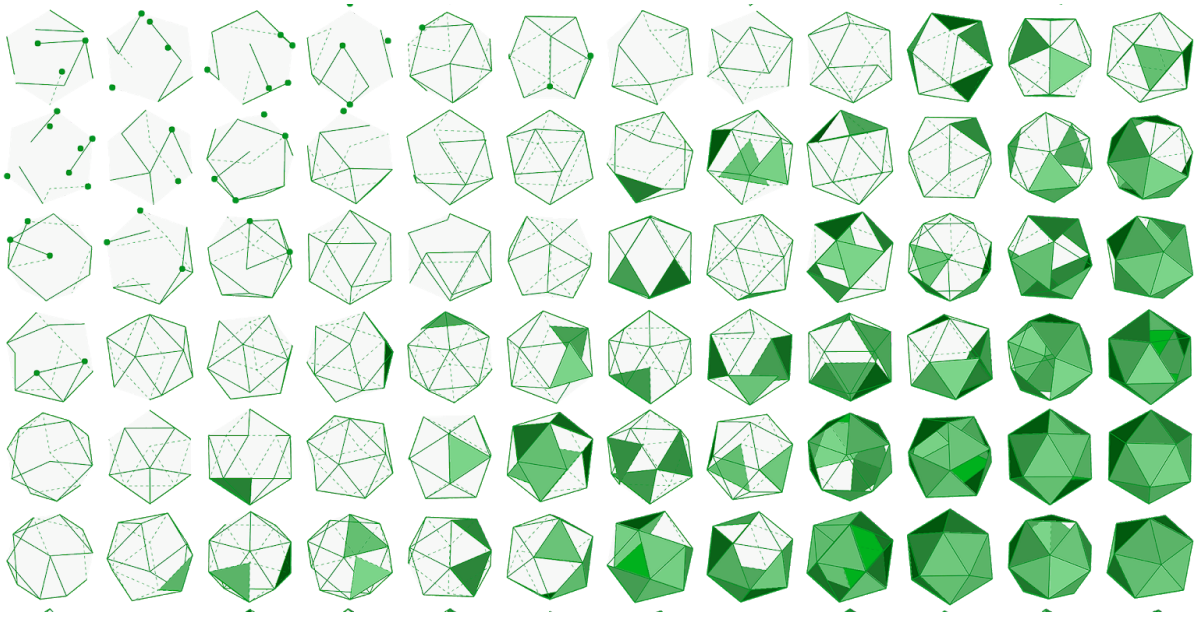
ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ & ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ



Assignment 1

Computational Geometry 2020

Ομάδα 10
Σουλούνιας Νικόλαος (sdi1600156)
Ιγιάμου Περισανίδης Σίμων (sdi1600051)

Table of Contents

Exercise 1

> Implement an algorithm that takes as input three points in the plane. checks that they form a triangle and whether the interior of the triangle contains the origin (0, 0) or not.

Dependencies

Python 3.7.6

Usage

```
$ python q1/main.py
```

Background

- Three points form a triangle if the area of the shape they form is > 0.
- Let A,B,C be the vertices of a triangle, and P a point. Let A0,A1,A2,A3 be the area of the triangles ABC, PBC, PAC, PAB respectively. Then, the point P is inside the triangle ABC iff A0 = A1+A2+A3.

The statements are trivial so no proof is provided.

Approach

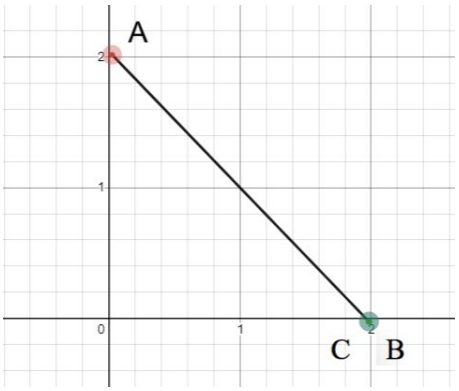
The aforementioned statements are translated into code.

