1. **Properties of Minkowski Sums and Euler’s theorem:**

* Given sets A, B and C formally prove that =

By definition, ,

We’ll prove this equality by showing double inclusion:

We’ll show that: :

Let there be , there exist such that

Since it holds that .

If :

If :

In any case, we’ve shown:

We’ll show that: :

Let there be ,

If then and therefore,

If then and therefore,

We’ve shown that for every , it holds that:

. Therefore,

From set theory we can infer that due to this double inclusion:

= as required.

* What is the Minkowski sum (what geometric object and what can you say about it) of

1. Two points?

Let there be two points, : which is another point represented by their vector sum.

1. point and a line?

Let there be a point and a line :

Where is a parallel identical line to L, where each of its points are shifted by the coordinates of a.

1. Two lines segments (think of all possible cases)?

Let there be a line and a line :

If then which is another line.

If then which is a parallelogram.

First, lets examine that there exists a plane which is parallel to both lines. (where d=2, this is trivial, when d>2, we can find this plane by getting the perpendicular plane to )  
observation: that rests entirely on a plane parallel to both lines.

Proof: lets choose a parallel plane to both lines.  
every point in is a set distance from this plane, likewise every point in is distance.  
This means, is in a set distance from this plane.

1. Two Disks?

Let there be a disk and a disk : .

Let there be: is another disk that was shifted by the coordinates of . In the previous part, we proved that:

= therefore, it holds that:

which is another disk with a radius equal to .

* Recall that for the proof of Lemma. 6.2 (complexity of a trapezoidal map) we used the property that in a planar graph we have that

Here E and V are the number of edges and vertices in a planar graph, respectively. Prove Eq. 1 (you can assume that V ≥3).

1. **Exact Motion Planning for a Diamond-Shaped Robot:**
   * 1. **Preprocessing phase (1)—constructing the C-space:**

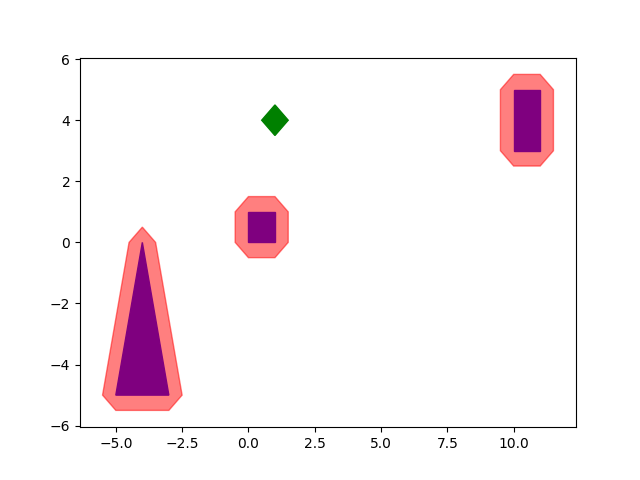
Assumptions**:**

1. All obstacles are convex. ( has vertices)
2. our robot is rotated square-shaped robot . (a convex polygon with vertices)

Computational Complexity:

Since our robot is a rotated square, we have a convex polygon with vertices. Let be an obstacle with vertices, we receive that the time complexity of the Minkowski-sum of those 2 convex polygons is .   
This complexity can be seen in our implementation in the while loop where we iterate over both the obstacle coordinates with and the robot coordinates with , each time advancing by one. Such that in the worst case it would take iterations to finish iterating over the lists in order to compute the sum.

For the specific data that was provided in the homework, we received the following visualization:



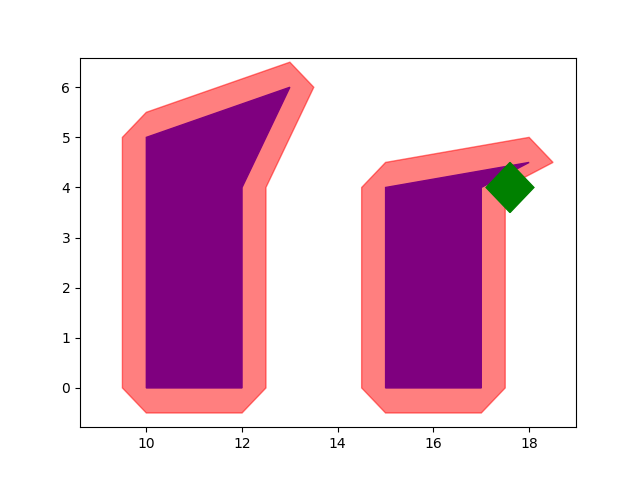
How can non-convex obstacles affect the results? Why? When? Support your claim with examples.