In order to check if three points form a triangle, the function `area()`, which computes the area of the triangle formed by the three given points, is used. If the returning value is bigger than zero, then the answer is yes. This is exactly what the function `isTriangle()` checks.

For the second problem, we have implemented the function `containsOrigin()` which takes three points as input and returns true if the formed triangle contains the axis origin. The way it's done is based on the second statement from above, by using the origin (0,0) as point P. Initially, the areas of the four triangles are computed. If the area of the big triangle is equal to the sum of the areas of the three small triangles, then the axis origin is contained in the triangle ABC.

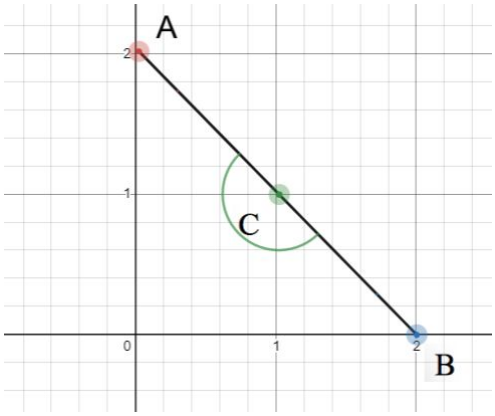
In the python script there are four examples that show the functionality of the algorithms.

Example 1



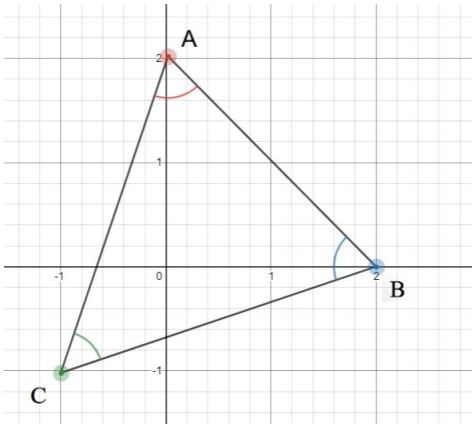
The vertices C and B are identical, so the expected output is that the points A,B and C do not form a triangle.

Example 2



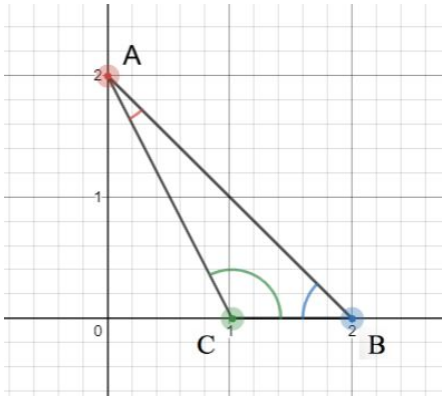
The vertices C is in the line AB, so the expected output is again that the points A,B and C do not form a triangle.

Example 3



The vertices form a triangle and also the axis origin is contained in it.

Example 4



The vertices form a triangle but the axis origin is contained in it.

Exercise 2

Given a circle of radius r in the plane with (0, 0) as center, implement an algorithm that finds the total lattice points on the circumference. Lattice Points are points with integer coordinates.

Implementation

lattice_points.py

Execution Instructions

```
$ python3 lattice.py
```

When executed, the program waits for the user to insert the value of the radius r. (Be sure to press ENTER after typing the number you want). The radius r must be a positive number (either float or integer). Otherwise, an error message is printed and the application terminates unsuccessfully.

If a positive number is given as the radius r, the program finds the lattice points of the circle, prints them and also plots them in a figure that is also saved as “lattice_points.png”. To terminate the application, the user must close the figure window that pops up.

Dependencies

matplotlib (pip3 install matplotlib)

The library is used to create a plot of the circle with center (0,0) and radius r and its lattice points. The circle is coloured blue and its lattice points are coloured red. The coordinates of the center and of the lattice points are also printed in the plot.

Basic Idea and Concept

The equation of a circle with center (0,0) and radius r is the following:

$$x^2 + y^2 = r^2$$

Let x_L, y_L be the coordinates of a (random) lattice point P of the circle. Since L is a point of the circle:

$$x_L^2 + y_L^2 = r^2$$

$$\Rightarrow \qquad 0 \leq x_L^2, y_L^2 \leq r^2$$

$$\Rightarrow \qquad -r \leq x_L, y_L \leq r$$

Moreover, by the lattice points definition:

$$x_L, y_L \in \mathbb{Z}$$

So, a first approach to the problem would be the following:

Lattice Points: 1^{st} edition

```
for x := -r to r:
     y := √(r^2 − x^2)
     if isInteger(y):
          (x,y) is a lattice point
          if y>0:
               (x,-y) is a lattice point
```

In the 1^{st} edition of the lattice points algorithm we calculate in the same repetition whether a point (x,y) and its symmetrical point across the x-axis (x,-y) are lattice points.

So, the 1^{st} edition of the algorithm is based on the following entailment (symmetry of the circle across the x-axis):

$$(x\ ,\ y)\ \ is\ \ a\ \ lattice\ point \quad \Rightarrow \quad (x\ ,\ -y)\ is\ a\ lattice\ point$$

Since the values given to x throughout the loop are integer values, for a point (x,y) to be a lattice point of the circle, we must simply check that y is also an integer.

To avoid counting each of the points (-r,0) and (r,0) twice, we check if y>0.

How could we make this algorithm better? We should reduce the number of repetitions, therefore the number of values we check for x.

How will we do this? We will take advantage of the symmetry of the circle across the y-axis.

Lattice Points: 2^{nd} edition

```
for x := 0 to r:
     y := √(r^2 − x^2)
     if isInteger(y):
          (x,y) is a lattice point
          if x>0:
                (-x,y) is a lattice point
          if y>0:
                (x,-y) is a lattice point
          if x>0 and y>0:
                (-x,-y) is a lattice point
```

The 2^{nd} edition of the algorithm is based on the following entailment (symmetry of the circle across the y-axis):

$$(x\ ,\ y)\ \ is\ \ a\ \ lattice\ point\ \ \Rightarrow\ \ (-x\ ,\ y)\ is\ a\ lattice\ point$$

To avoid counting each of the points (-r,0), (r,0), (0,r) and (0,-r) twice, we check if x>0 and y>0 .

Is this the best we could do though? Nope.
The next edition of the lattice points algorithm is the following:

Lattice Points: 3^{rd} edition

```
for x := 0 to floor(r/√2):
     y := √(r^2 − x^2)
     if isInteger(y):
          (x,y) is a lattice point
          (y,x) is a lattice point
          (x,-y) is a lattice point
          (-y,x) is a lattice point
          if x>0:
                (-x,y) is a lattice point
                (y,-x) is a lattice point
               (-x,-y) is a lattice point
                (-y,-x) is a lattice point
```

The 3^{rd} edition of the algorithm is based on the following entailment (symmetry of the circle across the y=x line):

$$(x\ ,\ y)\ \ is\ \ a\ \ lattice\ point\ \ \Rightarrow\ \ (\ y,x)\ is\ a\ lattice\ point$$

So, while in the 2^{nd} edition we were checking a quartet, in the 3rd edition we are checking an octant.

There is no need to check whether y>0 as we did in the 2^{nd} edition, since we are working in the 2^{nd} octant, which mean that $r/\sqrt{2} \le y \le r$.

NOTE: All the numbers that are represented in a computer have the form $\pm 2^{e} * number$. So r will be a positive rational number.

So taking that note into consideration, there is no need to eliminate duplicates if a lattice point has y = $r/\sqrt{2}$, simply because it can't. That is because $\sqrt{2}$ is an irrational number, therefore $r/\sqrt{2}$ is an irrational number, not an integer.

I must also note that the computation of r/$\sqrt{2}$ will not be 100% precise (because $\sqrt{2}$ cannot be represented accurately). However, the precision error of the bound is not a problem practically.

This is the concept of the algorithm pretty much. However, there is still one improvement we could implement.

Lattice Points: 4^{th} edition

```
if isNotInteger(r):
     There are no lattice points.
     terminate
for x := 0 to floor(r/√2):
     y := √(r^2 − x^2)
     if isInteger(y):
          (x,y) is a lattice point
          (y,x) is a lattice point
          (x,-y) is a lattice point
          (-y,x) is a lattice point
          if x>0:
                (-x,y) is a lattice point
                (y,-x) is a lattice point
                (-x,-y) is a lattice point
                (-y,-x) is a lattice point
```

The 4^{th} edition of the algorithm is based on the following entailment:

$r \notin \mathbb{N}$ and r represented by a finite number of digits and $x \in \mathbb{N} \Rightarrow$

$\Rightarrow \quad y = \sqrt{r^2 - x^2} \notin \mathbb{N}$

We are once again based on the fact that the given radius cannot be an irrational number.

Mathematical Proofs

Let $(x\ ,\ y)\ \ be\ \ a\ \ lattice\ point$:

$$x^2 + y^2 = r^2$$

and

$$x, y \in \mathbb{Z}$$

- $(x\ ,\ y)\ \ is\ \ a\ \ lattice\ point \quad \Rightarrow \quad (x\ ,\ \ -y)\ is\ a\ lattice\ point$

  For the point (x,-y) we have:

  $x^2 + (-y)^2 = x^2 + y^2 = r^2$, which means that (x,-y) is a point of the circle.

  and

  $x, -y \in \mathbb{Z}$, since $x, y \in \mathbb{Z}$

  So, the point (x,-y) is a lattice point.

- $(x\ ,\ y)\ \ is\ \ a\ \ lattice\ point \quad \Rightarrow \quad (-x\ ,\ y)\ is\ a\ lattice\ point$

  For the point (-x,y) we have:

  $(-x)^2 + y^2 = x^2 + y^2 = r^2$, which means that (-x,y) is a point of the circle.

  and

  $-x, y \in \mathbb{Z}$, since $x, y \in \mathbb{Z}$

  So, the point (-x,y) is a lattice point.

- $(x\ ,\ y)\ \ is\ \ a\ \ lattice\ point\ \ \Rightarrow\ \ (\ y,x)\ is\ a\ lattice\ point$

  For the point (y,x) we have:

  $y^2 + x^2 = x^2 + y^2 = r^2$, which means that (y,x) is a point of the circle.

  and

  $$x, y \in \mathbb{Z}$$

  So the point (y,x) is a lattice point.

- $r \notin \mathbb{N}$ and r represented by a finite number of digits and $x \in \mathbb{N}$ $\Rightarrow$

  $\Rightarrow$ $$y = \sqrt{r^2 - x^2} \notin \mathbb{N}$$

  Let d be the last non-zero digit of the decimal part of r. Then

  $$r^2 \in \mathbb{N} \quad \Rightarrow \quad d^2 mod\ 10\ = 0\ \text{(S)}$$

  since $d^2 mod\ 10$ will be in the decimal part of r^2, which we want to be 0.

  - $d = 1\ \Rightarrow d^2 = 1\ \Rightarrow d^2 mod\ 10\ = 1\ \neq\ 0$
  - $d = 2\ \Rightarrow d^2 = 4\ \Rightarrow d^2 mod\ 10\ =\ 4\ \neq\ 0$
  - $d = 3\ \Rightarrow d^2 = 9\ \Rightarrow d^2 mod\ 10\ =\ 9\ \neq\ 0$
  - $d = 4\ \Rightarrow d^2 = 16\ \Rightarrow d^2 mod\ 10\ =\ 6\ \neq\ 0$
  - $d = 5\ \Rightarrow d^2 = 25\ \Rightarrow d^2 mod\ 10\ =\ 5\ \neq\ 0$
  - $d = 6\ \Rightarrow d^2 = 36\ \Rightarrow d^2 mod\ 10\ =\ 6\ \neq\ 0$
  - $d = 7\ \Rightarrow d^2 = 49\ \Rightarrow d^2 mod\ 10\ =\ 9\ \neq\ 0$
  - $d = 8\ \Rightarrow d^2 = 64\ \Rightarrow d^2 mod\ 10\ =\ 4\ \neq\ 0$
  - $d = 9\ \Rightarrow d^2 = 81\ \Rightarrow d^2 mod\ 10\ =\ 1\ \neq\ 0$

  So, $d^2 mod\ 10\ \neq\ 0$.

  $$d^2 mod\ 10\ \neq\ 0 \quad \Rightarrow \quad r^2 \notin \mathbb{N}$$

Is true as the contrapositive of sentence S.

Therefore, $r^2 \notin \mathbb{N}$.

Since multiplication is closed in the set $\mathbb{N}$ and $x \in \mathbb{N} \Rightarrow x^2 \in \mathbb{N}$.

Now, let $r^2 - x^2 = y^2 \in \mathbb{N}$.

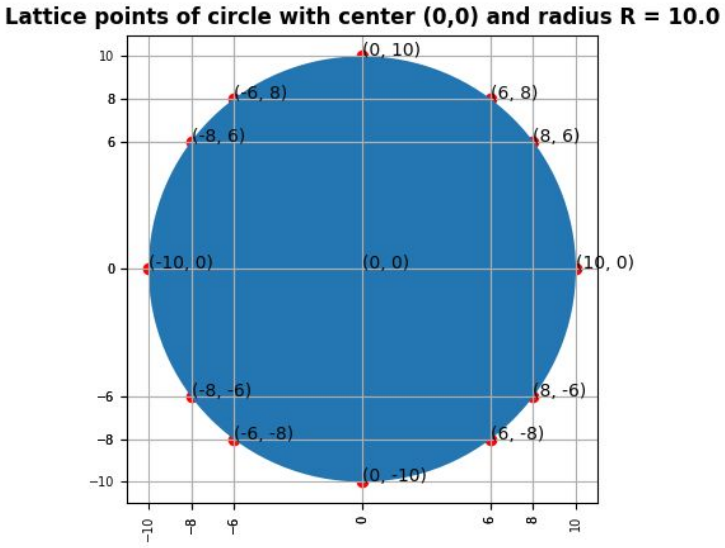
Since addition is closed in the set $\mathbb{N}$ and $x^2 \in \mathbb{N}$ and $y^2 \in \mathbb{N} \Rightarrow$

$\Rightarrow r^2 = x^2 + y^2 \in \mathbb{N}$ contradiction So, $r^2 - x^2 = y^2 \notin \mathbb{N}$

Now, let $y = \sqrt{y^2} \in \mathbb{N}$

Since multiplication is closed in the set $\mathbb{N}$ and $y \in \mathbb{N}$ $\Rightarrow$ $y^2 \in \mathbb{N}$.

contradiction So, $y \notin \mathbb{N}$



Exercise 3

> Implement the incremental 2D algorithm for computing the convex hull of a finite set of points in the plane.

Dependencies

- Python 3.7.6
- matplotlib
- numpy

Usage

```
$ python q3/main.py
```

Background

- The **convex hull** of a shape is the smallest convex set that contains it. It can be visualized as the shape enclosed by a rubber band stretched around a bounded subset of a planet.
- **Incremental Convex Hull Algorithm** basic idea: First take a subset of the input small enough so that the problem is easily solved. Then, one by one add remaining elements while maintaining the solution at each step.
- **Tangent** to Polygon: a line that touches it without crossing its boundary.
- For a convex polygon, there are **exactly two unique tangents** from a point outside the polygon.

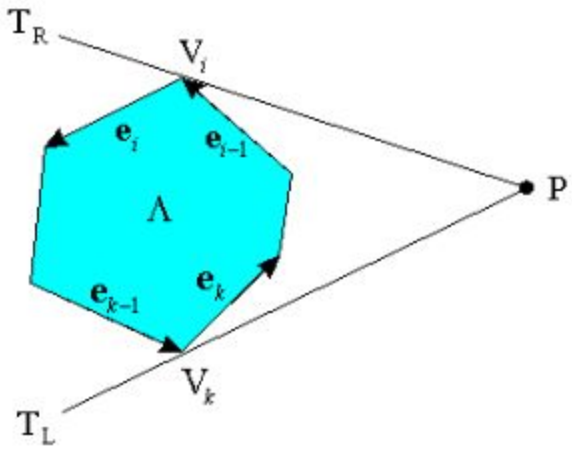
Approach

We define a class named Point in Point.py, in order to represent the points in the 2D plane.

The algorithm is implemented in main.py. With the aim of forming the input set, 20 random sample points are created.

Initially, **the points are sorted** based on their x-coordinate. Then the three first points are chosen so as to begin with a trivial problem as that of a triangle. Obviously, the convex hull of a triangle is composed of all of its edges.

Next, we add the rest of the points one by one. For each point P we work as follows:



- Find the Upper Tangent point from point P to the convex hull.

- Search for the Lower Tangent, going clockwise around the convex hull polygon, starting from the Upper Tangent.

- Expand the convex hull to contain point P and the new edges (PV_i and PV_k)

Note that, while searching for the Lower Tangent, the points that are visited but are not tangency points, are removed from the hull.

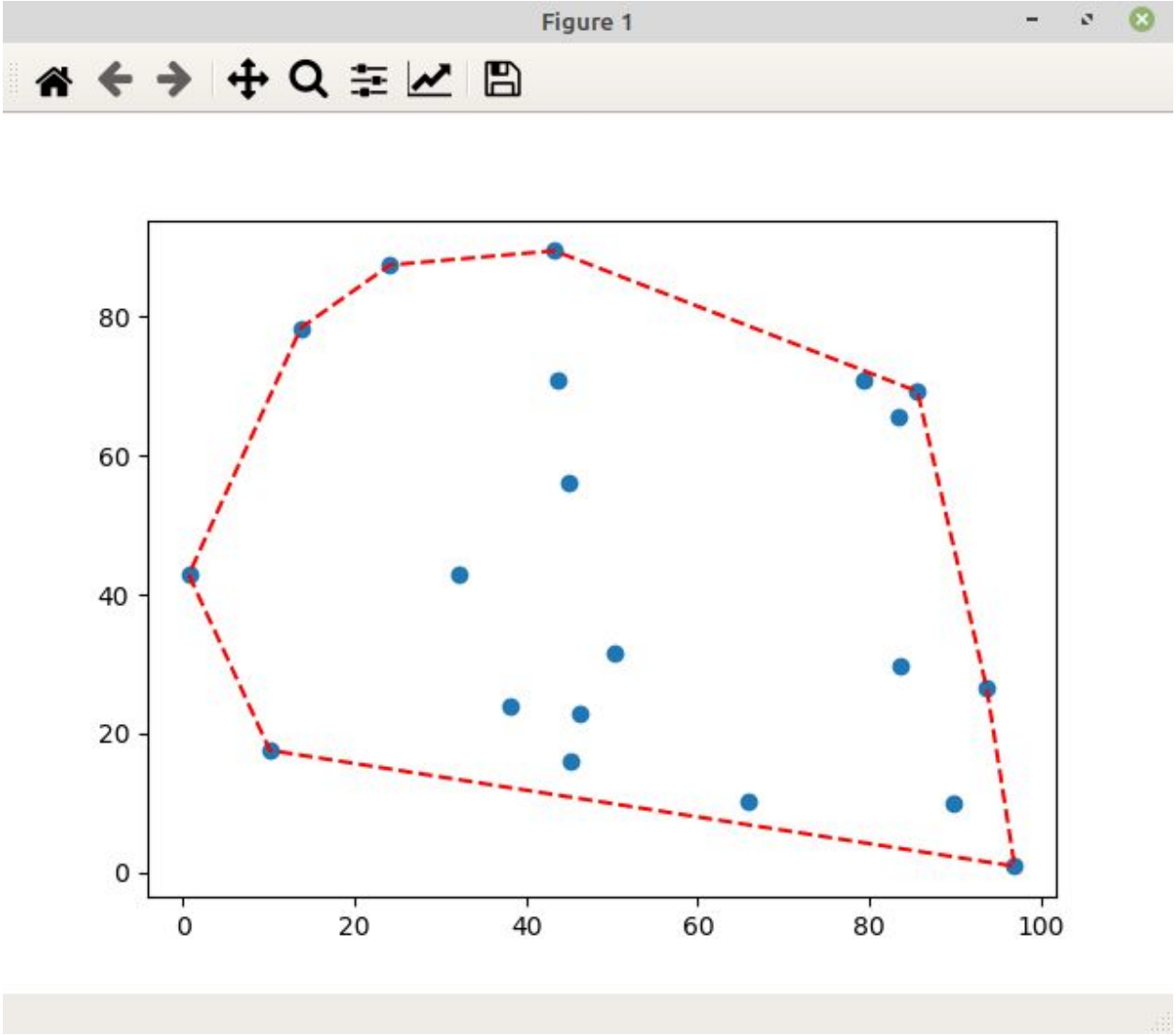
Now one could ask, how do you check whether a point is an Upper/Lower tangency point or not? Well, it's simple. Take for example the vertex Vi. In order to determine whether it is the upper tangency point, we can check whether the point P is to the right of edge e_{i-1} and to the left of edge e_i. The reverse applies to the lower tangency point Vk.

In general, if P is inside both of these edges, or outside both edges, then those two edges are on different sides of PVi, and thus PVi cannot be a tangent. On the other hand, if P is inside one edge and outside the other, then PVi is locally tangent to the polygon at Vi.

This idea is implemented by the functions `isUpperTangent()`, `isLowerTangent()`, `isLeftOf()`, `isRightOf()`.

In the end, the points and the convex hull are plotted to give us a visual representation of the results.

An example output plot is the following.



Exercise 4

Implement the gift wrap algorithm for computing the convex hull of a finite set of points in the plane

Implementation

gift_wrap.py

Execution Instructions

```
$ python3 gift_wrap.py
```

When executed, the program waits for the user to insert the number of 2D points that will be randomly generated. (Be sure to press ENTER after typing the number you want). The given number must be a positive integer. Otherwise, an error message is printed and the application terminates unsuccessfully.

If a positive number is given as the number of 2D points, the program generates that number of 2D points. Then the program finds the convex hull of the points using the gift wrap algorithm. The program prints the generated points, as well as the points of the convex hull. The program plots the points and the convex hull in a figure that is saved as “gift_wrap.png”. To terminate the application, the user must close the figure window that pops up.

Dependencies

- Python 3.6.9
- matplotlib (pip3 install matplotlib - check the Dependencies of Exercise 2 for more details)
- numpy (pre-installed library)
- Sys (pre-installed library)

Basic Idea and Concept

The algorithm starts by finding the left-most point in the convex hull points. If multiple points of the set have the same minimum coordinate on the x-axis, we choose among them the point with minimum coordinate on the y-axis. For any given set of points, this point will belong to the convex hull of the set.

Then, in each iteration the algorithm chooses a point that has not already been placed in the convex hull list. Then the algorithm checks if this point is the next point of the convex hull polygon. For this to be true, the new point must maximize the angle formed when connected with the previous point of the convex hull polygon. So, how will we check that? Let's suppose r is the last point inserted in the convex hull polygon and u is our candidate for the next point of the convex hull polygon and t is a random point different from u that also has not been inserted in the convex hull list.

We form the vectors $\vec{ru}$ and $\vec{rt}$. To maximize the angle, when we rotate the vector $\vec{ru}$ towards $\vec{rt}$ the rotation rotation should be counterclockwise(CCW). Otherwise, the angle formed when t is chosen as the next point of the convex hull is greater. So, in this case, we set u = t. If we have checked that angle formed when a point s is chosen is less than the angle formed when u is chosen, then the angle for t will be greater than the angle for s. So, there is no need to double check points in a single iteration of the algorithm.

Now, a detail that we have to address is what happens when r,u,t are collinear. Well, we just check if u is in the middle of them. If that's the case we set u = t. Otherwise we continue our search for the best candidate.

The algorithm terminates when u = r0 and the convex hull polygon is complete.

Check out the pseudocode for more details.

Convex Hull Gift Wrap Algorithm

Input: 2d_points (a list containing the 2d points of the plane)

Output: convex_hull_list (a list of 2d points that form the convex hull of all the points in the 2d_list)

```
if 2d_points contains 1 or 2 points:
     convex_hull_list = 2d_points
     return convex_hull_list
Find the left-most points in the 2d_list. If there are
multiple points with the same minimum x-coordinate, choose
the point with the minimum y-coordinate. Let r0 be the
point we found.
convex_hull_list = [ r0 ]
unused_points_list = 2d_points (Notice we keep r0 in the
                                list)
u = None
while u != r0:
     r = last element inserted in the convex_hull_list
     u = unused_points_list[0]
     counter = 1

     if r == r0 and u == r0:
          u = unused_points_list[1]
```

```
        counter = 2
    while counter < #elements in unused_points_list:
        t = unused_points_list [ counter ]
        if r==r0 and t == r0:
            counter ++
            Go to next repetition
        if CW( r, u, t ) or (r,u,t are collinear and u is
         between r and t):
            u = t
        counter ++
    Remove point u from the unused_points_list.
    if u != r0:
        Append u at the end of the convex_hull_list
return convex_hull_list
```

Implementation Details

There are 3 things I'd like to point out in my implementation:

- In the function *ordered(point0, point1, point2)* the 3 points passed as arguments are supposed to be collinear. However, due to arithmetical precision issues:

$$\lambda_A = (point2.y - point0.y)/(point1.y - point0.y)$$

and

$$\lambda_B = (point2.x - point0.x)/(point1.x - point0.x)$$

may not be equal as they should be.

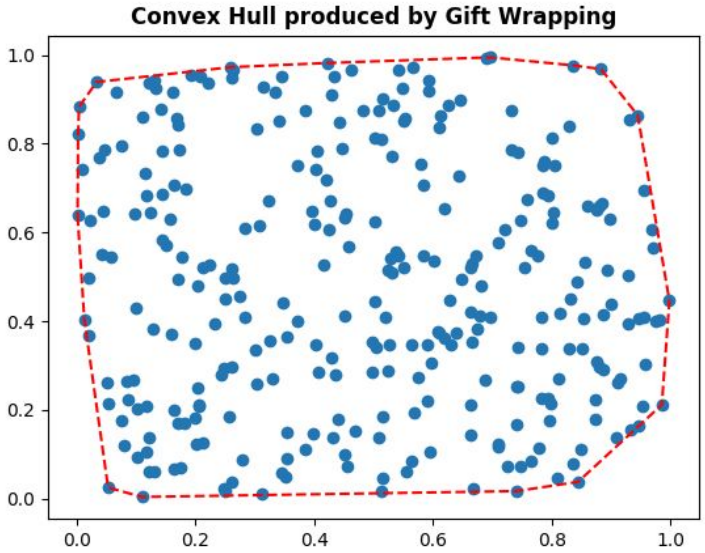
So to assert that the points passed as arguments are, in fact, collinear, I check whether:

$$|\lambda_A - \lambda_B| < 10^{-15}$$

where 10^{-15} is the upper bound of the acceptable precision error.

- Instead of using a list for the `unused_points_list` of the pseudocode, a set is used instead. That is because a deletion in a set has time complexity O(1) compared to O(n) for a deletion in a list.

- An improvement compared to the pseudocode is that in one iteration more points than one may be removed. That's the case when the new point of the convex hull has one or more collinear points.



References

- Lecture slides
- http://geomalgorithms.com/a15-_tangents.html