Taking the Minkowski-sum of non-convex obstacles first of all raises the complexity of the resulting shape:

such that , .

Therefore, our current algorithm is incapable of computing the correct Minkowski-sum, simply from the complexity standpoint of the loop.

If we attempt to input non-convex obstacles, it will result in broken shapes in our current algorithm. Our current algorithm doesn’t work with non-convex obstacles because the assumption that the angle between the positive x-axis and the next line in the polygon is always increasing (when we start from the point with the lowest y-coordinate, and we run counter-clockwise on the vertices). However, in concave obstacles this is not true because there could be 2 consecutive lines where the angle decreases between them.

The resulting Minkowski-sum will be incorrect, if while iterating the edges of the robot and the obstacle, the robot needs to ‘return to an edge it has already past’. More precisely, this happens when an obstacle edge which has a larger X-axis angle than an edge of the robot, is followed by an edge that has a less sharp angle than the robot edge. For instance:  


In the right concave obstacle, our algorithm runs from an edge with a sharper angle (90 deg) than the robots (45 deg), to a less sharp one(30 deg).  
if we place the robot in a seemingly free configuration, we erroneously get a collision.  
In the left example, the algorithm would run from the second edge (90 deg) to the third one (65 deg). Although this object is concave, the minkowski sum we’ve calculated for it is still correct!

* + 1. **Preprocessing phase (2)—building the visibility diagram:**

Assumptions**:**

1. is the overall number of vertices.

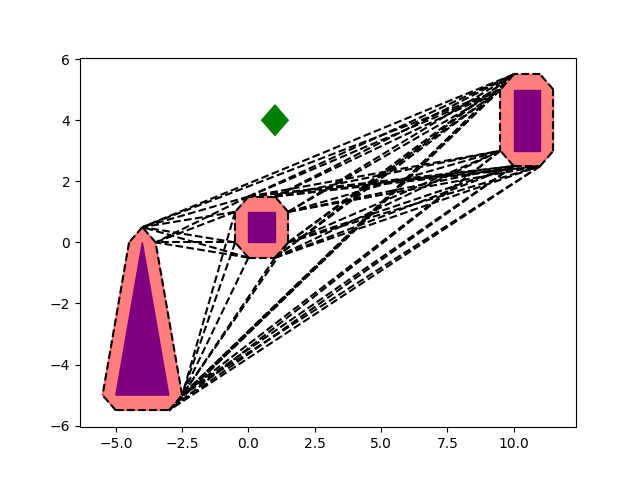
Computational Complexity:

In our implementation of building the visibility graph,, we have a time complexity of because we have a nested for loop where we check for every 2 vertices whether they are visible from each other or not. We know there are pairs of points we need to check because for every vertex it could be connected to any other vertex, and to check visibility requires time in our implementation because we need to check whether the line segment connecting intersects any of the obstacles, therefore because we perform an operation that requires computation time because we iterate over all the obstacles and determine whether the line segment connecting intersects any of the obstacles.

We perform an operation that takes computation time, different times. Therefore, we receive an overall complexity of for our algorithm.

Note: we saw in class that there exists an implementation in but we chose to implement the naïve implementation for its code simplicity.

For the specific data that was provided in the homework, we received the following visualization:



* + 1. **Query phase—computing shortest paths:**

Assumptions**:**

1. We have a list of the Line-Strings that constitute the visibility graph,.

Computational Complexity:

To compute the shortest path, we first need to build a graph we can run Dijkstra’s algorithm on.

Building the graph, , takes time because we iterate over the line segments from the visibility graph which could reach a number of , and for each line we add its end points as vertices to the new graph with a non-directed edge connecting them with a weight equal to the length of the line-segment in time.

After building the graph we can run Dijkstra’s algorithm on it which takes up to and if we plugin , we receive that Dijkstra’s algorithm finishes in . Therefore, the overall complexity of computing the shortest-path given the Line-Strings of the visibility graph is: .

For the specific data that was provided in the homework, we received the following visualization:

