Proyecto Fin de Carrera
Grado en Ingeniería de Tecnologías Industriales

A path planning algorithm for a marsupial robotic system (Algoritmo de planificación de trayectorias para un sistema robótico marsupial)

Autor: Adam Rodríguez González

Tutor: José Miguel Díaz Báñez

Dpto. de matemática aplicada II
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2023

DEPARTAMENTO DE
MATEMÁTICA APLICADA II

Proyecto Fin de Carrera
Grado en Ingeniería de Tecnologías Industriales

A path planning algorithm for a marsupial robotic system (Algoritmo de planificación de trayectorias para un sistema robótico marsupial)

Autor:
Adam Rodríguez González

Tutor:
José Miguel Díaz Báñez
Catedrático de universidad

Dpto. de matemática aplicada II
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2023

Proyecto Fin de Carrera:	A path planning algorithm for a marsupial robotic system (Algoritmo de planificación de trayectorias para un sistema robótico marsupial)

Autor:	Adam Rodríguez González
Tutor:	José Miguel Díaz Báñez

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Agradecimientos

A mis padres por ayudarme en todo lo que han podido. A mi familia, por proporcionarme bastantes momentos de alegría y diversión. En especial a mi hermano, que siempre está conmigo y compartimos la vida. A mi abuela, que ha hecho todo lo que ha estado en su mano por estar conmigo día a día y ha escuchado todos los problemas a los que me he enfrentado durante estos años de universidad sin entender ni la mitad.

A mi perra a la que parece que le falta medio cerebro y sus pocas neuronas no son capaces de entender más de dos palabras, y a mi gato Mino, que no para de pisarme el teclado mientras escri/O?Eo7w47ovr3nt.

Resumen

El uso de UAVs y AGVs durante los últimos años ha permitido disminuir el número de tareas potencialmente peligrosas que debían ser realizadas por humanos. Las tareas de inspección que deben realizar son extensas en el tiempo, por lo que se recurre al uso de drones marsupial. En este trabajo se desarrolla un algoritmo capaz de encontrar la posición en la que debe colocarse la parte terrestre del dron marsupial para poder desplegar a la aérea sin producir colisiones con el entorno. Para lograr este objetivo, el entorno se divide en planos bidimensionales en los que se estudia la viabilidad de la posición, evitando colisiones en el vehículo terrestre, el aéreo y el cable de alimentación que permite aumentar la autonomía. Conocidos el punto inicial y el punto objetivo, así como el entorno, el algoritmo calcula una solución válida en los planos que permitan encontrar la solución de mínimo desplazamiento en el menor tiempo posible. Una vez lograda esta función, se introducen cambios en el comportamiento del algoritmo, los parámetros del sistema y la definición del entorno para estudiar los efectos que produce en la eficiencia del algoritmo y cómo estos pueden ser subsanados en el caso de que no sean beneficiosos.

Abstract

The use of UAVs and AGVs in recent years has reduced the number of potentially dangerous tasks to be performed by humans. The inspection tasks they have to perform are extensive in time, so the use of marsupial drones is resorted to. In this work, an algorithm that is able to find the position where the ground part of the marsupial drone should be placed is developed, in order to deploy the aerial part without causing collisions with the environment. To achieve this objective, the environment is divided into two-dimensional planes in which the viability of the position is studied, avoiding collisions in the terrestrial vehicle, the aerial vehicle and the power cable that allows the autonomy to be increased. Knowing the initial point and the target point, as well as the environment, the algorithm calculates a valid solution in the planes that allow the minimum displacement solution to be found in the shortest possible time. Once this function has been achieved, changes are made to the behaviour of the algorithm, the system parameters and the definition of the environment to study the effects on the efficiency of the algorithm and how these can be remedied if they are not beneficial.

Abbreviated Index

Contents

List of Figures

List of Tables

1 Introduction

1.1 Problem description and motivation

The goal of this project is to develop an algorithm capable of enabling the arrival of a marsupial drone from an initial point to a target point on a known obstacle map. This technology allows reducing the danger to which a worker must be exposed and replacing it with autonomous or remotely controlled drones, which are able to obtain more features of the building in a shorter time and have greater manoeuvrability than the worker (see Figure 3.35).

Drones simplify the work to be done, but their autonomy (tens of minutes) is not enough to do the job without having to use a new drone to replace the old one or wait for the battery to be recharged.

The solution to the problem of autonomy, which greatly reduces the applicability of drones, is the use of marsupial drones. A marsupial drone is a technology that began to be developed in the 1990s [1] and consists of making use of a team of drones: a ground vehicle and an aerial vehicle (see Figure 1.2). The ground vehicle will be able to transport the aerial vehicle, allow the aerial vehicle to be deployed, and continuously power it through the use of a wire linking the two vehicles, which solves the problem of autonomy.

Although using a marsupial drone solves the time of use of the aerial vehicle, it introduces a new problem in the inspection, the wire. The wire that connects the two vehicles is a source of collisions with obstacles in the terrain to be inspected and limits the range of the aerial vehicle with respect to the position of the ground

Figure 1.1 Worker and drone in building inspection.

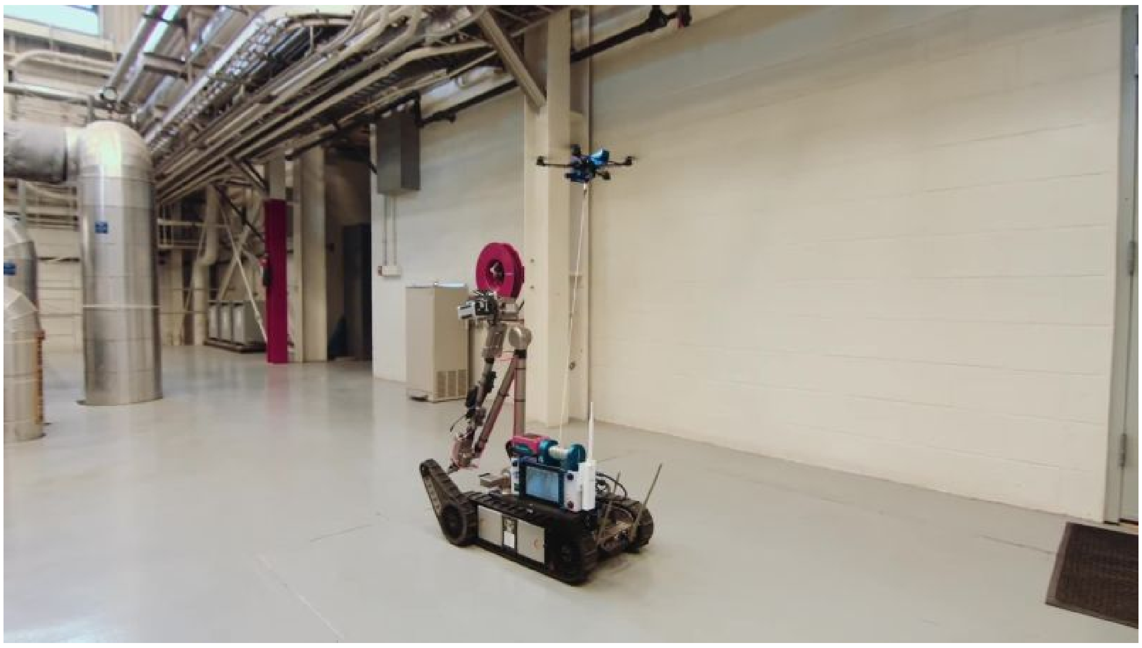


Figure 1.2 Marsupial drone.

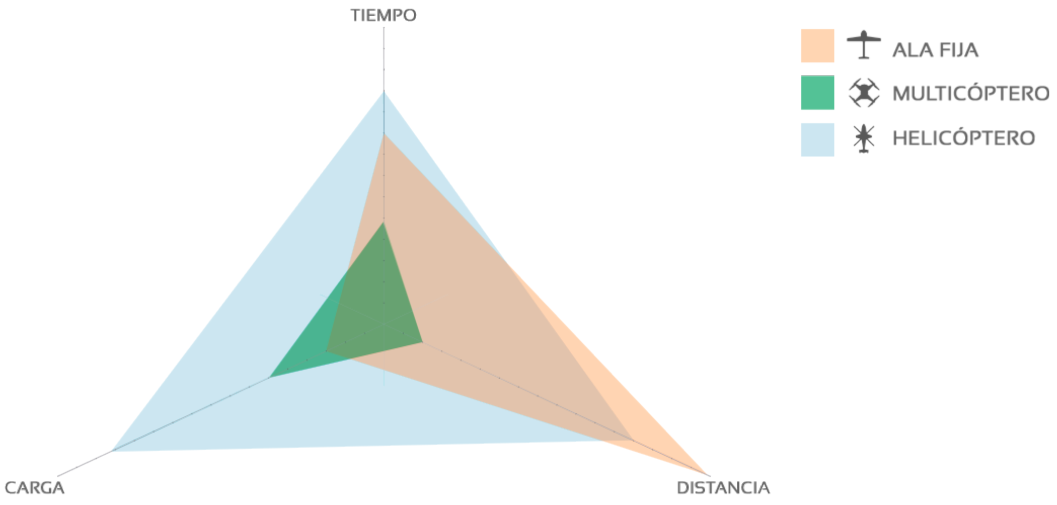


Figure 1.3 Aerial vehicle characteristics.

vehicle. In this study, it will be assumed that it is possible to change the length of this wire to a proportionate maximum length.

Depending on the type of aerial vehicle, a number of features are available that make it stand out from other types of configurations (see Figure 3.38). The most useful type of aerial drone in this case is the helicopter type, but it allows less maneuverability than the multirotor. If a marsupial drone is chosen, the time of use can be increased and the distance of travel will be limited by the power wire. It makes more sense to use a multirotor, which are easier to maneuver and cheaper to build.

Once a configuration has been chosen which allows inspection, an algorithm has to be developed to calculate the position of the ground vehicle which enables the aerial vehicle to be deployed to the target point while avoiding collisions between the aerial vehicle and the wire with obstacles in the environment. In other works, the ground vehicle remains static [16] and the focus is on the calculation of the time position of the tether and the aerial vehicle.

Configurations consisting of an unmanned ground vehicle (UGV) carrying the unmanned aerial vehicle (UAV) are becoming more noticeable in recent years. Some of the most noteworthy examples using this type of technology are Team CSIRO Data61 [11], the DARPA 2021 Subterranean Challenge [12] and, as an honourable mention, Mars 2020 rover and helicopter [13].

1.2 Related work

It should be noted that not all studies of marsupial robotic systems are based on the use of a hybrid UGV-UAV system, on which this study will be based. There are other typologies worthy of study, such as marsupial systems of unmanned underwater vehicles (UUVs), linked by a cable whose primary function is to serve both as a means of communication between vehicles and as a means of data transmission [14]. In other approaches, as [17], the marsupial system is formed by the aerial vehicle and a human, connected by a cable.

Another interesting use of this type of system is the use of the mooring as an aid in the exploration of difficult terrain or in maneuvers. In [15], the tether is attached from an UGV to an object in the environment, allowing for greater exploration capability. In [18], the authors propose the using of a rigid tether connected to an aerial vehicle to increase the stability in landing. The use of the tether can provide stability not only in landing, but in flights, as in [19]. In this study, the tether improves the stability of a UAV in confined spaces. In [22], a marsupial system formed by an UAV, an UGV and a tight tether, navigates through confined spaces using the UAV as a visual assistant.

This study does not take into account the trajectory that the aerial vehicle must follow once deployed, but rather considers that by following the path formed by the wire between the two vehicles and not colliding with any obstacle, the vehicle will be able to reach the target point. In [20], the aerial vehicle follows the same type of trajectory, but in this case, the cable is considered as a taut tether. Because of that, the UAV can only reach targets with a direct line of sight from the point it takes off. In order to solve this problem, in [21], the tether is modeled as a catenary, which allows to find solutions away from the direct line of sight, but the starting point of the tether still static.

Most of the studies developed are based on the deployment of the aerial vehicle and use Djikstra, A* and Theta* algorithms to avoid collisions [2], dividing the three-dimensional environment in a meshed way. There are also algorithms capable of calculating the areas accessible by each vehicle, as in [3], but they are not able to consider the case where both are joined, so it may help to find some initial position points, but it does not ensure an optimal solution search.

Studies of aerial robots that make use of wires are mostly focused on the control of these robots [4] and consider that the wire cannot cause collisions with the environment, which is not in line with the research conducted in this study, which is based on the influence of the wire and puts the vehicles to which it is anchored in the background. In [23], the approach considers the same marsupial system as this study, but the cable is a tight tether, only providing ray-tracing solutions, but with a dynamic system.

The most common trajectory algorithms calculate the path to be followed by the ground vehicle and constrain it by using the position in which the aerial must be placed and the length of the wire [5], but still do not consider collisions that occur between the wire and the environment.

As can be seen, although there are studies that make it possible to determine the position of the catenary that forms the wire between the two vehicles [6], this is not an aspect that is usually considered when carrying out the collision study and which can make many of the solutions found inaccessible. An approach that consider the same marsupial system, a tether modeled by a catenary and considers the collisions with the environment is [24], being the most similar to this study. In this case, the marsupial system moves in a coordinated way, unlike this study, where the UAV only moves when the UGV can provide a trajectory to the target.

1.3 Scope of this work

The collisions that exist between the catenary and the environment are very little developed in the academic work that can be found. This work seeks to find a point, or several of them, from which it is possible to deploy the aerial vehicle in the shortest possible time. Once the various points have been found, the ground vehicle must proceed to one of them, and trajectory tracking can be used. When it has reached the deployment point, a real-time system can be incorporated into the air vehicle in case there may be unanticipated obstacles, but from the outset, the wire will not cause any collisions.

The work is carried out in *Python* and by means of the available libraries, the aim is to achieve an algorithm capable of implementing the above. The work is developed as follows:

- Creation of the different elements necessary to build the environment and the catenary formed by the wire. The chapter 2 explains in detail the study to be carried out and the objects to be created. In addition, the way in which the obstacles are going to be found and how the map is different in each simulation will also be explained, in order to obtain unbiased data.
- In chapter 3 it will be explained how the created algorithms work, where there is a main algorithm that will call an auxiliary one as many times as necessary. These algorithms are in charge of finding a solution that allows the deployment of the aerial vehicle avoiding collisions of the aerial vehicle and the wire with obstacles in the shortest possible time.
- In order to obtain the best possible solutions and to observe how the different variables of the system affect the performance of the algorithms, in the chapter 3.2.3 the algorithms will have the fundamental parameters developed in the previous chapter modified, looking for the most efficient solution.
- To make use of more complex functionalities and new obstacle definitions, extensions to the initial problem are developed in chapter 4 and the results are studied.
- Once the most optimal possible algorithm has been obtained, different cases are studied in chapter 5, testing the correct operation in different environments.
- In chapter 7, the development of the work is discussed, the objectives achieved, the usefulness of the study is checked and the future applications it can have are considered.

2 Description of the problem under study

The problem under study is defined as follows: given an initial position of the marsupial system as a whole $A = (x_A, y_A, z_A)$, a target position $B = (x_B, y_B, z_B)$, where the air vehicle should be placed, the maximum length of the tether connecting the two vehicles L and a three-dimensional map made up of obstacles, find the nearest point of deployment $C = (x_C, y_C, z_A)$ to which the ground vehicle must travel in order to deploy the air vehicle, enabling it to reach the target point as well as the length of wire that allows such deployment while avoiding collisions with the obstacles.

The movement of the marsupial system will not take place simultaneously, but there will be a first move, in which the ground vehicle travels from A to C, and a second move, in which the aerial vehicle flies from C to B. To perform this function, the point C to be found in each configuration must be calculated from the target point, and once found, movement will be allowed.

Point C must satisfy the following conditions:

1. The position of C must be accessible to the marsupial system, that is, it cannot be positioned on an obstacle and there must be an accessible path between A and C.
2. The position of C must allow for a wire of less than the maximum specified length connecting points C and B without collision between surrounding obstacles and the wire.

Before developing the algorithm, it is necessary to know how some parameters that belong both to the marsupial drone and to the three-dimensional space behave and how they will be treated.

2.1 Treatment of obstacles and workspace

An obstacle is a set of three-dimensional points defined by three components:

1. Height.
2. Horizontal distance to the target point.
3. The angle it forms with respect to the union of the target point with the initial point.

We propose a strategy to obtain an approximate solution the optimization problem defined above. The approach considers a range of planes passing through the target point and perpendicular to the ground. Thus, it will be assumed that the obstacle points are given within a plane, as in Figure 2.1.

Each plane is defined by the angle it makes with respect to the plane perpendicular to the ground passing through the initial and target points.

The three-dimensional points of the obstacle must be translated into two-dimensional positions within a plane.

The workspace is divided into M planes with a centre at the target point, as shown in figure 2.1, and identified by a degree between 0º and 360º with respect to the plane joining the target point and the initial point.

To each plane corresponds a degree i, so that:

$$i = \frac{360 * K}{M}$$

where

$$K \in [0,M-1]$$

$$K \in Z$$

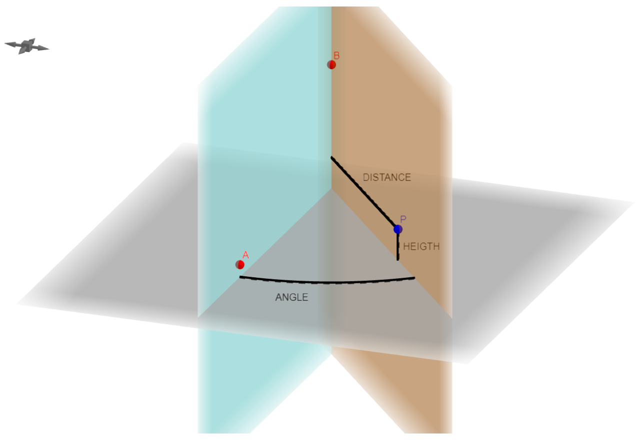


Figure 2.1 Planes and obstacle 1.

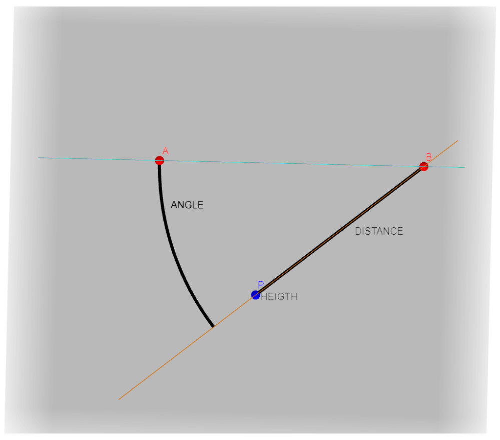


Figure 2.2 Planes and obstacle 2.

In both Figure 2.1 and Figure 2.2, P represents a point belonging to an obstacle, A is the starting point and B is the target point. It also highlights the three components necessary to define the obstacle point.

In order to perform this action, the drone will be considered to be equivalent to a sphere of radius R. Whenever any obstacle point is at a distance less than or equal to R with respect to one of the defined planes, the point will be included in the two-dimensional plane.

Once the plane to which the obstacle point belongs has been obtained, it is necessary to calculate its position within the plane.

To do this,it is necessary to calculate both the horizontal distance to the target point and the vertical distance to the ground. With this, a three-dimensional obstacle point is defined in the discretised space in planes, it would only remain to repeat this process for each given obstacle point.

In practice, the points defining the obstacles will not be obtained in this way, they will be given within the two-dimensional space of planes, but it will be considered possible to traduce the three-dimensional points into two-dimensional points as explained above.

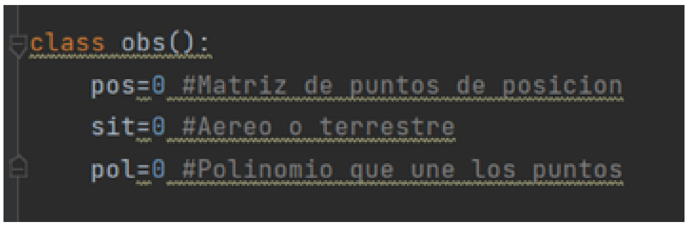
Furthermore, each obstacle point belonging to a plane is labeled by one of the following three categories:

1. **Ground**. A discrete set of points whose height is constant and considered to be zero.

2. **Land obstacles**. A discrete set of points at least one of which touches or is below the ground.
3. **Aerial obstacles**. A discrete set of points in which none of them touches or is below the ground.

As with obstacles, any three-dimensional point to be evaluated must be in a defined plane and, in turn, each plane will be defined by the following components: An aerial obstacle, a land obstacle, the ground and a degree, which defines the position of the plane.

Each obstacle category within a plan will be defined by the class **obs**, which implements the necessary parameters to determinate the collision between the wire and the obstacles.



```
class obs():
    pos=0 #Matriz de puntos de posicion
    sit=0 #Aereo o terrestre
    pol=0 #Polinomio que une los puntos
```

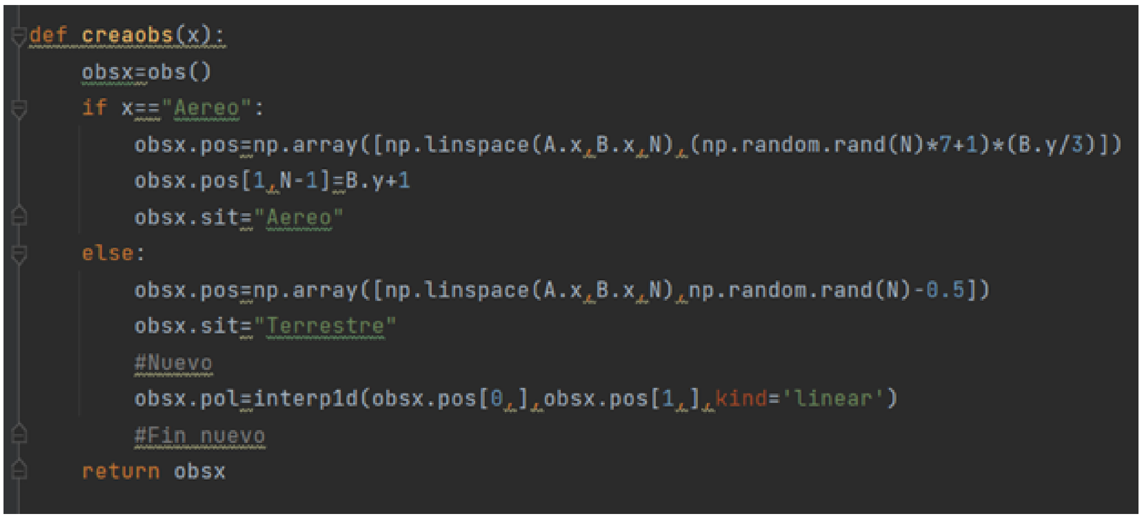
Figure 2.3 Class obs.

- **POS:** Matrix containing the values of the points that define the obstacle within its plane. The first row contains the horizontal position of the obstacle and the second row contains the vertical position.
- **SIT:** Situation of the obstacle.
  1. Aerial obstacle.
  2. Land obstacle.
- **POL:** Polynomial that interpolates the points defined by the matrix POS. It is necessary to determinate the intermediate values of heights between the defined obstacles points.

Two different types of obstacle maps will be used:

1. **Random obstacle maps:** The position matrix of the obstacle categories POS will be randomly generated. These workspaces will be used to test the correct working of the algorithm and its improvements.
2. **Complex obstacle maps:** They will be used to test the working of the algorithm in realistic workspaces.

In order to create the different obstacle categories found in each random plane, a specific function will be used. This function is **creaobs** (see Figure 2.4).



```
def creaobs(x):
    obsx=obs()
    if x=="Aereo":
        obsx.pos=np.array([np.linspace(A.x,B.x,N),(np.random.rand(N)*7+1)*(B.y/3)])
        obsx.pos[1,N-1]=B.y+1
        obsx.sit="Aereo"
    else:
        obsx.pos=np.array([np.linspace(A.x,B.x,N),np.random.rand(N)-0.5])
        obsx.sit="Terrestre"
        #Nuevo
        obsx.pol=interp1d(obsx.pos[0,],obsx.pos[1,],kind='linear')
        #Fin nuevo
    return obsx
```

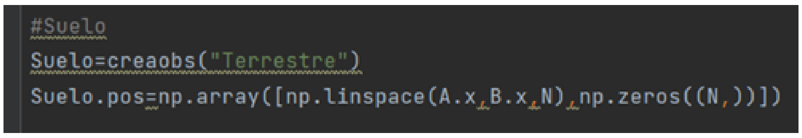
Figure 2.4 Function creaobs.

As it can be seen, each obstacle is created making use of a modifiable mean value and standard deviation. The standard deviation of the aerial obstacles is greater than standard deviation of the land obstacles, this is because the land obstacles must be traversed by the ground vehicle. Obstacles act like stalactites and stalagmites in a cave.

It is due to this form of obstacle creation that some land obstacle points may obtain negative height values. In that case, they should be treated as zero height obstacle points.

In addition, the target point must be accessible, so there cannot be an aerial obstacle below its height in its horizontal position.

The creation of the ground obstacles is much simpler than the other two types of obstacles, in this case, there are not a mean and a standard deviation value, the height will always be zero (see Figure 2.5).



```
#Suelo
Suelo=creaobs("Terrestre")
Suelo.pos=np.array([np.linspace(A.x,B.x,N),np.zeros((N,))])
```

Figure 2.5 Creation of ground obstacles.

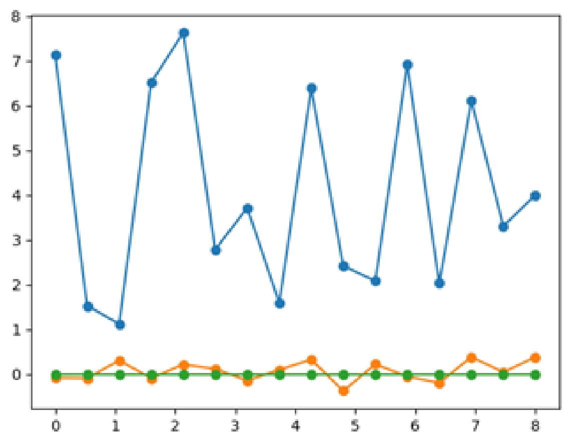


Figure 2.6 Obstacles example.

An example of the types of obstacles is shown in Figure 2.6. In this figure, the aerial obstacle is the blue curve, the land obstacle is orange and the ground is green.

As mentioned above, land obstacle points with height less than zero should be treated as zero height points.

The circles represent the defined obstacle points and the lines represent the intermediate values of heights of the polynomial.

2.2 Wire

As it is a physical object, gravity will act on the wire, which does not allow it to be implemented using a simple curve, so it is necessary to make use of the **catenary**, which is an approximation to a real wire, rope or chain making use of an ideal curve.

The curve is defined by the position of the initial and final points of the wire (mooring points) and the length of the curve (wire's length) and, as explained in the previous section, it will be contained in a plane. The wire's length can vary from a minimum length (to be determined later) to a maximum length, which is a initial available data. For different lengths, the catenary will have a larger or smaller fall, and this is what must be determined to avoid the wire colliding with obstacles.

As an example of how the implementation works in a two-dimensional plane, three catenaries of different lengths with the same mooring points are represented in Figure 2.7.

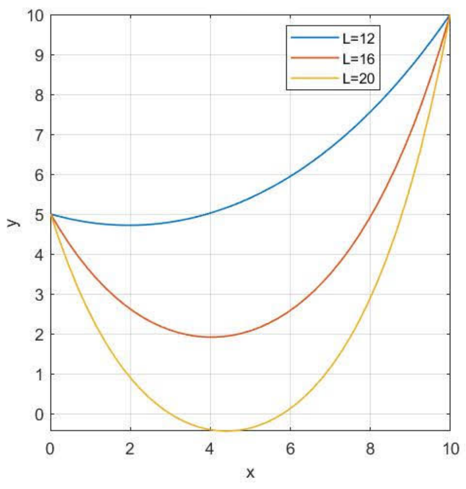
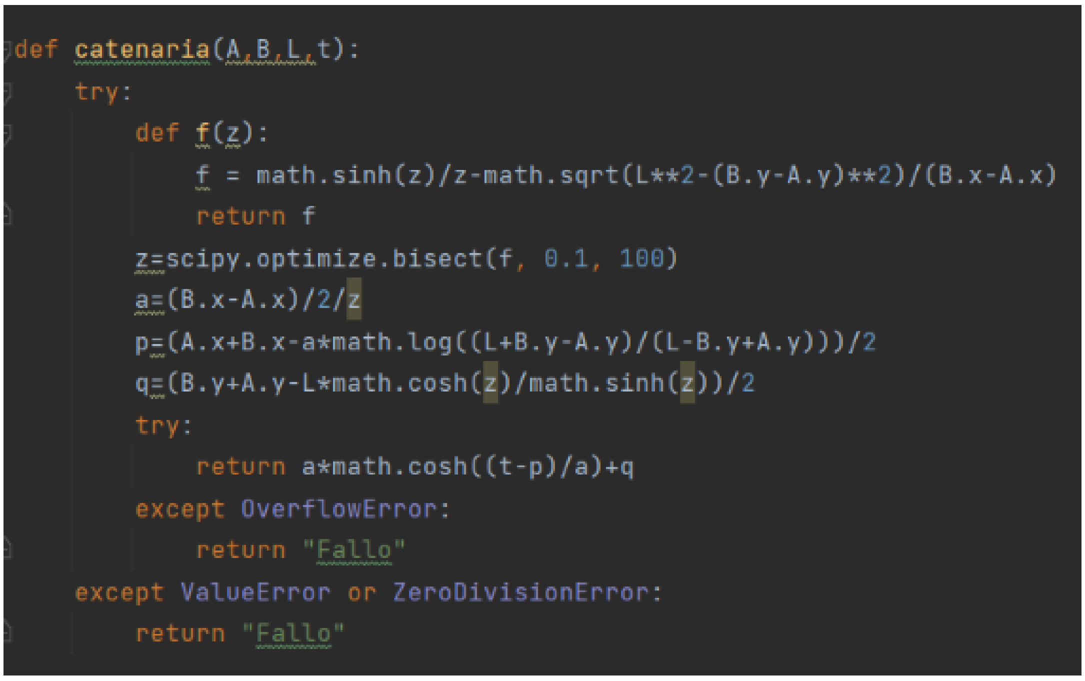


Figure 2.7 Catenaries of different lengths.

It can be seen how, although the mooring points of the catenary are the same, the length creates a completely different curve.

The function that implements the catenary is defined as **catenaria** and defines the curve that forms a catenary that connects two points with a provided length, returning the value of the curve at point **t**, any point at a distance between the mooring points (see Figure 2.8).



```
def catenaria(A,B,L,t):
    try:
        def f(z):
            f = math.sinh(z)/z-math.sqrt(L**2-(B.y-A.y)**2)/(B.x-A.x)
            return f
        z=scipy.optimize.bisect(f, 0.1, 100)
        a=(B.x-A.x)/2/z
        p=(A.x+B.x-a*math.log((L+B.y-A.y)/(L-B.y+A.y)))/2
        q=(B.y+A.y-L*math.cosh(z)/math.sinh(z))/2
        try:
            return a*math.cosh((t-p)/a)+q
        except OverflowError:
            return "Fallo"
    except ValueError or ZeroDivisionError:
        return "Fallo"
```

Figure 2.8 Function that implements the catenary.

In case it is not possible to obtain a curve implementing the catenary with the data provided, the function will return an error and the algorithms that make use of the function must interpret these data as an invalid solution.

3 Algorithms

Two algorithms will be used to solve the problem: the main algorithm, which will be in charge of traversing the three-dimensional space finding a free-collision catanery and the auxiliary algorithm, to solve the local problem in a two-dimensional space, being called by the main algorithm every time it is necessary.

3.1 Auxiliary algorithm

Given a plane and obstacle points belonging to a plane, the objective is to calculate the point C that allows the deployment of a collision free catenary connecting the point C and the target point B so that the distance between C and the initial point A is minimum.

- **Input:** A, B and obstacle points in the plane.
- **Output:** Horizontal distance between A and the point C.

The output of the algorithm will be an approximation of the optimization problem In our approach, both the horizontal axis of the plane and the catenary's length will be defined by a discrete set of predetermined values.

Figure 3.1 is an example of a plane in which a catenary has been deployed from point C to the target point. Before explaining the steps to be followed by the algorithm, it is necessary to define some parameters.

- A_x and B_x are the horizontal positions of the initial and target points, respectively. A_y and B_y are the vertical positions.

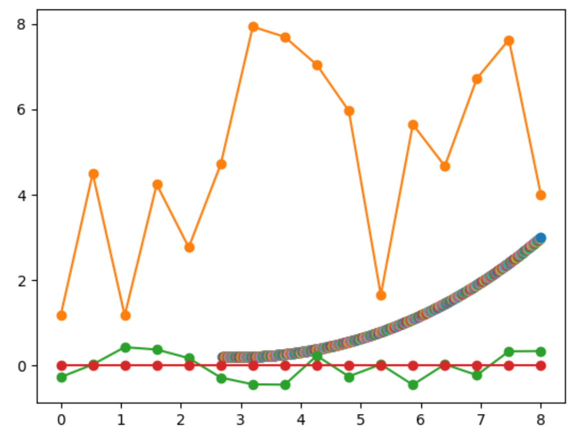


Figure 3.1 Example of a two dimensional instance.

- X_{min} is the first point to the right of A_x at which there is a (non necessarily collision free) catenary connecting that point and B.
- l_{max} and l_{min} are the maximum and minimum catenary's lengths.

The detailed explanation of how each parameter is calculated will be described later. The steps to be followed by the algorithm are the following:

- **Step 1:** Generate the range of horizontal positions $[X_{min}, B_x]$ by calculating the value of X_{min}.
- **Step 2:** Uniformly discretise the interval $I = [X_{min}, B_x]$ into h position points.
- **Step 3:** Move to point a of I closest to A_x that has not been evaluated.
- **Step 4:** Calculate $l_{min}(a)$, corresponding to the position of the point to be evaluated.
- **Step 5:** Generate the range of catenary's lengths $[l_{min}, l_{max}]$.
- **Step 6:** Uniformly discretise the interval $[l_{min}, l_{max}]$ into p length values.
- **Step 7:** Test all catenary's lengths to be evaluated and check for solutions, adjusting the value of l_{max} in the process.
- **Step 8:** If a collision free solution is found, the value of the evaluated position point (point C) is stored and the algorithm is finished. If no solution is found in the length's range, go to step 3.

3.1.1 Detailed description of the algorithm

Position points

The first parameter to be calculated, X_{min}, is the first location which allows a possible solution. The value of X_{min} is determined by the maximum catenary's length, the shorter the catenary's length, the closer X_{min} will be to B_x.

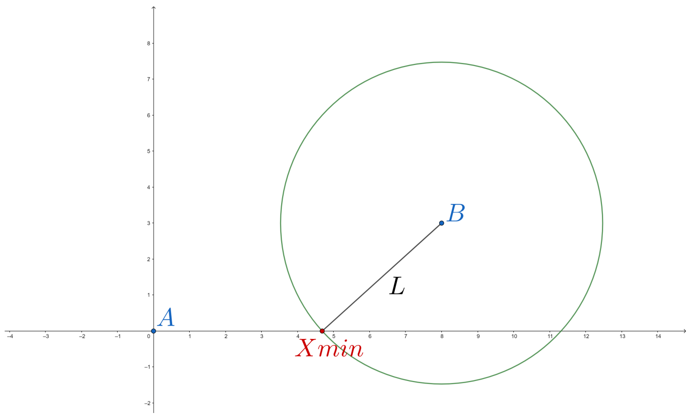


Figure 3.2 Minimum distance by length.

In Figure 3.2 if the target point is B and the maximum catenary's length is L, the minimum distance where the aerial vehicle can be deployed is the intersection of the circumference of radius L and centre B with the straight line on the ground (D). This way, there can be zero, one or two solutions.

1. $L < B_y$. There is no solution for X_{min}. It is to be assumed that this case will not take place in practice.
2. $L = B_y$. There is only one solution: $X_{min} = B_x$, and the catenary must be an straight line.
3. $L > B_y$. There are two solutions, the point nearest to A is chosen.

$$L = \sqrt{(B_x - X_{min})^2 + (B_y)^2}$$

$$L^2 = (B_x - X_{min})^2 + (B_y)^2$$

$$B_x - X_{min} = \sqrt{(L^2 - (B_y)^2)}$$

$$X_{min} = B_x - \sqrt{(L^2 - (B_y)^2)}$$

4. $L >> B_y$. If the solution calculated before returns a negative value, $X_{min} = A_x$.

Additionally, note that if there exists a low aerial obstacle near to B, it can limit the number of valid solutions far from that obstacle point. In this case, we consider the length of two line segments of lengths L_1 and L_2, where $L_1 + L_2 = L$. In this case, the length of the line connecting the points B and the lowest point of the obstacle (P) must be calculated, say L_1. Once obtained L_1, X_{min} is determined by the circle centered at P of radius $L_2 = L - L_1$.

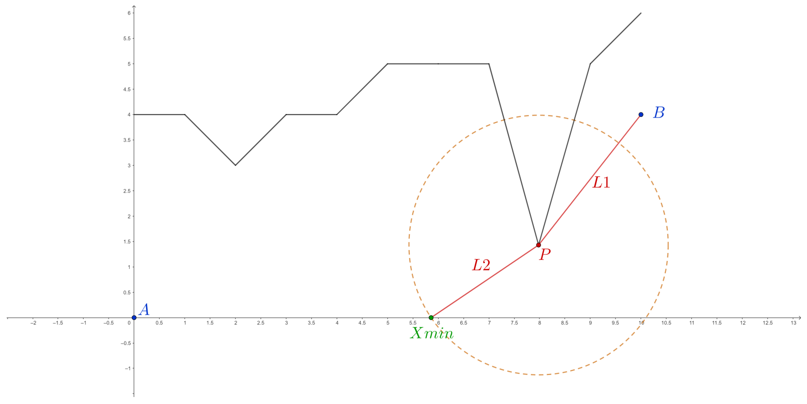


Figure 3.3 Minimum distance by aerial obstacle.

$$L_1 + L_2 = L$$

$$L_1 = \sqrt{((B_y - P_y)^2 + (B_x - P_x)^2)}$$

$$L_2 = L - \sqrt{((B_y - P_y)^2 + (B_x - P_x)^2)} = \sqrt{((P_y)^2 + (P_x - C_x)^2)}$$

$$(P_y)^2 + (P_x - C_x)^2 = L^2 - 2L\sqrt{((B_y - P_y)^2 + (B_x - P_x)^2)} + (B_y - P_y)^2 + (B_x - P_x)^2$$

$$X_1 = -2P_x$$

$$X_2 = (P_y)^2 + (P_x)^2(-L)^2 + 2L\sqrt{((B_y - P_y)^2 + (B_x - P_x)^2)} - (B_y - P_y)^2 - (B_x - P_x)^2$$

$$(C_x)^2 + X_1 * C_x + X_2 = 0$$

$$C_x = \frac{-X_1 \pm \sqrt{((X_1)^2 - 4 * X_2))}}{2}$$

$$X_{min} = \frac{1}{2} * (-X_1 - \sqrt{((X_1)^2 - 4 * X_2))}$$

To recapitulate, there are three solutions to choose as the initial search point: A_x, the point of maximum catenary's length, which may return an invalid negative position, and the point calculated from an aerial obstacles, which may return no solution.

The algorithm selects the maximum value of the three independent solutions.

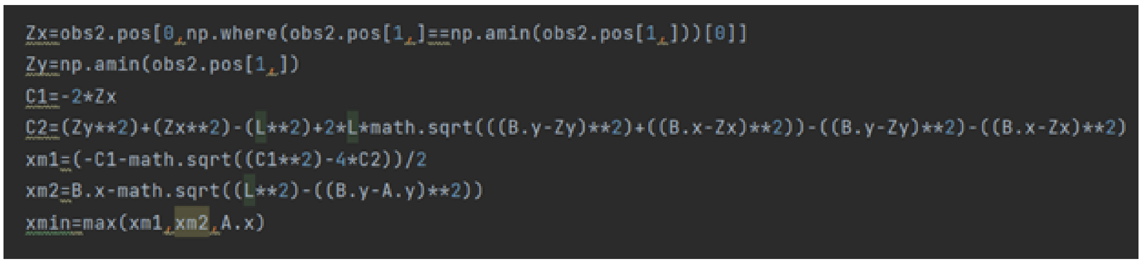
$$X_{min} = MAX\{X_{min}(1), X_{min}(2), X_{min}(3)\}$$

where:

$$X_{min}(1) = A_x$$

$$X_{min}(2) = B_x - \sqrt{(L^2 - (B_y)^2)}$$

$$X_{min}(3) = \frac{1}{2} * (-X_1 - \sqrt{((X_1)^2 - 4 * X_2))}$$



```
Zx=obs2.pos[0,np.where(obs2.pos[1,]==np.amin(obs2.pos[1,]))[0]]
Zy=np.amin(obs2.pos[1,])
C1=-2*Zx
C2=(Zy**2)+(Zx**2)-(L**2)+2*L*math.sqrt(((B.y-Zy)**2)+((B.x-Zx)**2))-((B.y-Zy)**2)-((B.x-Zx)**2)
xm1=(-C1-math.sqrt((C1**2)-4*C2))/2
xm2=B.x-math.sqrt((L**2)-((B.y-A.y)**2))
xmin=max(xm1,xm2,A.x)
```

Figure 3.4 Implementation of the initial point.

In the code of Figure 3.4, Z_x and Z_y act as P_x and P_y, the lowest point of the aerial obstacle in the plane. C_1 and C_2 are the constants of the quadratic equation. X_{m1} is the solution of the third method, X_{m2} is the solution of the second method and A_x is the solution of the first method.

Catenary's length

This is one of the most important parameters of the proposed approach. As with the position, there are restrictions, but they are much simpler.

1. The length must be in the interval $[l_{min}, l_{max}]$.
2. The length can not exceed L.

The values of l_{min} and l_{max} are updated at each position point. The value of l_{min} is defined as:

$$l_{min} = \sqrt{((B_x - X_{ix})^2 + (B_y - X_{iy})^2)}$$

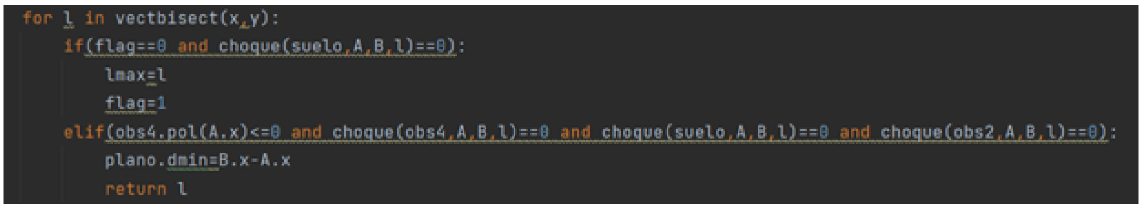
where $X_i = (X_{ix}, X_{iy})$ is the evaluated point.

l_{max} is calculated as follows. A first estimation is to choose the maximum length allowed by the user L, but it will only work at large distances from the target points ($L \approx B_x$). Once the points are closer, the length values near to L will not be able to solve the collision problem (specifically with land and ground obstacles). Thus, we tune l_{max} at every step.

For the first position point, the value of l_{max} is obtained using the following procedure:

1. The maximum permitted length L is set as l_{max}.
2. The algorithm enters the decrementing length loop.
3. The first length that does not collide with the ground is stored as l_{max} for the next step.

In this way, Note that l_{max} will decrease as the points get closer and cases impossible to solve will not be calculated. In addition, l_{max} can not exceed L, fulfilling the second restriction.



```
for l in vectbisect(x,y):
    if(flag==0 and choque(suelo,A,B,l)==0):
        lmax=l
        flag=1
    elif(obs4.pol(A.x)<=0 and choque(obs4,A,B,l)==0 and choque(suelo,A,B,l)==0 and choque(obs2,A,B,l)==0):
        plano.dmin=B.x-A.x
        return l
```

Figure 3.5 Lmax and length loop.

Once obtained the values of l_{min} and l_{max}, the next question is how to traverse the interval $[l_{min}, l_{max}]$. Since we do not know if the solution will be close to l_{max} or l_{min}, the values to be tested will "jump" from values close to l_{max} to values close to l_{min} starting at the middle point.

When a catenary's length avoids collision with obstacles, the solution C_x (distance from the origin to point X_i) will be saved (see Figure 3.5) and returned for future calculations with the **Main Algorithm**.

Number of positions and lengths

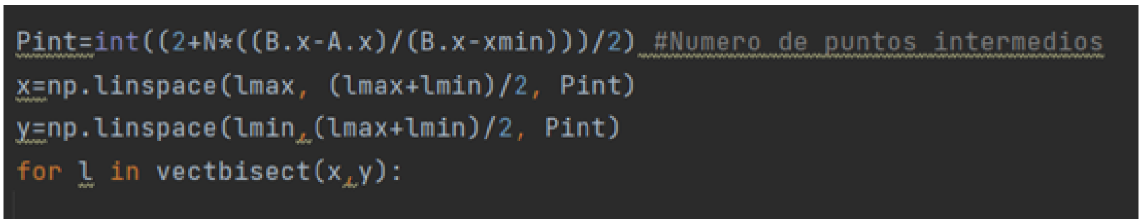
The number of position points and lengths will be experimentally studied in Section 3.2.3. At the this point, we take starting values.

The initial value of number of position points will be the number of obstacle points of the largest obstacle between the straight lines $x = A_x$ and $x = B_x$. Set this number as N.

Apart from that, the number of lengths will be variable depending on the position. The closer the search position is to $x = B_x$, the closer the values of l_{max} and l_{min} will be. In order to prevent similar searches, an initial value for the number of lengths is the following:

$$2 + N * \frac{(B_x - X_i)}{(B_x - X_{min})},$$

which decreases as the position point moves away from X_{min}.



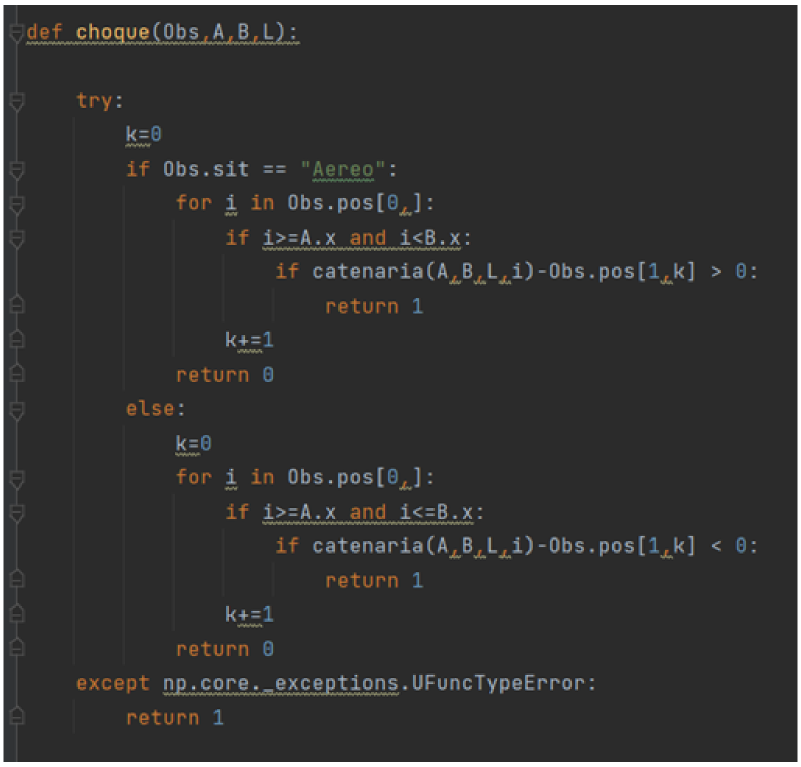
```
Pint=int((2+N*((B.x-A.x)/(B.x-xmin)))/2) #Numero de puntos intermedios
x=np.linspace(lmax, (lmax+lmin)/2, Pint)
y=np.linspace(lmin,(lmax+lmin)/2, Pint)
for l in vectbisect(x,y):
```

Figure 3.6 Catenary's lengths.

Collision with obstacles

The catenary has to avoid collision with obstacles. In the same two-dimensional plane, it is necessary to check that each point defining the aerial obstacle is above the catenary and those defining the terrestrial obstacles are below. Once a point that does not verify this rule is found, the catenary deployed is invalid.

To implement this simple algorithm, the function **choque** is created (see Figure 3.7). This function acts as a module of the **auxiliary algorithm**.



```
def choque(Obs,A,B,L):

    try:
        k=0
        if Obs.sit == "Aereo":
            for i in Obs.pos[0,]:
                if i>=A.x and i<B.x:
                    if catenaria(A,B,L,i)-Obs.pos[1,k] > 0:
                        return 1
                k+=1
            return 0
        else:
            k=0
            for i in Obs.pos[0,]:
                if i>=A.x and i<=B.x:
                    if catenaria(A,B,L,i)-Obs.pos[1,k] < 0:
                        return 1
                k+=1
            return 0
    except np.core._exceptions.UFuncTypeError:
        return 1
```

Figure 3.7 Function choque.

The function needs the following parameters that act as local variables:

- **Obs:** Matrix which stores the coordinates of the obstacle points.
- **A:** Ground vehicle's position.
- **B:** Target point.
- **L:** Wire's length.

The algorithm will evaluate every obstacle point between the tie points of the catenary. The collision will be determined by subtracting the height of the catenary with respect the height of the obstacle point. Depending on the type of obstacle, the value of the subtraction will be:

- Greater than zero for terrestrial obstacles.
- Less than zero for aerial obstacles.

If it exists at least one obstacle point which does not verify these conditions, the catenary will collide with the environment and it is not a valid solution for the auxiliary algorithm.
In order to check the correct functioning of the algorithm, a test will be done following the next steps:

1. Create a random aerial obstacle, a random land obstacle and the ground.
2. Determinate if exists a collision between the wire and the obstacles individually.
3. Determinate if it is a valid solution for the auxiliary algorithm.

Table 3.1 is created using the results of the test with some lengths between 5 and 10 metres. Furthermore, in order to verify these results, Figure 3.8 shows a visual representation of each try.

Table 3.1 Wire's length and collisions.

Wire's length	Aerial collision	Land collision	Ground collision	Solution
5	YES	NO	NO	NO
6.25	YES	NO	NO	NO
7.5	NO	YES	YES	NO
8.75	NO	YES	YES	NO
10	NO	YES	YES	NO

The code is able to interpret the information of the different types of existing obstacles and identify the collision with the catenary. With this information, it is possible to start searching the optimal position point of the ground vehicle.

3.1.2 Processing time

$$T_{aux} = O(h * n * p)$$

where:
h = Number of position points.
n = Number of obstacle points.
p = Number of catenary's lengths.

Both h and p are user-defined variables, while n is an immutable input, thus the auxiliary algorithm scales linearly with respect to the obstacle points.

3.1.3 Auxiliary algorithm flowchart

In order to understand the functioning of the algorithm in a more visual way, the flowchart is illustated in Figure 5.2:

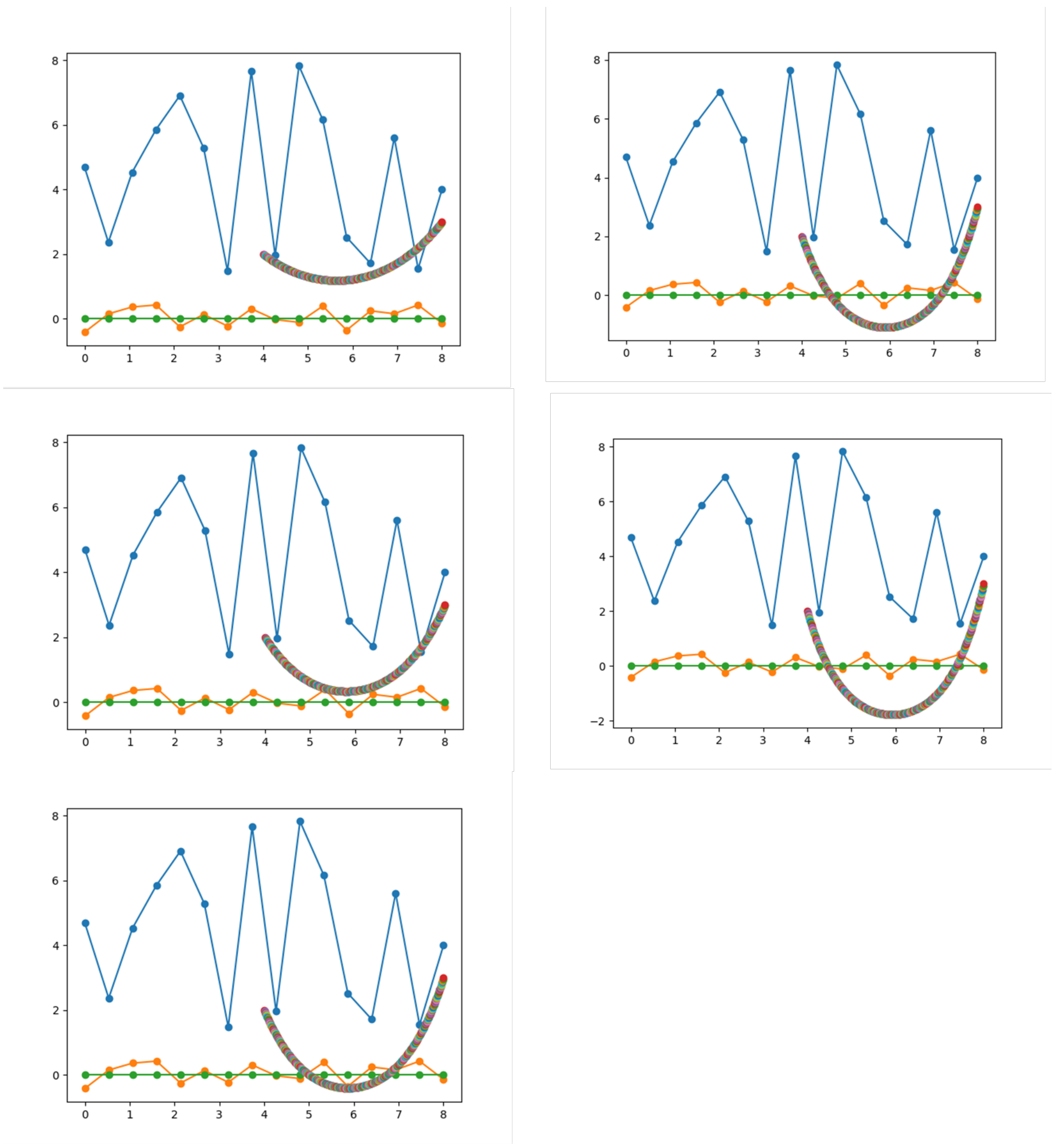


Figure 3.8 Representation of obstacles and wire.

3.2 Main algorithm

The problem is as follows. Given an initial position of the marsupial system as a whole, a target position, a maximum length of catenary and a three-dimensional workspace, find the nearest point of deployment to which the ground vehicle must travel in order to deploy the air vehicle, avoiding collisions with the obstacles.

The three-dimensional space is discretised using two-dimensional planes as explained in Section 3.1 and then the auxiliary algorithm is applied to every plane. Let $\pi(\alpha_j)$, $j = 1, \cdots, k$ be a bundle of planes passing through B and perpendicular to the ground and $\pi^*(\alpha_j)$ be the set of points C_j on the ground in $\pi(\alpha_j)$ so that there exists a free collision catenary ζ_j connecting C_j and B.

The overview of the main algorithm is as follows:

- **Input:** A, B, planes $\pi(\alpha_j)$ and the maximum length L.
- **Output:** $C_i \in \pi^*(\alpha_i)$ such that $d(A,C_i) = min_{C_j \in \pi^*(\alpha_j)} d(A,C_j)$.

1. Find the solution C_0 in $\pi^*(\alpha_0)$, where $\pi(\alpha_0)$ is the plane passing through points A and B and perpendicular to the ground.

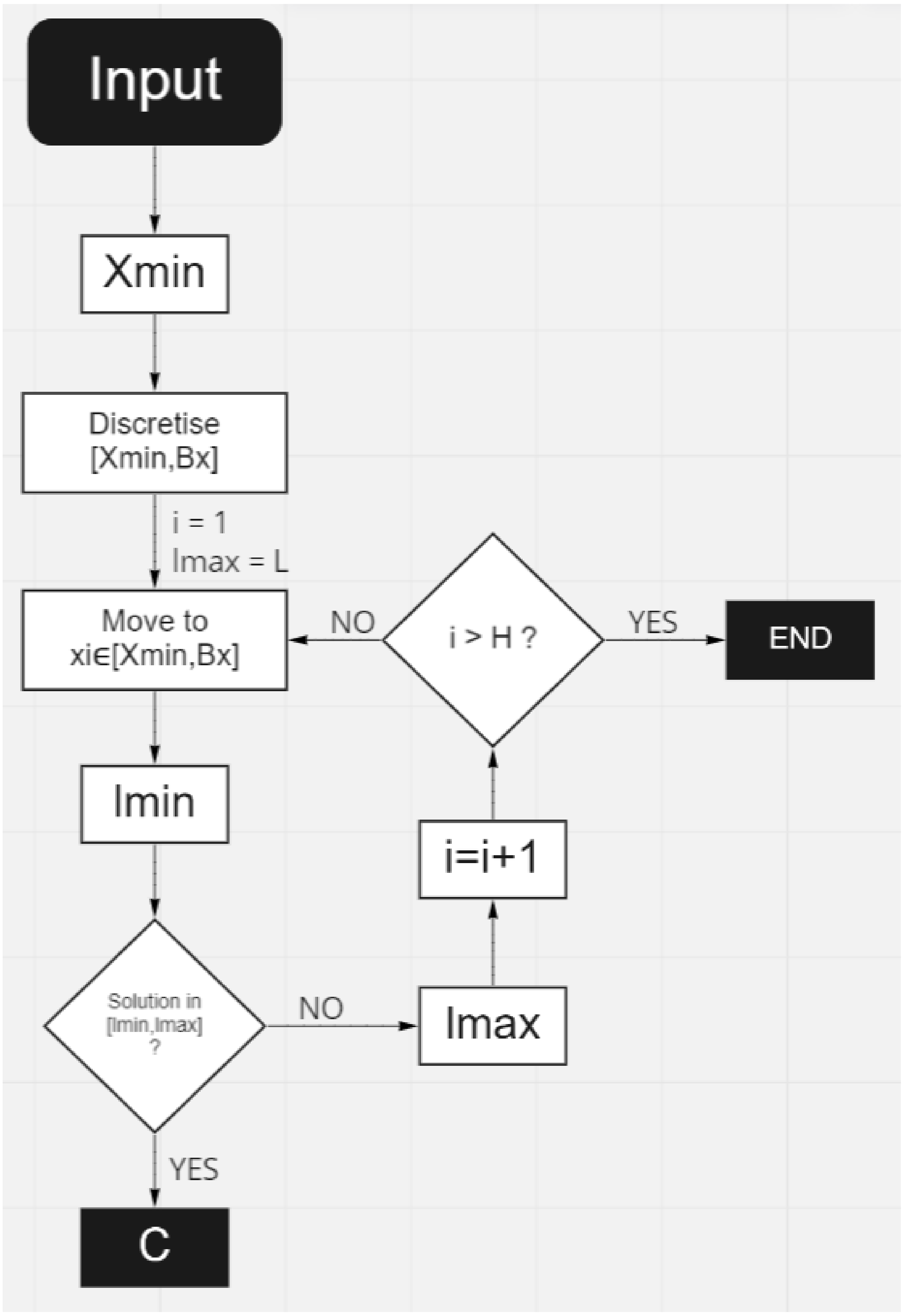


Figure 3.9 Auxiliary algorithm flowchart.

2. Calculate the distance $d(A,C_0)$.

3. From the calculated distance, determine the range of planes $[\pi(\alpha_\mu),\pi(\alpha_\eta)]$ in which it is convenient to search for a solution, where $\alpha_\mu \leq 0 \leq \alpha_\eta$.

4. If a solution is found in $\pi(\alpha_i) \in [\pi(\alpha_\mu),\pi(\alpha_\eta)]$ such that $d(A,C_i) < d(A,C_0)$, go to step 5, otherwise go to step 6.

5. Recalculate $[\pi(\alpha_\mu),\pi(\alpha_\eta)]$ according to $d(A,C_i)$ and return to step 4, comparing the new solutions with $d(A,C_i)$.
6. The output is C_0 if no solution of smaller displacement has been found or, otherwise, the last point C_i obtained in step 4 such that $d(A,C_j) < d(A,C_i)$.

3.2.1 Detailed explanation of the steps

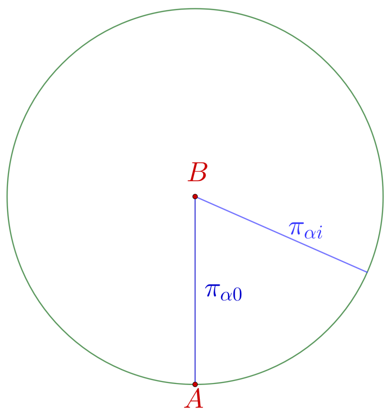
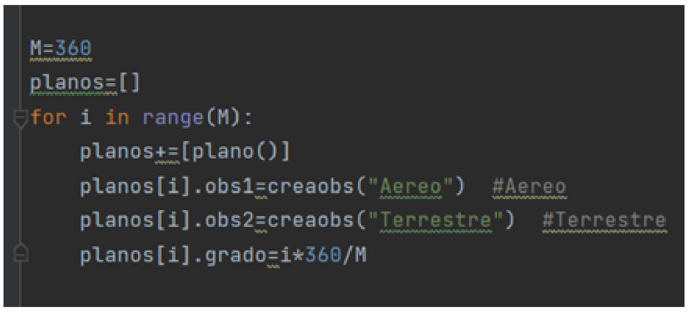


Figure 3.10 Example of planes.

As explained above, a plane is defined by the obstacles (aerial, land and ground obstacles) and a degree. The degree is determined with the current obstacle plane with respect to the plane that connects A and B in a counterclockwise direction (looking in a plan view) as illustrated in Figure 3.10 where the plane connecting points A and B, $\pi(\alpha_0)$, and the obstacle plane to be studied, $\pi(\alpha_i)$, are represented in a circle of radius:

$$r = d(A_x,B_x)$$

Creation of the planes



```
M=360
planos=[]
for i in range(M):
    planos+=[plano()]
    planos[i].obs1=creaobs("Aereo")  #Aereo
    planos[i].obs2=creaobs("Terrestre")  #Terrestre
    planos[i].grado=i*360/M
```

Figure 3.11 Plane creation.

Once the number of planes to be used is known, they are created as shown in Figure 3.11. The created planes are stored in an ordered vector from 0° to 360° with a total number of M planes. The degree of the plane depends on the number of planes, the lesser the number of planes, the more distance there is between them. If the auxiliary algorithm is applied to a different number of planes, the distance between planes can be seen (see Figure 3.12).

If the solutions obtained are counted, 57 of the 90 existing planes are obtained, this is due to the fact that many of the randomly generated obstacles do not allow a solution. The blue square corresponds to the initial position of the drone and the origin of coordinates is the target position. If 360 planes are used, one for each degree, Figure 3.13 is obtained. In this case, 220 solutions have been found.

In addition to changing the number of planes, another important aspect in the solutions is the length of the catenary. It is assumed that, if it is reduced, the solutions should be found at points closer to the target

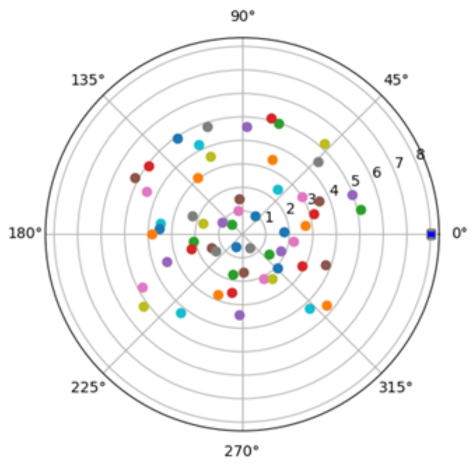


Figure 3.12 Solutions in 90 random planes.

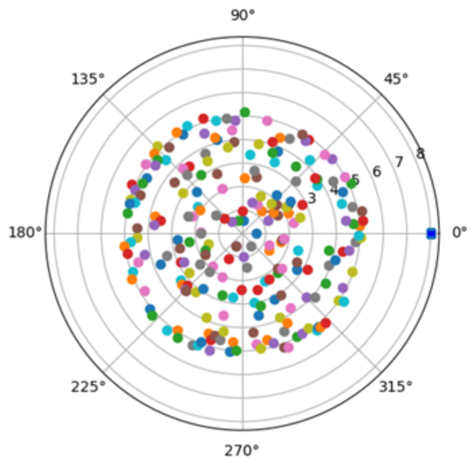


Figure 3.13 360 planes solutions.

position. To determinate if this theory is valid, a new simulation where the wire length is reduced is done, obtaining the results in Figure 3.14. Indeed, solutions are grouped more according around the target, as expected.

Finding the optimal location C

We now elaborate on finding to the plane the ground vehicle should be placed in order to arrive as quickly as possible.

In order to perform this action, three conditions will be assumed:

1. The ground vehicle is slower than the aerial vehicle, so its movement determines the speed at which the drone reaches the target point.
2. The building is accessible on one side, i.e., only obstacle planes between -90° and 90° are to be considered. Otherwise, the algorithm can be applied using different intervals of 180°, exploring all the possibilities.
3. The ground vehicle can reach any point using a linear trajectory, i.e., there is no ground obstacle in the planes.

Since the goal is the ground vehicle moves at little as possible, once the first solution for a plane has been determined, the system should not search at points further away from this solution.

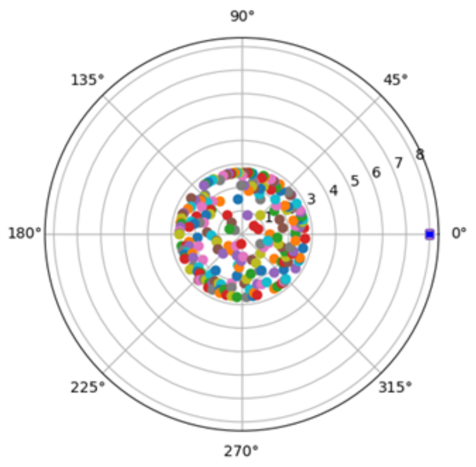


Figure 3.14 360 planes solutions with a shorter wire's length.

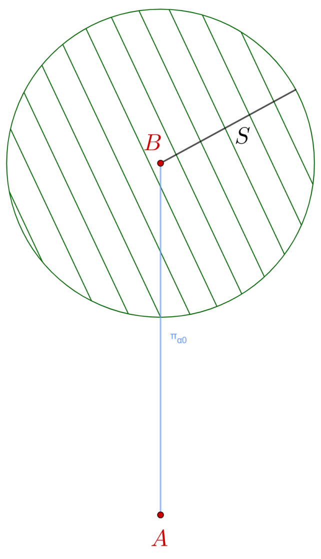


Figure 3.15 Solutions area.

With A being the initial point and B the target point (see Figure 3.15), the solutions are in the circle of centre B and radius:

$$S = \sqrt{(L^2 - (B.y - A.y)^2)}$$

If the algorithm finds a first solution in the plane $\pi(\alpha_0)$, C_0, the solutions that allow a displacement less than or equal to the one found are found inside the black circle in Figure 3.16. To find the planes in which a solution can be found, the intersection points of both circles must be determined and, from these, the plane to which they belong (see Figure 3.17).

The algorithm creates an interleaved vector, like the one used for the catenary's lengths, which makes use of all the planes between $\pi(\alpha_\mu)$ and $\pi(\alpha_\eta)$. The search will be done from the extremes of the plane interval to $\pi(\alpha_0)$.

The first plane to be evaluated is $\pi(\alpha_0)$. Once the plane has been evaluated, it is saved in a list of evaluated planes, and it is checked if the number of planes has been limited according to the distance obtained; in other case, the search interval would be [-90°,90°].

When the first plane has been evaluated, the search of a new valid solution begins. Once a solution to the plane position problem has been obtained, it is checked if it is valid, and if so, the distance $d(A,C_i)$ is calculated (see Figure 3.18). If it is less than the distance calculated in the previous points, C_i is taken as the

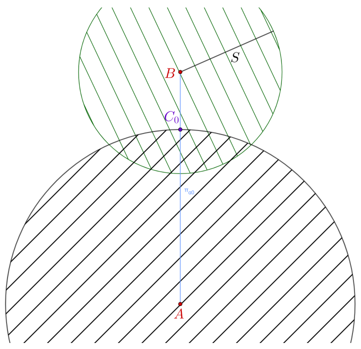


Figure 3.16 Reduction of the solutions area.

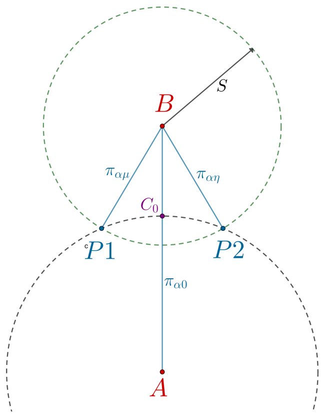
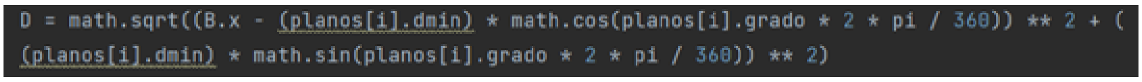


Figure 3.17 New range of planes.

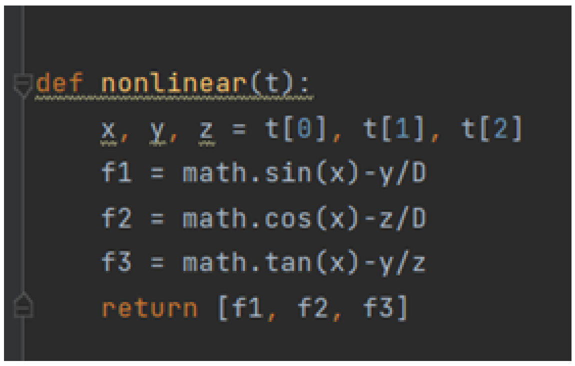


```
D = math.sqrt((B.x - (planos[i].dmin) * math.cos(planos[i].grado * 2 * pi / 360)) ** 2 + (
(planos[i].dmin) * math.sin(planos[i].grado * 2 * pi / 360)) ** 2)
```

Figure 3.18 $d(A,C_i)$.

new solution, the points P_1 and P_2 and the planes to which they belong are recalculated and iterate again. Notice that this step reduces the new interval of search planes.

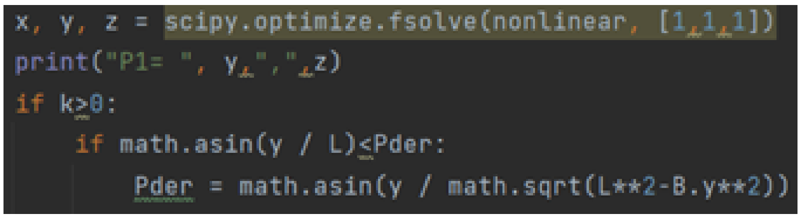
The function **nonlinear** is used to calculate P_1 and P_2 (see Figure 3.19). It calculates the trigonometric relations that exist between points, and the function **fsolve** is used.



```
def nonlinear(t):
    x, y, z = t[0], t[1], t[2]
    f1 = math.sin(x)-y/D
    f2 = math.cos(x)-z/D
    f3 = math.tan(x)-y/z
    return [f1, f2, f3]
```

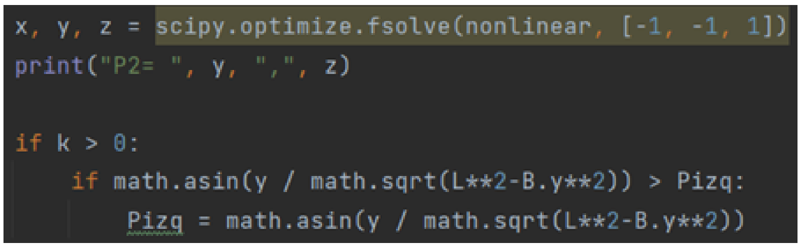
Figure 3.19 Function nonlinear.

Once the value of P_1 and P_2 are known, the new list of planes is created (see Figure 3.22), and those values are translated from two-dimensional points to degrees (see Figure 3.23). The algorithm ends with the solution nearest to the initial point A.



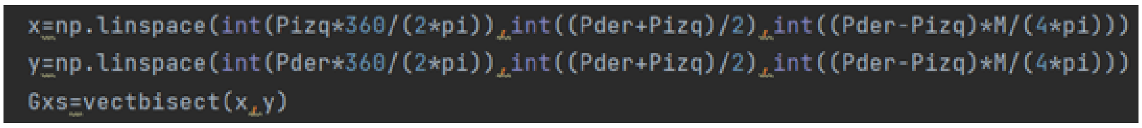
```
x, y, z = scipy.optimize.fsolve(nonlinear, [1,1,1])
print("P1= ", y,",",z)
if k>0:
    if math.asin(y / L)<Pder:
        Pder = math.asin(y / math.sqrt(L**2-B.y**2))
```

Figure 3.20 Calculation of P1.



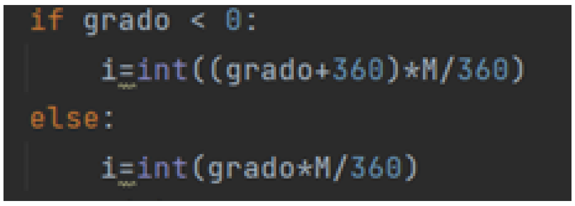
```
x, y, z = scipy.optimize.fsolve(nonlinear, [-1, -1, 1])
print("P2= ", y, ",", z)

if k > 0:
    if math.asin(y / math.sqrt(L**2-B.y**2)) > Pizq:
        Pizq = math.asin(y / math.sqrt(L**2-B.y**2))
```

Figure 3.21 Calculation of P2.



```
x=np.linspace(int(Pizq*360/(2*pi)),int((Pder+Pizq)/2),int((Pder-Pizq)*M/(4*pi)))
y=np.linspace(int(Pder*360/(2*pi)),int((Pder+Pizq)/2),int((Pder-Pizq)*M/(4*pi)))
Gxs=vectbisect(x,y)
```

Figure 3.22 List of planes.



```
if grado < 0:
    i=int((grado+360)*M/360)
else:
    i=int(grado*M/360)
```

Figure 3.23 Point to degree conversion.

3.2.2 Processing time

Since the Auxiliary algorithm is applied for every search plane, we have:

$$T_{main} = O(m*(h*n*p)) = O(m*T_{aux})$$

where:
h = Number of position points.
n = Number of points that define all the obstacles within the plane.
p = Number of catenary's lengths.
m = Number of evaluated planes

The parameters h, p and m are user-defined variables. In the next chapter these parameters are tuned to obtain good solutions in a efficient way.

3.2.3 Main algorithm flowchart

The Main algorithm flowchart is described in Figure 3.24.

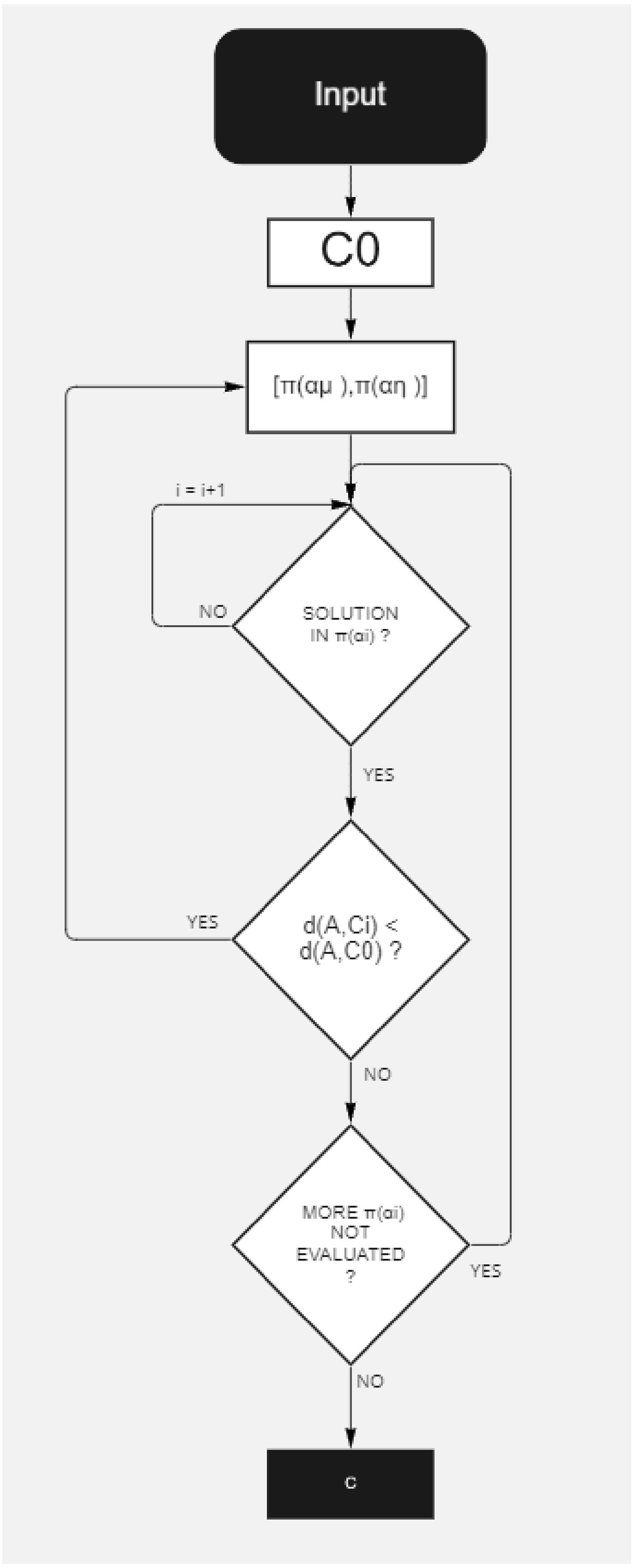


Figure 3.24 Main algorithm flowchart.

Tuning the parameters

The goal of this Chapter is study how the different parameters of the main algorithm affect the performance.

3.3 Testing the algorithm

We first check if the main algorithm works correctly. Data is entered to allow correct operation of the algorithm The response should be a map (in plan view) with the solutions of the evaluated plans, where the adopted solution is marked. Once the plan map and the solutions have been shown, the obstacle plane of the adopted solution is plotted and the catenary that allows the solution is plotted on it.

As explained above, the number of evaluated planes depends on the position of the initial solution (corresponding to the zero degrees plane) and the subsequent solutions found by the algorithm. If this solution is far from the initial point, it will search in more distant planes.

3.3.1 First solution near the initial point

A first simulation is carried out, in which the following solutions are obtained.

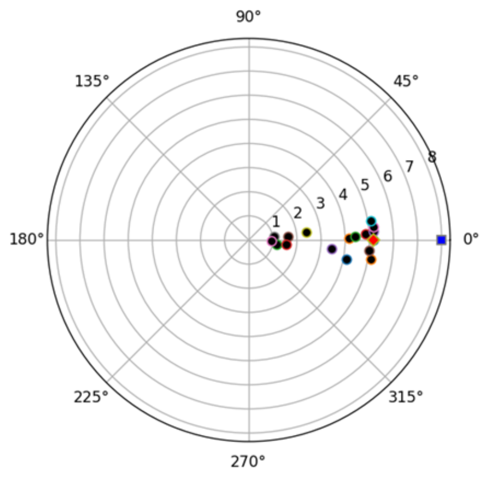


Figure 3.25 Plane solution 1.

The red point in Figure 3.25 is the chosen solution. The algorithm works correctly and is able to identify the solution with the smallest displacement. The number of planes where a new solution has been searched for is small and close to the initial plane. The solution chosen was that of the initial plane, hence there is not a solution that allows a smaller displacement in the other planes evaluated.

In addition, the algorithm also provides the local solution of the obstacle plane, to check that the ground vehicle has not been placed in an unallowed position (see Figure 3.26).

The position in which the ground vehicle has been placed is valid and there are no collisions between the wire and obstacles. Then the algorithm works correctly.

3.3.2 First solution far from the initial point

The chosen solution has not been the initial plane, and the algorithm knows how to act in the case that this happens. The number of planes examined was higher and the size of the interval was also bigger, as expected, see Figure 3.27.

3.4 Study of the processing time

The time it takes for the main algorithm to act is negligible. Therefore, the processing time of the auxiliary algorithm must be reduced as much as possible. To achieve this, the effect of each parameter with respect to the computational time and the number of solutions found will be studied.

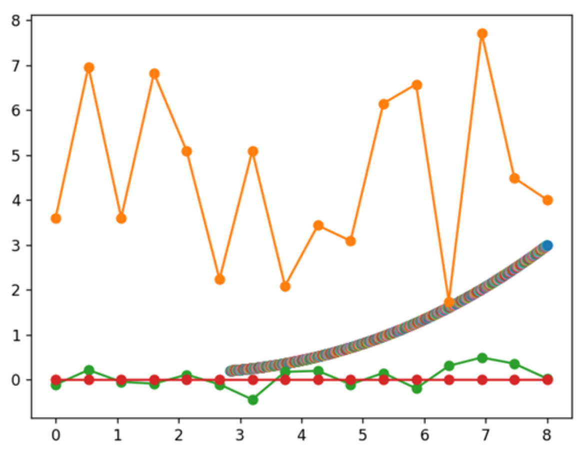


Figure 3.26 Local solution 1.

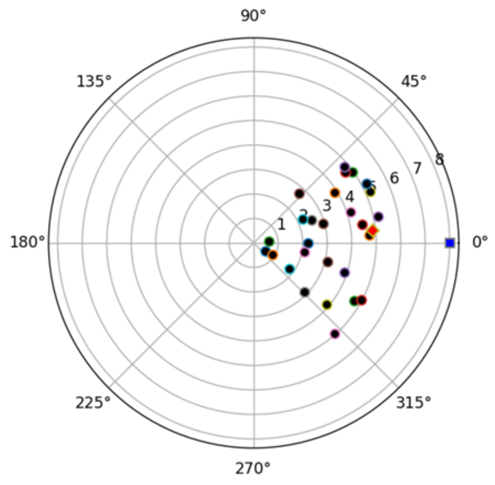


Figure 3.27 Plane solution 2.

3.4.1 Variation with respect to the number of obstacles

One of the most important computational aspects defining the problem is the number of intermediate obstacles. To check this, different simulations are carried out with various values of the number of obstacles, N, and the same number of planes, 360, with a separation between points of 8m.

Table 3.2 Obstacles and time.

N	Ttotal	Tmed	Tmin	Tmax	Solutions
5	0.7624	0.0021	0.0	0.0176	115
8	1.9341	0.0054	0.0	0.0625	164
11	4.1098	0.0114	0.0	0.1250	223
14	10.2102	0.0284	0.0	0.3282	209
17	15.6756	0.0435	0.0	0.5123	207
20	22.4702	0.0624	0.0	0.8402	217
23	36.7650	0.1021	0.0	1.3561	217
26	62.0619	0.1724	0.0	1.7657	232

For a more visual solution, the average processing time is plotted.

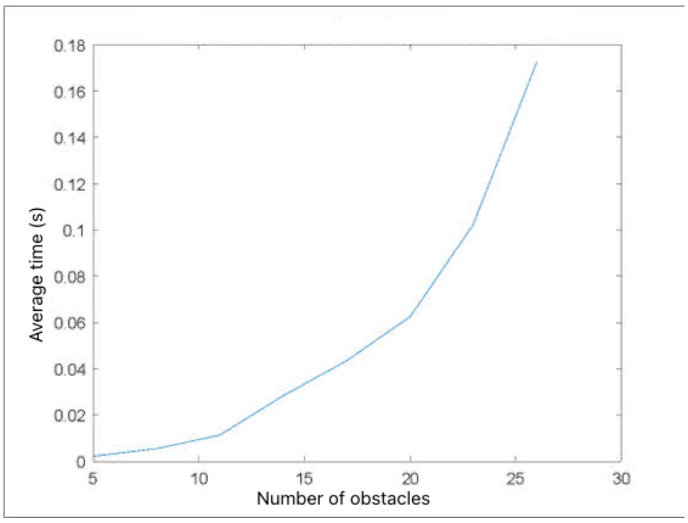


Figure 3.28 Average time vs obstacles.

As expected, the average solution calculation time increases rapidly with the number of intermediate obstacles. It is also interesting to note that the number of solutions found increases, since, as N increases, both the number of intermediate positions at which it searches and the number of different catenary's lengths also increases.

The minimum processing time is kept constant at a value equal to zero, which means that it always finds at least one solution in the first iterations of the loop.

3.4.2 Variation with respect to the number of position iterations

In order to find the optimal solution, loops must be modified depending on the number of possible solutions or the average processing time.

To find the best number of iterations, it will be compared with the number of solutions and the average execution time. As in the previous case, 360 planes, a distance between points of 8m, a catenary's length of 6 m and a number of obstacles of 16 (where the fewest solutions were found) will be used.

Table 3.3 Time, solutions and position iterations 1.

I	Tmed	Solutions
16	0.0388	224
22	0.0534	238
25	0.0515	211
28	0.0611	223
31	0.0651	222
34	0.0710	236
37	0.0734	226
50	0.0831	232

According to the data obtained, the number of solutions does not vary much according to the number of iterations of this loop, but this is in the case of a higher number. Let us see if the iterations are decreased:

There is definitely no solution that satisfactorily increases the number of solutions, but it does seem to increase when the number of iterations is more than double the number of obstacles, although it means doubling the execution time (see Figures 3.29 and 3.30).

Table 3.4 Time, solutions and position iterations 2.

I	Tmed	Solutions
16	0.0388	224
13	0.0286	218
10	0.0281	210
7	0.0212	208

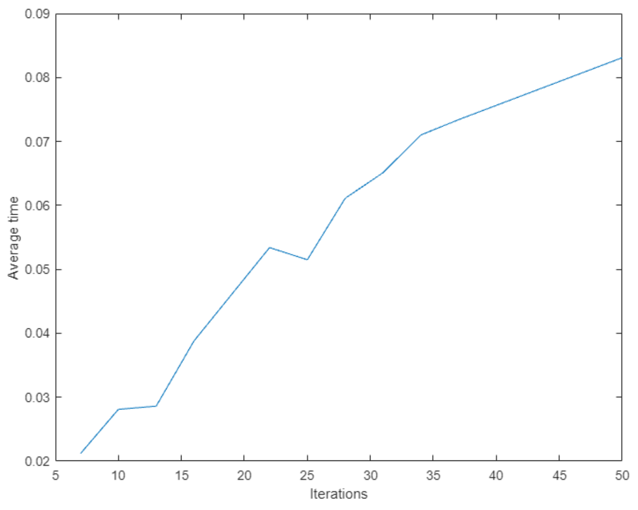


Figure 3.29 Average time vs position iterations.

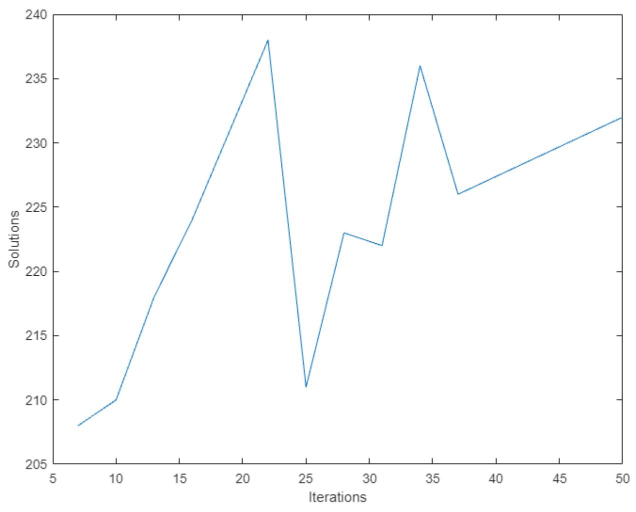


Figure 3.30 Solutions vs position iterations.

3.4.3 Variation with respect to the number of catenary iterations

Once it has been verified that the number of divisions on the displacement axis is not significant, there is one more loop to inspect, the one searching for the catenary's length.

As indicated above, the number of iterations this loop performs is

$$2 + N * \frac{(B.x - A.x)}{(B.x - Xmin)}$$

To increase the number of iterations, the independent term of the expression is going to be modified, being in this case 2, and it is going to be done in an odd way, as this number is introduced in two length vectors.

Starting from the data of the previous example and with N equal to the independent term, the following table of results is obtained.

Table 3.5 Time, solutions and catenary iterations 1.

N	Tmed	Solutions
2	0.0398	221
4	0.0421	212
6	0.0461	226
8	0.0521	215
10	0.0502	225
12	0.0591	231
14	0.0487	215
16	0.0703	221
18	0.0612	223
20	0.0769	234

As before, the number of solutions found oscillates at a constant point, probably due to the randomness of the obstacles created, which only allows for that number of solutions.

If, by increasing the number of iterations, it appears that the only factor that fluctuates is time, which must not increase, the behaviour of the algorithm if the number of iterations is reduced will be simulated.

To perform this action, acting on the independent term does not have a great effect, as once it is less than two, the choice of the integer of the expression will not take it into account. Then the action will occur on N and the expression will be as follows:

$$2+\frac{N}{U}*\frac{(B.x-A.x)}{(B.x-Xmin)},$$

where U is a factor that will decrease the number of searches in each loop.

Table 3.6 Time, solutions and catenary iterations 2.

U	Tmed	Solutions
1	0.0368	212
1.2	0.0280	210
1.4	0.0278	206
1.6	0.0291	190
1.8	0.0285	191
2	0.0216	197
2.2	0.0241	191
2.4	0.0201	179
2.6	0.0227	187
2.8	0.0197	178
3	0.0182	161

By plotting the data obtained in both tables (see Tables 3.5 and 3.6), the behaviour of the algorithm can be understood.

Figure 3.31 is not a direct input-output relationship. Changes from 0 to 1 represent the value of $1/U$ and changes from 2 to 20 represent the value of the independent term.

The number of solutions grows rapidly as the factor multiplying the average number of obstacles found in the plane segment to be examined increases, but once it becomes equal to the total number of obstacles ($U = 1$), the number of solutions found grows slowly, increasing the execution time considerably and, depending on the shape of the points, tending to a maximum average value.

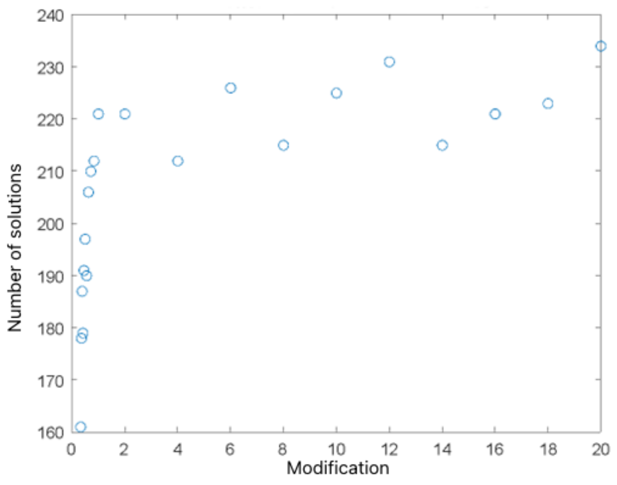


Figure 3.31 Solutions vs modifications.

Surprisingly, it appears that both the position loop and the catenary's length choice loop were already in the optimal number of iterations for the solution search algorithm.

3.4.4 Variation with respect to maximum catenary's length

Another variable that can influence the performance of the solution search is the maximum length of the catenary, since the distance between points is kept constant for all tests, allowing to observe the changes in behaviour with the same initial conditions.

The distance at which the initial and target points are located is 8 metres on the X-axis and 2.8 metres on the Y-axis, since the catenary has been given an initial height of 0.2 metres. The maximum length with which the simulations of the previous points have been made is 6 metres. Changing this, and simulating the same number of planes, the following results are obtained:

Table 3.7 Time, solutions and catenary's length.

L	Tmed	Solutions
3	0.0013	156
3.5	0.0029	224
4	0.0056	205
4.5	0.0140	209
5	0.0220	211
5.5	0.0339	239
6	0.0370	209
6.5	0.0451	225
7	0.0551	225
7.5	0.0528	226
8	0.0610	256
8.5	0.0710	283
9	0.0340	291
9.5	0.0228	318
10	0.0220	331

Breaking with the previous points, the number of solutions found and the average calculation time are highly dependent on the maximum catenary's length. To get a better idea of this variation, the data obtained are shown (see Figures 3.32 and 3.33).

Knowing that the search point is not smaller than the origin point of the plane in any way, it is possible to make use of catenary's lengths greater than the distance between points without worrying about obtaining invalid solutions. What can be observed in the results is that the number of solutions found by the algorithm

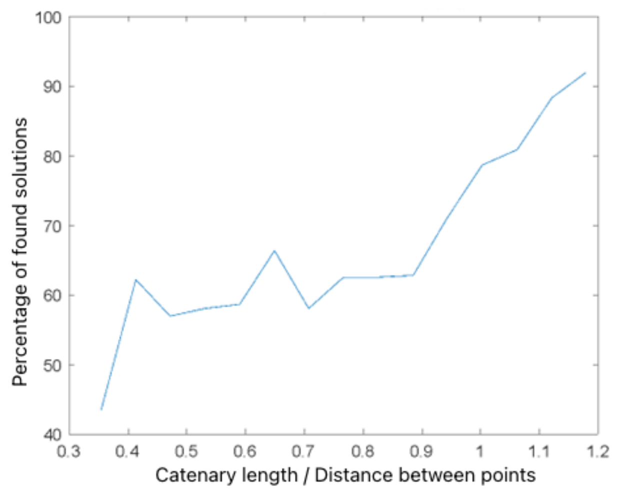


Figure 3.32 Solutions vs catenary.

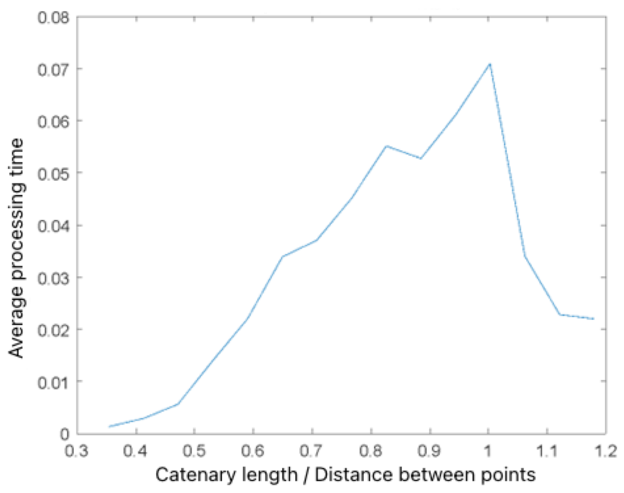


Figure 3.33 Time vs catenary.

grows significantly when the maximum catenary's length is equal to or greater than 90% of the length between the origin point and the target, and the processing time has a maximum when the length is equal to the distance.

Creating a very simple cost function:

$$J = cs + ct$$

Where **cs** is the cost with respect to the solutions and **ct** is the average time cost.

To obtain the solution of the cost function that minimises the processing and maximises the number of solutions obtained, cs will be defined as the percentage of solutions not found and ct as the percentage of time with respect to the maximum time.

Smaller values are given for small variations to the right side or large variations to the left side (see Figure 3.34). If a catenary's length capable of operating on the right side of the maximum of the cost function is available, it is possible to obtain many solutions in a small time interval. If, on the other hand, not so much catenary's length is available, one option is to use 40% of the distance between points, which allows 60% of solutions to be found with much shorter times than those to the right of the maximum.

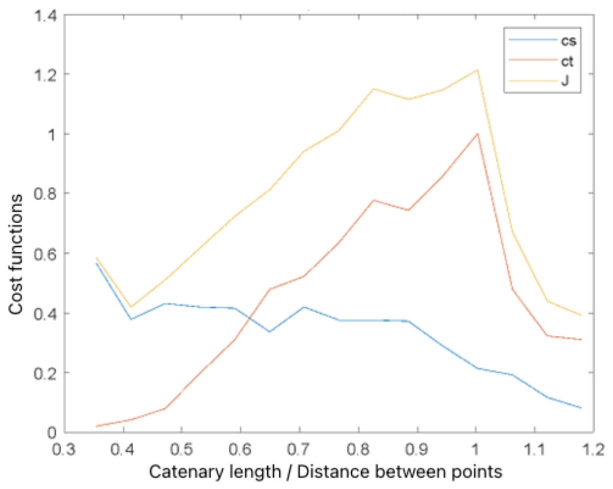


Figure 3.34 Cost function.

3.5 Initial length value

As mentioned in Chapter 3.1.1 the loop of lengths is traversed from values close to l_{max} to values close to l_{min} starting at the middle point. This middle point will be name as l_{med}.

The reason to use l_{med} is to avoid making searches near the extremes of the length interval, increasing the possibilities of finding a solution in the first searches. It is for this reason that l_{med} must be adjust to its optimal value.

This value will be obtained empirically, making use of the solution found in the evaluated planes. Plotting the length that allows the solution of the auxiliary algorithm against the horizontal distance between the position point and the target, the Figure 3.35 is obtained.

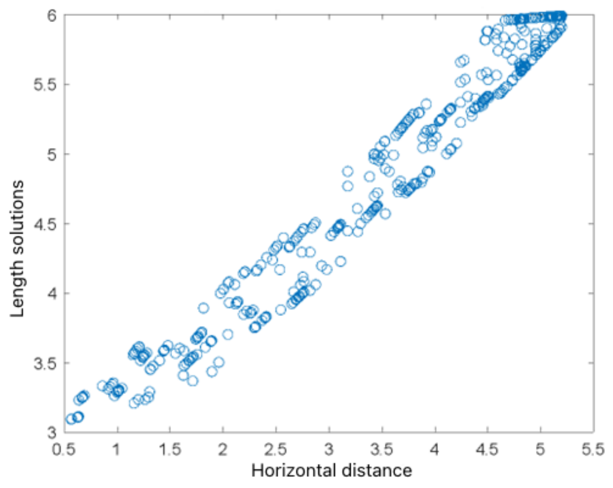


Figure 3.35 Solutions.

The solutions for each distance seem to vary up to 0.5 metres from each other as maximum. This variation should change depending on the characteristics of the environment, but it can help to provide an initial value for l_{med}, since this value will be the mean length for each distance. The simplest way to obtain this value is to fit the data by using least squares, in this case, with a straight line (see Figure 3.36).

The solution obtained was as follows:

$$l_{med} = 0.6597 * d + 2.5307$$

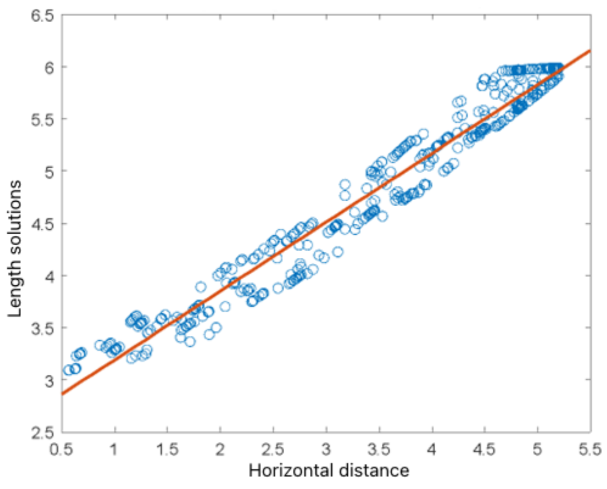
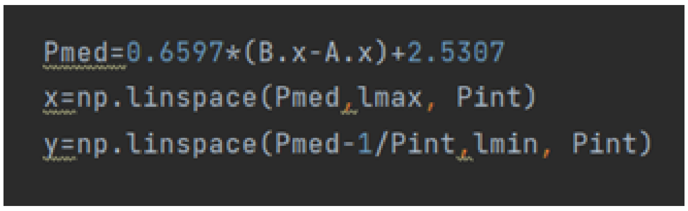


Figure 3.36 Least square line.

Introducing this value into the algorithm is as simple as doing as shown in Figure 3.37.



```
Pmed=0.6597*(B.x-A.x)+2.5307
x=np.linspace(Pmed,lmax, Pint)
y=np.linspace(Pmed-1/Pint,lmin, Pint)
```

Figure 3.37 Initial point in algorithm.

To find out if there is a significant variation with respect to the value used originally, it is used a table of times according to the number of obstacles and compared with the new results (see Table 3.8).

Table 3.8 Time, obstacles and methods 1.

L	Tprevious	Tnew
5	0.0021	0.0012
8	0.0054	0.0041
11	0.0114	0.0106
14	0.0284	0.0227
17	0.0435	0.0404
20	0.0624	0.0816
23	0.1021	0.1063
26	0.1724	0.1718

The processing time is smaller until 17 intermediate obstacles are reached, thereafter, the algorithm works worse. This is because the data from which the least squares line is derived comes from simulations with 16 intermediate obstacle points.

To solve this error, the number of intermediate obstacles can be introduced as a new variable and the above calculations can be repeated, obtaining a new formula to l_{med}.

$$l_{med} = 0.9981 * d - 0.0017 * z + 2.5090$$

Where z is the number of obstacles between A and B. Now, Table 4.1 is created with the same purpose as the last one, to see if there is a significant variation in the time of the auxiliary algorithm.

Table 3.9 Time, obstacles and methods 2.

L	Tprevious	Tnew
5	0.0021	0.0017
8	0.0054	0.0032
11	0.0114	0.0087
14	0.0284	0.0139
17	0.0435	0.0238
20	0.0624	0.0381
23	0.1021	0.0438
26	0.1724	0.0578

To get a clearer idea of the solutions obtained, the average processing times obtained are plotted (see Figure 3.38).

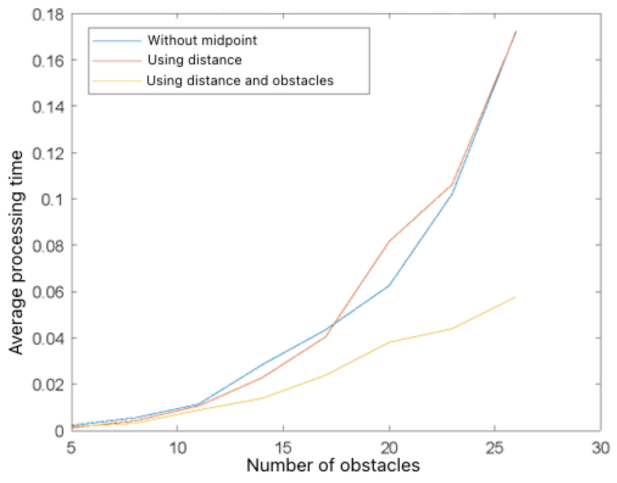


Figure 3.38 Time and methods.

Making use of distance has no significant effect when increasing the number of obstacles, but taking the obstacles into account to calculate l_{med} reduces the time needed to find a solution by almost half.

4 Extensions

In the above Chapters we showed that the algorithm is capable of finding a solution to the proposed problem and its parameters have been modified to obtain the solution in the shortest possible time as possible.

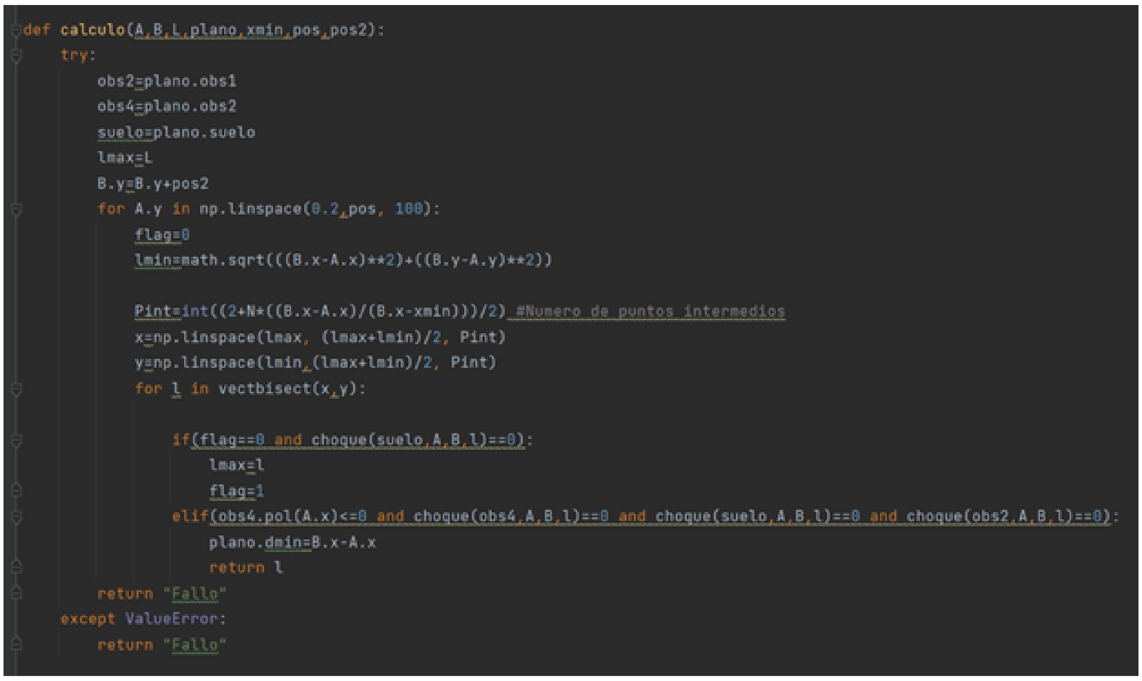
In this Chapter, we explore how to adapt the method to find solutions for some extensions of the problem.

4.1 Exploring around the target point

Once the aerial vehicle has reached the target point, it may need to be able to move in different directions to complete the inspection.

If it is assumed that the obstacles will not be highly variable and that the ground vehicle is able to increase the initial height of the catenary to a specific point, it is possible to create an algorithm that allows the aerial vehicle to move.

By making a few small changes to the function that previously calculated the position in the plane (auxiliary algorithm), it can be converted into one that calculates the initial vertical position of the catenary (see Figure 4.1).



```python
def calculo(A,B,L,plano,xmin,pos,pos2):
    try:
        obs2=plano.obs1
        obs4=plano.obs2
        suelo=plano.suelo
        lmax=L
        B.y=B.y+pos2
        for A.y in np.linspace(0.2,pos, 100):
            flag=0
            lmin=math.sqrt(((B.x-A.x)**2)+((B.y-A.y)**2))

            Pint=int((2+N*((B.x-A.x)/(B.x-xmin)))/2) #Numero de puntos intermedios
            x=np.linspace(lmax, (lmax+lmin)/2, Pint)
            y=np.linspace(lmin,(lmax+lmin)/2, Pint)
            for l in vectbisect(x,y):

                if(flag==0 and choque(suelo,A,B,l)==0):
                    lmax=l
                    flag=1
                elif(obs4.pol(A.x)<=0 and choque(obs4,A,B,l)==0 and choque(suelo,A,B,l)==0 and choque(obs2,A,B,l)==0):
                    plano.dmin=B.x-A.x
                    return l
        return "Fallo"
    except ValueError:
        return "Fallo"
```

Figure 4.1 Function vertical position.

This new function requires 2 more inputs than before:

1. **Pos:** This is the maximum vertical position of the catenary starting point.
2. **Pos2:** This is the vertical position with respects to point B_y. If positive, the drone has a pos2 value greater than B_y, otherwise, in the negative case, it has a pos2 value less than B_y.

What this algorithm allows to do is to find a catenary's length that allows the drone to move vertically. Figure 4.2 shows the value of the vertical position (of the initial point) against the catenary's length that allows the solution of the problem.

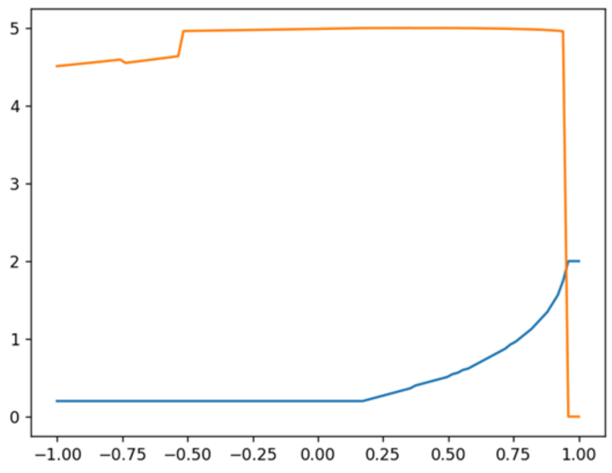


Figure 4.2 Wire's length and vertical position 1.

The orange graph represents the catenary's length and the blue graph the vertical position. For a movement from -1 metre to 1 metre with respect to the target point, it is able to find a solution up to a point quite close to one metre high, since there is always a collision of an obstacle with the catenary.

Repeating the simulation with other randomly generated obstacles, the aerial vehicle is able to perform the complete movement. In this case, the catenary's length does not continue to increase, as it has reached its defined maximum (see Figure 4.3).

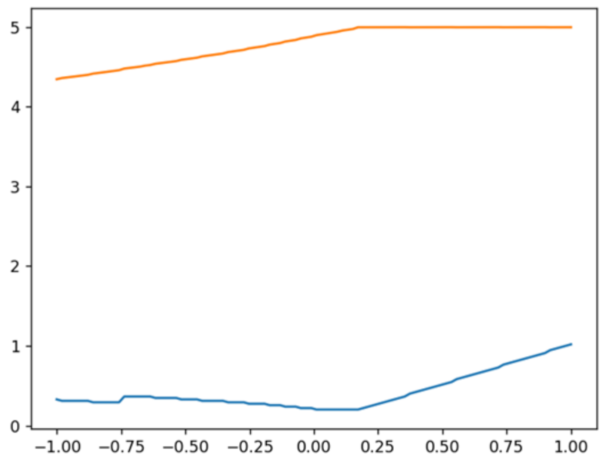
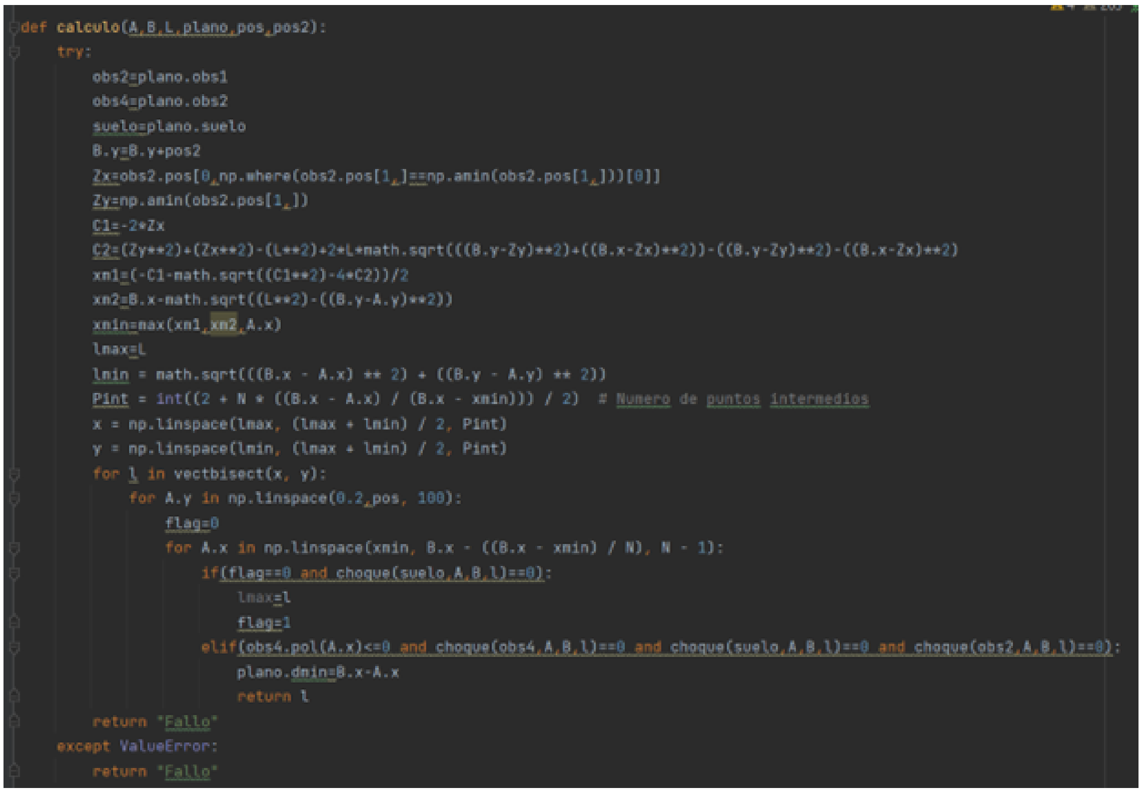


Figure 4.3 Wire's length and vertical position 2.

If in addition to the ability for vertical positions, it is also able to move horizontally, it is possible to implement an algorithm that allows the complete movement of the starting point of the catenary (see Figure 4.4).



```
def calculo(A,B,L,plano,pos,pos2):
    try:
        obs2=plano.obs1
        obs4=plano.obs2
        suelo=plano.suelo
        B.y=B.y+pos2
        Zx=obs2.pos[0,np.where(obs2.pos[1,]==np.amin(obs2.pos[1,]))[0]]
        Zy=np.amin(obs2.pos[1,])
        C1=-2*Zx
        C2=(Zy**2)+(Zx**2)-(L**2)+2*L*math.sqrt(((B.y-Zy)**2)+((B.x-Zx)**2))-((B.y-Zy)**2)-((B.x-Zx)**2)
        xn1=(-C1-math.sqrt((C1**2)-4*C2))/2
        xn2=B.x-math.sqrt((L**2)-((B.y-A.y)**2))
        xmin=max(xn1,xn2,A.x)
        lmax=L
        lmin = math.sqrt(((B.x - A.x) ** 2) + ((B.y - A.y) ** 2))
        Pint = int((2 + N * ((B.x - A.x) / (B.x - xmin))) / 2)  # Numero de puntos intermedios
        x = np.linspace(lmax, (lmax + lmin) / 2, Pint)
        y = np.linspace(lmin, (lmax + lmin) / 2, Pint)
        for l in vectbisect(x, y):
            for A.y in np.linspace(0.2,pos, 100):
                flag=0
                for A.x in np.linspace(xmin, B.x - ((B.x - xmin) / N), N - 1):
                    if(flag==0 and choque(suelo,A,B,l)==0):
                        lmax=l
                        flag=1
                    elif(obs4.pol(A.x)<=0 and choque(obs4,A,B,l)==0 and choque(suelo,A,B,l)==0 and choque(obs2,A,B,l)==0):
                        plano.dmin=B.x-A.x
                        return l
        return "Fallo"
    except ValueError:
        return "Fallo"
```

Figure 4.4 Complete position function 1.

Basically, it consists of adding the horizontal position loop as the last position option, since it takes the longest time (ground vehicle movement). If the results are plotted, the Figure 4.5 is obtained.

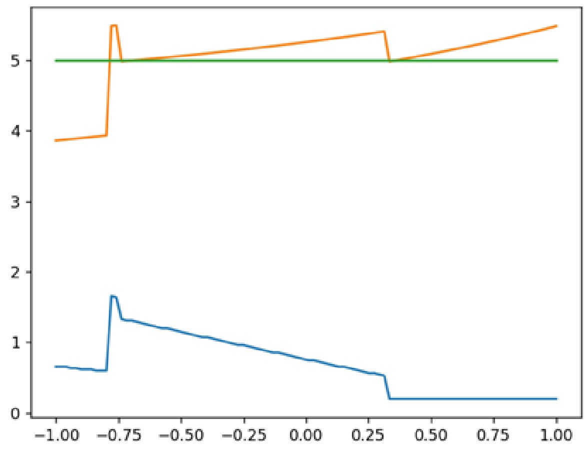


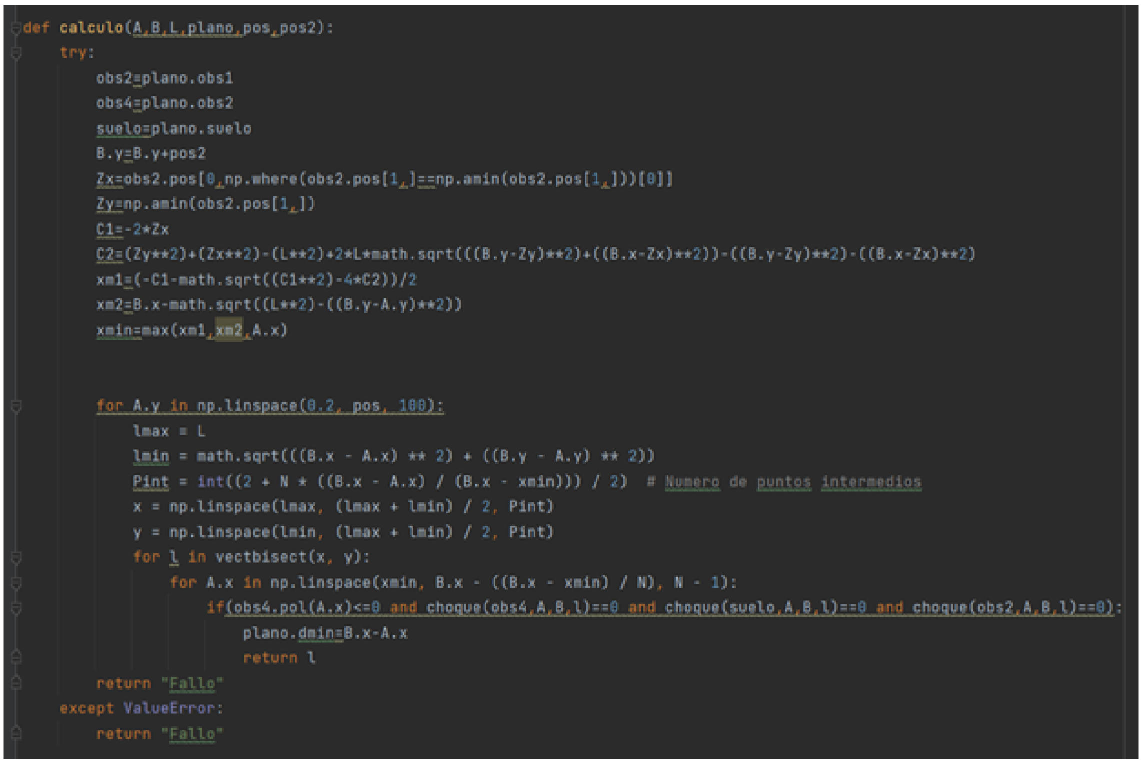
Figure 4.5 Wire's length and positions 1.

The blue curve represents the vertical position, the orange curve the horizontal position and the green curve the length of the catenary.

Notice that the length is always the maximum possible; this is because the length loop is the main one, to obtain something more realistic, the function is changed as follows:

Now, let's consider the vehicle with a horizontal movement. If the results of the new function are plotted, the Figure 4.7 is obtained.

It does not make much difference, the other simulated cases are quite similar, the system always dispenses with one of the variables, which seems to be fixed, and the others vary. Then, it is not worth using all three



```
def calculo(A,B,L,plano,pos,pos2):
    try:
        obs2=plano.obs1
        obs4=plano.obs2
        suelo=plano.suelo
        B.y=B.y+pos2
        Zx=obs2.pos[0,np.where(obs2.pos[1,]==np.amin(obs2.pos[1,]))[0]]
        Zy=np.amin(obs2.pos[1,])
        C1=-2*Zx
        C2=(Zy**2)+(Zx**2)-(L**2)+2*L*math.sqrt(((B.y-Zy)**2)+((B.x-Zx)**2))-((B.y-Zy)**2)-((B.x-Zx)**2)
        xm1=(-C1-math.sqrt((C1**2)-4*C2))/2
        xm2=B.x-math.sqrt((L**2)-((B.y-A.y)**2))
        xmin=max(xm1,xm2,A.x)


        for A.y in np.linspace(0.2, pos, 100):
            lmax = L
            lmin = math.sqrt(((B.x - A.x) ** 2) + ((B.y - A.y) ** 2))
            Pint = int((2 + N * ((B.x - A.x) / (B.x - xmin))) / 2)  # Numero de puntos intermedios
            x = np.linspace(lmax, (lmax + lmin) / 2, Pint)
            y = np.linspace(lmin, (lmax + lmin) / 2, Pint)
            for l in vectbisect(x, y):
                for A.x in np.linspace(xmin, B.x - ((B.x - xmin) / N), N - 1):
                    if(obs4.pol(A.x)<=0 and choque(obs4,A,B,l)==0 and choque(suelo,A,B,l)==0 and choque(obs2,A,B,l)==0):
                        plano.dmin=B.x-A.x
                        return l
        return "Fallo"
    except ValueError:
        return "Fallo"
```

Figure 4.6 Complete position function 2.

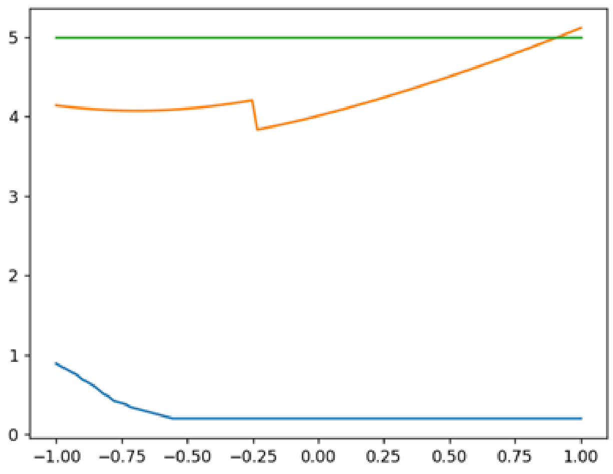


Figure 4.7 Wire's length and positions 2.

options we implement an algorithm that uses two of them.

4.2 Floating obstacles

Until now, only three types of obstacles have been defined: aerial, land and ground obstacles.

To add more applicability, a new type of obstacle will be created: the **floating obstacle**. For simplicity, a floating obstacle consists of three points in the plane. In order to avoid a collision with the wire, all points of the obstacle must be above the wire if the wire passes under it, or below it if the wire passes over it.

In Figure 4.8, the obstacle shown in green is the floating obstacle. To avoid having the same obstacle for all the planes to be evaluated, a random component is added to the creation. In this way, it is possible to add more floating obstacles.

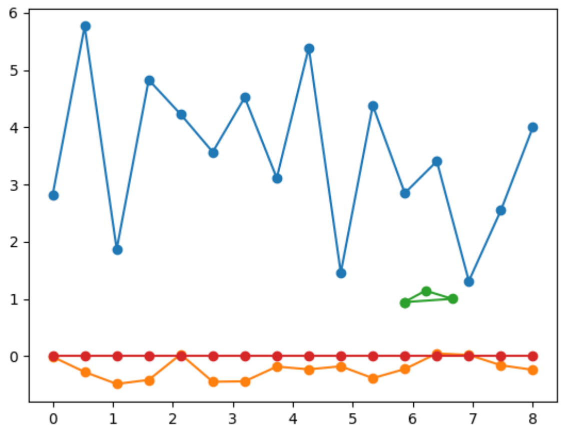


Figure 4.8 Floating obstacle.

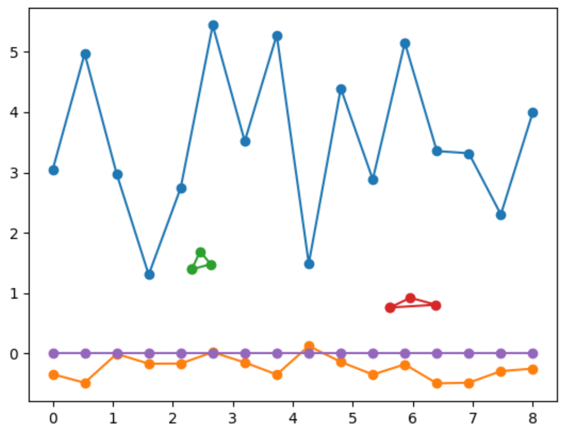


Figure 4.9 Two floating obstacles.

The tests in this chapter will be done using two or more floating obstacles (see Figure 4.9). The algorithm interprets obstacles as land or aerial depending on the relative position of the wire to each floating obstacle, allowing solutions like the one in the Figure 4.10

If the system is simulated with the same characteristics as the previous chapters (but different averages of aerial and land obstacles), an average computational time of 0.06788 seconds is obtained (doubling the previous one).

To try to improve this result, a new variable called **xp** is used.

$$xp = B_x - B_y * \frac{(B_x - X_n)}{(B_y - Y_n)},$$

where (X_n, Y_n) is the point of the floating obstacle closest to the target point. **xp** is the point of intersection of the straight line connecting the obstacle and the target point with the horizontal axis. The value of xp is going to be a new parameter that determines the value of X_{min}. Since there are two obstacles, the value to be used is the average of the two (see Figure 4.11).

Using this method, the processing time the auxiliary algorithm takes (depending on the number of floating obstacles) is as follows. The use of xp is able to improve processing time by at least 0.01 seconds. However, the higher the number of floating obstacles, the less time it takes to use xp compared to the simple algorithm.

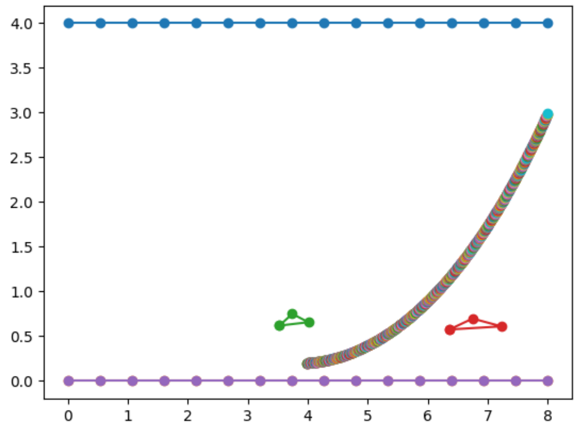
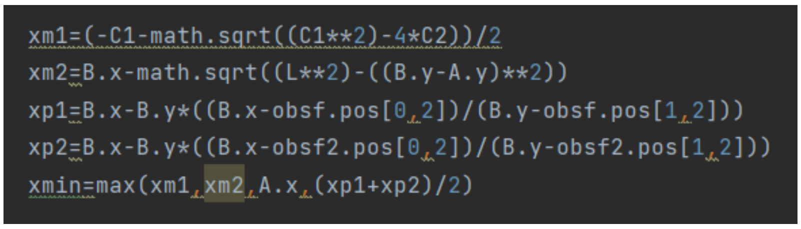


Figure 4.10 Two floating obstacles and wire.



```
xm1=(-C1-math.sqrt((C1**2)-4*C2))/2
xm2=B.x-math.sqrt((L**2)-((B.y-A.y)**2))
xp1=B.x-B.y*((B.x-obsf.pos[0,2])/(B.y-obsf.pos[1,2]))
xp2=B.x-B.y*((B.x-obsf2.pos[0,2])/(B.y-obsf2.pos[1,2]))
xmin=max(xm1,xm2,A.x,(xp1+xp2)/2)
```

Figure 4.11 New Xmin.

Table 4.1 Time, obstacles and methods 2.

N	Taverage with xp	Taverage without xp
1	0.05626	0.06821
2	0.05871	0.06788
3	0.08721	0.10800

4.3 Search for the optimal solution taking into account the obstacles

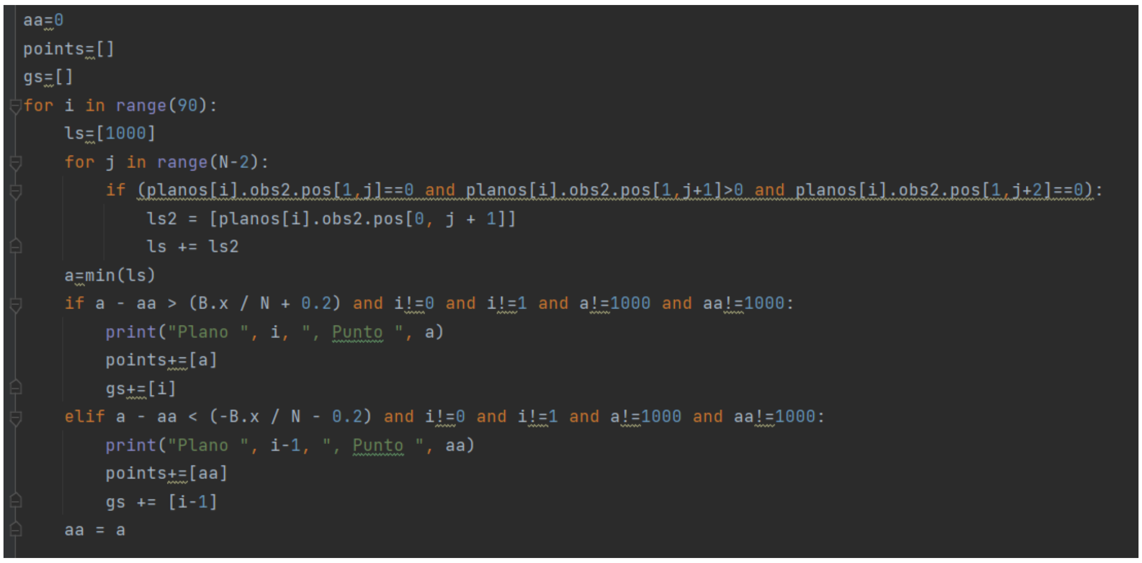
In order to choose the solution that allowed the least possible displacement of the ground vehicle, the distance between the solutions found and the starting point was calculated. If the solution search is performed in a three-dimensional space in which the ground vehicle cannot move in a straight line, the optimal solution found by the algorithm is no longer the one that produces the smallest displacement. To avoid this error, the algorithm must be able to evaluate the environment and find the three-dimensional point from which it is most convenient to calculate the distances to the solutions found, and this will depend on the obstacles.

Assuming that the air obstacles allow the ground vehicle to move, the land obstacles will be evaluated. To realise this function, the algorithm in Figure 4.12 is used. What this algorithm does is to calculate the distance between the target point and the obstacles around it. If the difference between the distance before and after is greater than the error that may exist due to the discretisation, the point found belongs to a corner and, assuming all obstacles are convex, it is a candidate point to start finding the solution of the problem.

The search is performed between -90º and 90º, thus the idea is to repeat the previous algorithm in the interval [270º,360º], obtain its solutions and save them in the same vector used in the previous one.

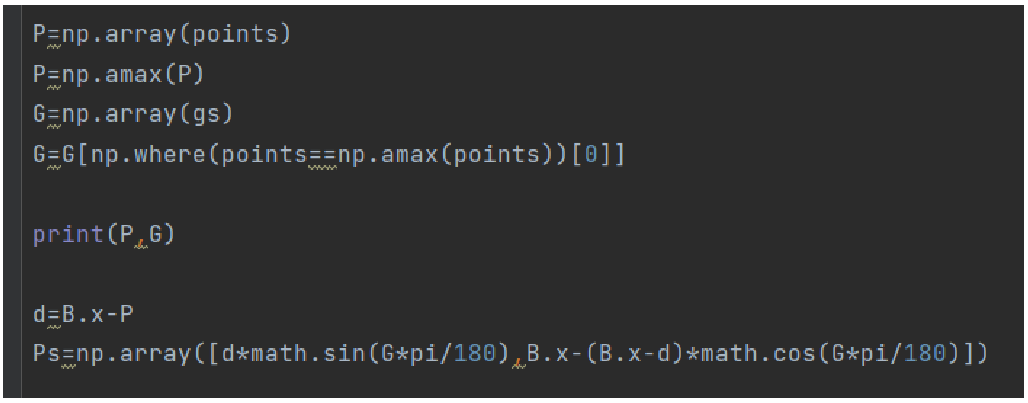
Once all the corner points have been obtained, the most convenient one for the solution search is chosen. This point is the furthest away from the target point (P, see Figure 4.13), that is, the closest to the path to be taken by the ground vehicle. Once the point has been obtained, the three-dimensional position is saved to be used in the main algorithm.

The developed algorithm is able to find a new search point as long as it is on a closed path and the obstacles



```
aa=0
points=[]
gs=[]
for i in range(90):
    ls=[1000]
    for j in range(N-2):
        if (planos[i].obs2.pos[1,j]==0 and planos[i].obs2.pos[1,j+1]>0 and planos[i].obs2.pos[1,j+2]==0):
            ls2 = [planos[i].obs2.pos[0, j + 1]]
            ls += ls2
    a=min(ls)
    if a - aa > (B.x / N + 0.2) and i!=0 and i!=1 and a!=1000 and aa!=1000:
        print("Plano ", i, ", Punto ", a)
        points+=[a]
        gs+=[i]
    elif a - aa < (-B.x / N - 0.2) and i!=0 and i!=1 and a!=1000 and aa!=1000:
        print("Plano ", i-1, ", Punto ", aa)
        points+=[aa]
        gs += [i-1]
    aa = a
```

Figure 4.12 Obstacle algorithm 1.



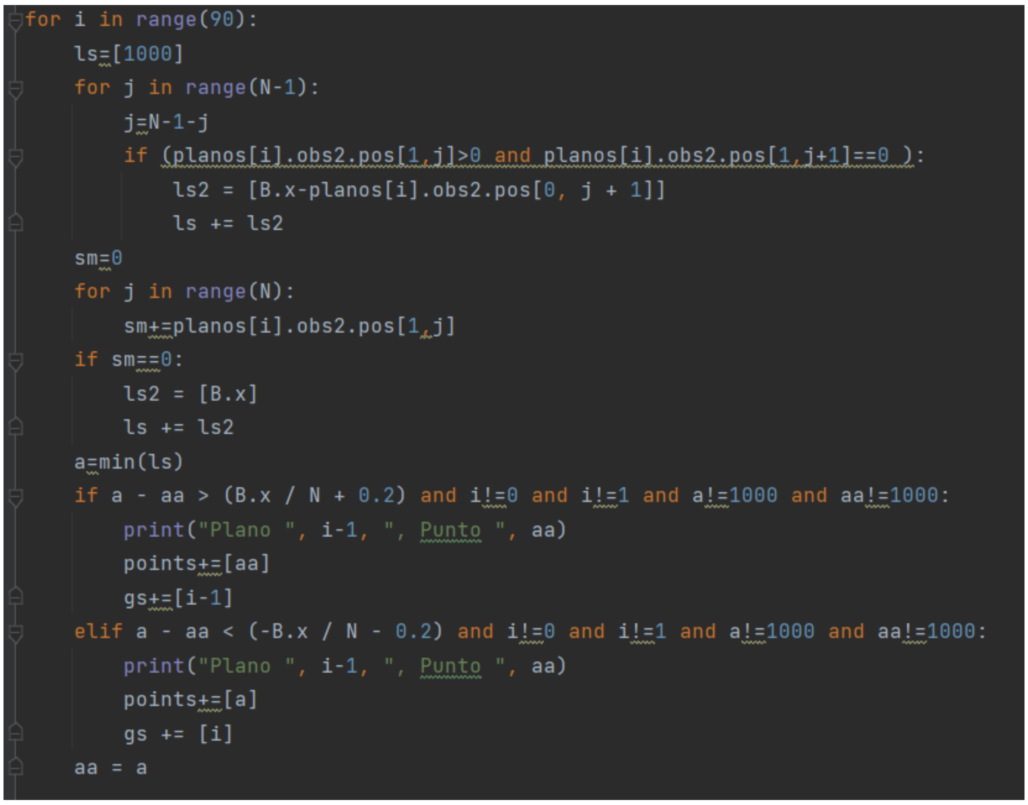
```
P=np.array(points)
P=np.amax(P)
G=np.array(gs)
G=G[np.where(points==np.amax(points))[0]]

print(P,G)

d=B.x-P
Ps=np.array([d*math.sin(G*pi/180),B.x-(B.x-d)*math.cos(G*pi/180)])
```

Figure 4.13 Obstacle algorithm 2.

are one unit thick at the scale at which the environment is discretised. With a few small changes (see Figure 4.14), this problem can be solved.

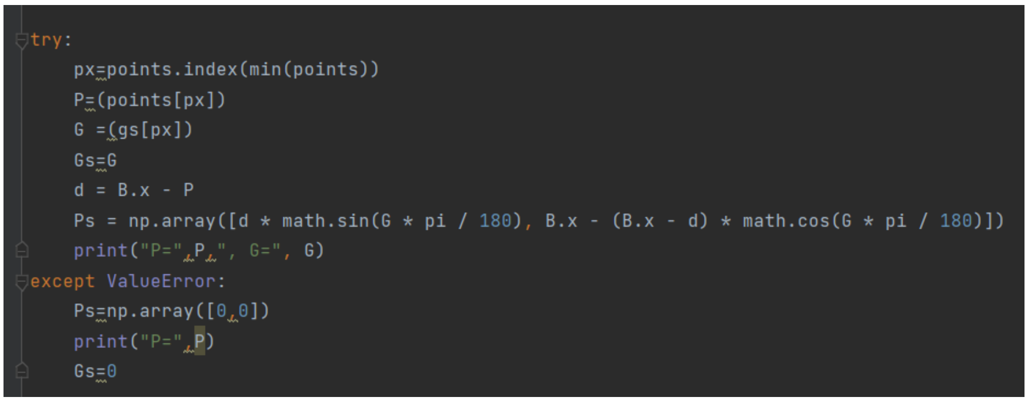
In this new algorithm, the obstacle is evaluated independently of its thickness and the planes in which there are no obstacles are taken into account.

When the search for candidate points for a new search point has been completed, the search point is checked for valid solutions. If none are found, the search point remains as the initial one, otherwise, we can use visibility graphs.



```
for i in range(90):
    ls=[1000]
    for j in range(N-1):
        j=N-1-j
        if (planos[i].obs2.pos[1,j]>0 and planos[i].obs2.pos[1,j+1]==0 ):
            ls2 = [B.x-planos[i].obs2.pos[0, j + 1]]
            ls += ls2
    sm=0
    for j in range(N):
        sm+=planos[i].obs2.pos[1,j]
    if sm==0:
        ls2 = [B.x]
        ls += ls2
    a=min(ls)
    if a - aa > (B.x / N + 0.2) and i!=0 and i!=1 and a!=1000 and aa!=1000:
        print("Plano ", i-1, ", Punto ", aa)
        points+=[aa]
        gs+=[i-1]
    elif a - aa < (-B.x / N - 0.2) and i!=0 and i!=1 and a!=1000 and aa!=1000:
        print("Plano ", i-1, ", Punto ", aa)
        points+=[a]
        gs += [i]
    aa = a
```

Figure 4.14 Obstacle algorithm. Version 2.



```
try:
    px=points.index(min(points))
    P=(points[px])
    G =(gs[px])
    Gs=G
    d = B.x - P
    Ps = np.array([d * math.sin(G * pi / 180), B.x - (B.x - d) * math.cos(G * pi / 180)])
    print("P=",P,", G=", G)
except ValueError:
    Ps=np.array([0,0])
    print("P=",P)
    Gs=0
```

Figure 4.15 Point selection. Version 2.

5 Experiments

Once all the parameters of the algorithm have been adjusted, in this Chapter some experiments have been done. It will be checked if the algorithm is able to find a solution calculating the processing time and the number of executions of the auxiliary algorithm.

In order to do this, a set of three-dimensional spaces will be created manually and studied individually, and then compared with each other in order to obtain the conclusions on the algorithm's performance.

5.1 First scenario: Simple scenario

The simplest three-dimensional map on which the algorithm can be run is one in which there are no obstacles (see Figure 5.1).

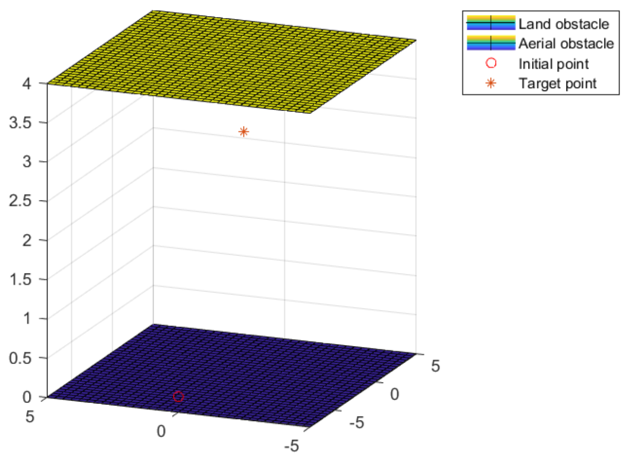


Figure 5.1 Scenario 1.

This scenario allows to calculate the lower bound of the processing time. The algorithm to be run in each scenario will have the same position and maximum length data, in order to be able to better observe the difference between each scenario. If any of the parameters are modified, the reason and the change made will be explained.

In this case, as before, the distance between the starting point and the target is 8 metres horizontally and 3 metres vertically, and the maximum wire's length is 6 metres.

Running the algorithm results in a total processing time of **0.04688 seconds** and an average search in 7 different planes before finding the optimal solution.

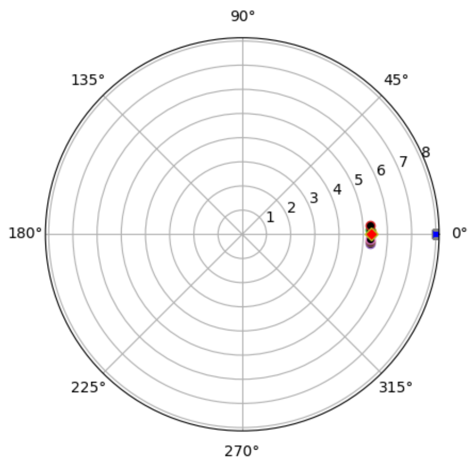


Figure 5.2 Scenario 1 solution.

5.2 Second scenario: Tunnel

The tunnel consists of a straight road from which the ground vehicle cannot exit and a road going upwards, where the target point is located. The solutions to be found by the algorithm must be inside the yellow zone (see Figure 5.3).

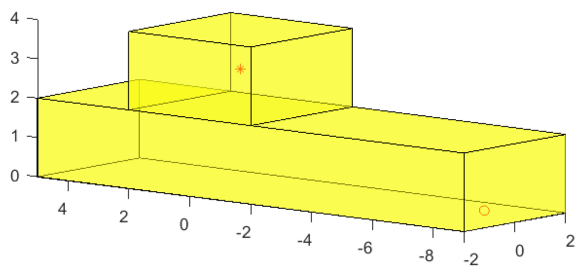


Figure 5.3 Scenario 2.

The tunnel walls are interpreted as high ground obstacles and the roof as aerial obstacles. There are no intermediate obstacles, so the processing time should not vary too much from the first scenario.

Once the algorithm is executed, the planes in which it has searched for a solution and the solution it has found in each one are obtained.

Since there are no obstacles inside the tunnel, the first search position of the algorithm only depends on the maximum wire's length, so if this does not allow the aerial vehicle to be deployed from the initial position, the first and optimal solution will always be the one in the plane connecting the initial and target points (see Figure 5.4), with all other solutions being at a greater distance.

To check that the algorithm is fully considering the tunnel, the shortest length solutions corresponding to each plane are plotted in Figure 5.5.

The tunnel is being studied correctly. The algorithm searched for a solution in 43 planes and took **1.5477 seconds** to determine the optimal position. Calculating the processing time required per plane gives 0.035993

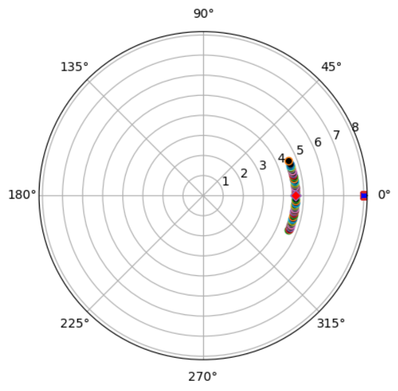


Figure 5.4 Scenario 2 solution.

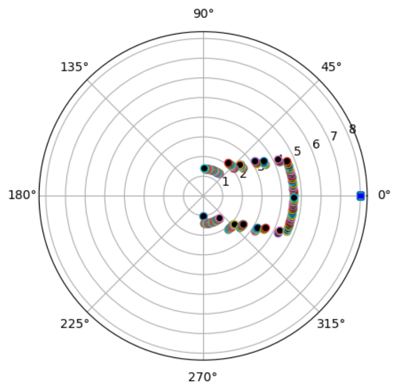


Figure 5.5 Scenario 2 plane solutions.

seconds per plane, five times more than the scenario without obstacles (0.006697 seconds per plane), which is not a bad result.

5.3 Third scenario: Zig-Zag Tunnel

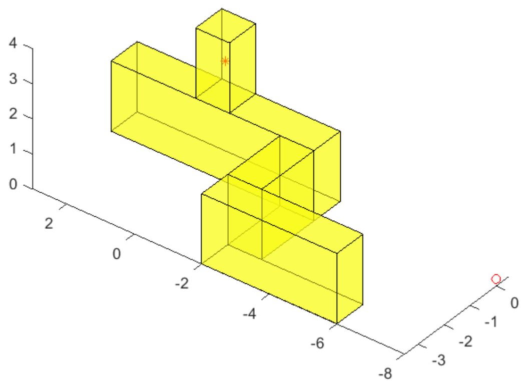


Figure 5.6 Scenario 3.

The Zig-Zag tunnel is a more complex version of the previous scenario, as it does not allow linear movement between the starting point and the target point (see Figure 5.6).

As in the previous scenario, movement is only allowed in the yellow zone. Once the algorithm is executed, the following solution map is obtained (see Figure 5.7) after searching within 99 planes and an average time of **3.69349 seconds** (0.037308 seconds per plane, an result similar to the previous scenario).

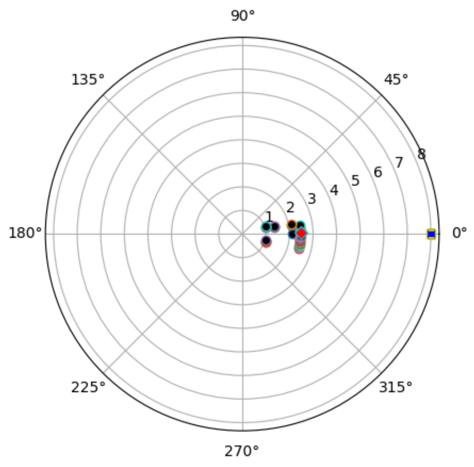


Figure 5.7 Scenario 3 solution.

The chosen solution is the closest to the initial point, but it is not the closest to the path of the ground vehicle, since it is not considering the obstacles to calculate the solutions.

In order to obtain a more realistic solution, the edge detection algorithm developed in the previous chapter will be used. When used, it finds the black spot as a source for finding solutions (see Figure 5.8).

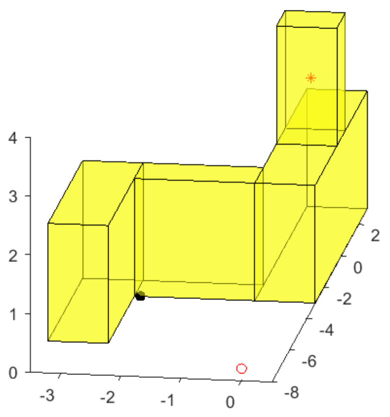


Figure 5.8 Edge detection algorithm point.

The following solutions have been obtained:

The algorithm finds the same number of solutions, but the optimal solution chosen is closer to the path that the ground vehicle must follow. The number of evaluated planes and the average solution search time is equal to the case without edge detection algorithm. If version 2 of the edge detection algorithm is used, the same solutions are obtained, but the initial search point is as shown in Figure 5.10.

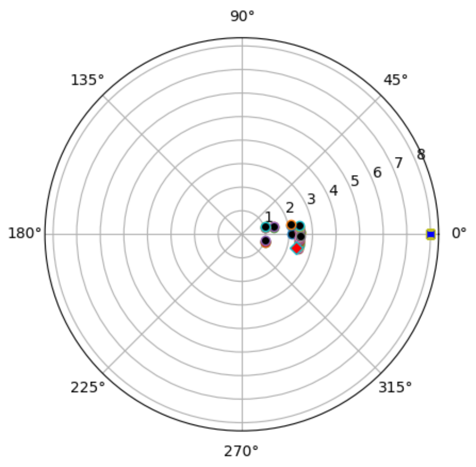


Figure 5.9 Edge detection algorithm solution.

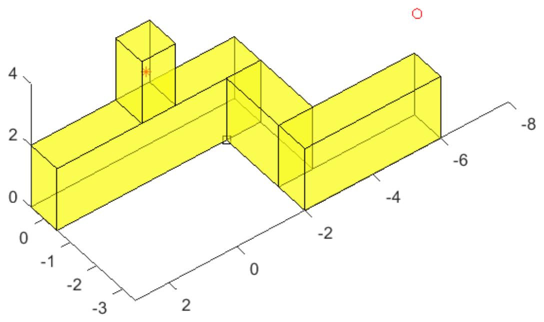


Figure 5.10 Edge detection algorithm point, version 2.

5.4 Fourth scenario: Walls

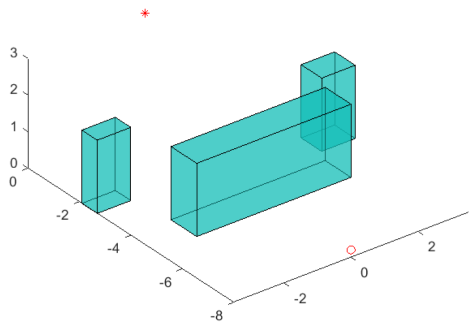


Figure 5.11 Scenario 4.

This scenario will allow the study of the functioning and efficiency of the edge detection algorithm.
The scenario consists of three walls that block the passage of the ground vehicle. For a better view, Figure 5.12 shows a plan view of the scenario.

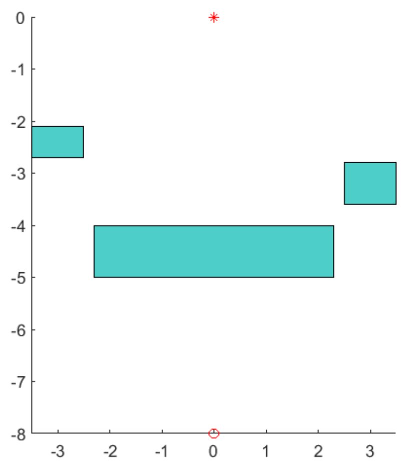


Figure 5.12 Scenario 4 plan view.

When running the edge detection algorithm, the following points are identified:

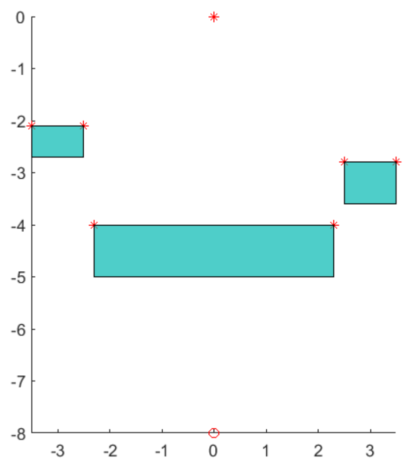


Figure 5.13 Scenario 4 Edge detection algorithm.

Recall that the exact position of these points is not what the algorithm is identifying, since each plane is being discretised. Having found the points from which a solution can be found, two simple solutions are possible:

1. Choose the solution closest to the point closest to the origin of the ground vehicle.
2. Choose the solution closest to the point furthest from the origin of the ground vehicle and closest to the target point.

If the second option is chosen, the results shown in Figure 5.14 are obtained. 130 plans were evaluated in an average time of **0.92667 seconds** (0.00713 seconds per plane). If, on the other hand, the first method is used, the solutions in Figure 5.15 are obtained.
This time, the algorithm searches only 5 planes in an average time of **0.22247 seconds** (0.044494 seconds per plane). It increases the number of seconds needed per plane, but drastically decreases the number of planes evaluated, making this option more convenient.

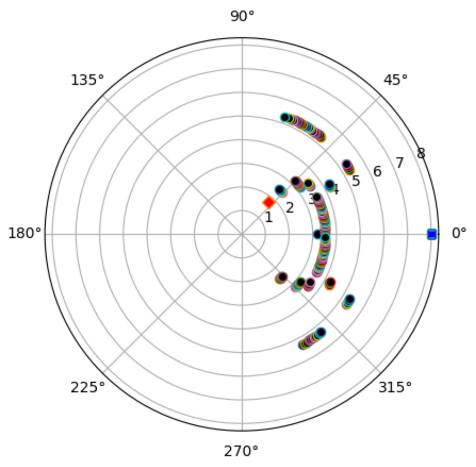


Figure 5.14 Scenario 4 Edge detection algorithm solution 2.

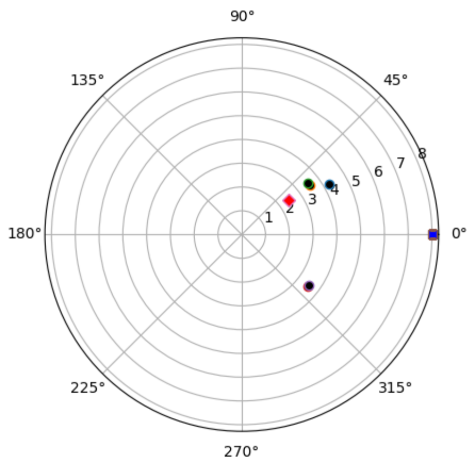


Figure 5.15 Scenario 4 Edge detection algorithm solution 1.

5.5 Adding obstacles

Finally, some obstacles are added to make the marsupial drone's movement even more difficult. To obtain the comparison, simulations are done by adding land obstacles, then aerial obstacles (both randomly generated) and, finally, the case where both obstacles are present is simulated.

A large number of simulations are run. Table 5.1 shows the average processing time and the average number of planes searched.

Table 5.1 Average times and planes. Second column without obstacles. Third column with land obstacles. Fourth column with aerial obstacles. Fifth column with both obstacles.

Scenario	Time/Planes	Time/Planes	Time/Planes	Time/Planes
Simple	0.04688 / 7	0.35510 / 31	2.50390 / 39	2.75723 / 73
Tunnel	1.54770 / 43	1.35981 / 78	0.62844 / 26	1.58846 / 61
Zig-Zag Tunnel	3.69349 / 99	1.98436 / 101	4.0088 / 101	3.66593 / 108
Walls	0.22247 / 5	0.56187 / 66	1.8837 / 71	2.29530 / 87

6 Conclusiones y trabajo futuro

Este estudio propone un método de planificación del movimiento para un sistema robótico marsupial. El sistema consta de un vehículo terrestre y un vehículo aéreo conectados por un cable, que se modela mediante una catenaria. El vehículo terrestre tiene que moverse hasta una posición que permita al vehículo aéreo alcanzar un objetivo dado sin colisiones. El movimiento del vehículo aéreo no se produce hasta que se garantiza que el vehículo de tierra está en la posición correcta. El objetivo es mover el vehículo terrestre lo menos posible, es decir, alcanzar el objetivo en el menor tiempo posible. Nuestro enfoque busca una solución de aproximación al problema de optimización. Para ello, el espacio tridimensional se ha discretizado considerando un conjunto discreto de planos perpendiculares al suelo y que pasan por el objetivo. El algoritmo principal se basa en un algoritmo auxiliar que resuelve la versión bidimensional del problema correspondiente a cada plano. Tras ajustar los parámetros correspondientes, ha sido posible diseñar un algoritmo eficiente de planificación de trayectorias que puede utilizarse en línea. La eficiencia del método utilizado ha sido probada utilizando diferentes escenarios, obteniéndose trayectorias eficientes incluso en situaciones complejas. Como trabajo futuro, queda pendiente la comparación de nuestro planificador con otros existentes en la literatura aplicados al mismo sistema marsupial, así como la posibilidad de diseñar un nuevo método de planificación en el que ambos vehículos se muevan al mismo tiempo. Esto permitiría la posibilidad de extender el enfoque para otras aplicaciones como la exploración del escenario en lugar de alcanzar el objetivo lo antes posible.

7 Conclusions and future work

This study proposes a motion planning method for a marsupial robotic system. The system consists of a ground vehicle and an aerial vehicle connected by a wire, which is modelled by a catenary. The ground vehicle has to move to a position that allows the aerial vehicle to reach a given target without collisions. The movement of the air vehicle does not take place until it is guaranteed that the ground vehicle is in the correct position. The goal is to move the ground vehicle as little as possible, that is, to reach the target in the shortest possible time. Our approach looks for an approximation solution to the optimization problem. Then, the three-dimensional space has been discretised by considering a discrete set of planes perpendicular to the ground and passing through the target. The main algorithm is based on an auxiliary algorithm that solves the two-dimensional version of the problem corresponding to each plane. After adjusting the corresponding parameters, it has been possible to design an efficient path planning algorithm that can be used online. The efficiency of the method used has been tested using different scenarios, obtaining efficient trajectories even in complex situations. As future work, it remains to compare our planner with others in the literature applied to the same marsupial system, as well as the possibility of designing a new planning method in which both vehicles move at the same time. This would allow the possibility of extending the approach for other applications as the exploration of the scenario instead of reaching the target as soon as possible.

Bibliography

[1] Murphy, R.R., Ausmus, M., Bugajska, M.D., Ellis, T., Johnson, T., Kelley, N., Kiefer, J., and Pollock, L. (1999). Marsupial-like mobile robot societies. International Conference on Autonomous Agents.

[2] Nash, A., Koenig, S., and Tovey, C. (2010). Lazy Theta*: Any-Angle Path Planning and Path Length Analysis in 3D. Proceedings of the AAAI Conference on Artificial Intelligence, 24(1), 147-154.

[3] Stankiewicz, P. G., Jenkins, S., Mullins, G. E., Wolfe, K. C., Johannes, M. S., and Moore, J. L. (2018). A motion planning approach for marsupial robotic systems, in 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Madrid, 2018 . IEEE, 2018, pp. 1–9.

[4] Sandino, L. A., Santamaria, D., Bejar, M., Viguria, A., Kondak, K., and Ollero, A. (2014). Tether-guided landing of unmanned helicopters without gps sensors, in 2014 IEEE International Conference on Robotics and Automation (ICRA), 2014, pp. 3096–3101.

[5] Papachristos, C., and Tzes, A. Tzes (2014). The power-tethered uav-ugv team: A collaborative strategy for navigation in partially-mapped environ-ments, in 22nd Mediterranean Conference on Control and Automation, 2014, pp. 1153–1158.

[6] Lockwood, E. (1961). A Book of Curves , 1st ed. Cambridge University Press, ISBN: 103156.

[7] Ivorra, C. La Catenaria. [Online] Available: https://www.uv.es/ ivorra/Libros/Catenaria.pdf

[8] Mecapedia. [Online] Available: http://www.mecapedia.uji.es/index.htm

[9] Curso python desde cero. [Online] Available:
https://www.youtube.com/playlist?list=PLU8oAlHdN5BlvPxziopYZRd55pdqFwkeS

[10] ProgramarYa, Curso de Python [Online] Available: https://www.programarya.com/Cursos/Python

[11] Hudson, N., Talbot, F., Cox, M., Williams, J., Hines, T., Pitt, A., ... and Arkin, R. C. (2021). Heterogeneous Ground and Air Platforms, Homogeneous Sensing: Team CSIRO Data61's Approach to the DARPA Subterranean Challenge. arXiv preprint arXiv:2104.09053.

[12] DARPA. (2021). "DARPA SUBTERRANEAN CHALLENGE 2021" https://https://www.subtchallenge.com/index.html.

[13] Grip, H. F., Scharf, D. P., Malpica, C., Johnson, W., Mandic, M., Singh, G., and Young, L. A. (2018). Guidance and control for a Mars helicopter. In 2018 AIAA Guidance, Navigation, and Control Conference (p. 1849).

[14] Laranjeira, M., Dune, C., and Hugel, V. (2020). Catenary-based visual servoing for tether shape control between underwater vehicles. Ocean Engineering, 200, 107018.

[15] Miki, T., Khrapchenkov, P., and Hori, K. (2019, May). UAV/UGV autonomous cooperation: UAV assists UGV to climb a cliff by attaching a tether. In 2019 International Conference on Robotics and Automation (ICRA) (pp. 8041-8047). IEEE.

[16] Martinez-Rozas, S., Alejo, D., Caballero, F., and Merino, L. (2020). Optimization-based Trajectory Planning for Tethered Marsupial Robots.

[17] Allenspach, M., Vyas, Y., Rubio, M., Siegwart, R., and Tognon, M. (2022). Human-State-Aware Controller for a Tethered Aerial Robot Guiding a Human by Physical Interaction. IEEE Robotics and Automation Letters, 7(2), 2827-2834.

[18] Sandino, L. A., Santamaria, D., Bejar, M., Viguria, A., Kondak, K., and Ollero, A. (2014, May). Tether-guided landing of unmanned helicopters without GPS sensors. In 2014 IEEE International Conference on Robotics and Automation (ICRA) (pp. 3096-3101). IEEE.

[19] Schulz, M., Augugliaro, F., Ritz, R., and D'Andrea, R. (2015, September). High-speed, steady flight with a quadrocopter in a confined environment using a tether. In 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (pp. 1279-1284). IEEE.

[20] Xiao, X., Dufek, J., and Murphy, R. (2019, September). Benchmarking tether-based uav motion primitives. In 2019 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR) (pp. 51-55). IEEE.

[21] Martinez-Rozas, S., Alejo, D., Caballero, F., and Merino, L. (2021, May). Optimization-based Trajectory Planning for Tethered Aerial Robots. In 2021 IEEE International Conference on Robotics and Automation (ICRA) (pp. 362-368). IEEE.

[22] Xiao, X., Dufek, J., and Murphy, R. R. (2021). Autonomous visual assistance for robot operations using a tethered uav. In Field and Service Robotics (pp. 15-29). Springer, Singapore.

[23] Xiao, X., Fan, Y., Dufek, J., and Murphy, R. (2018, August). Indoor uav localization using a tether. In 2018 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR) (pp. 1-6). IEEE.

[24] Martinez-Rozas, S., Alejo, D., Caballero, F. and Merino, L. (2020). Path and trajectory planning of a tethered UAV-UGV marsupial robotics system.

8 Appendix

8.1 Main and auxiliary algorithms

```
import scipy.optimize
import math
import numpy as np
from pylab import *
import time
from scipy.interpolate import interp1d

   Wire length:

   L=6

   class punto():
      x=0
      y=0.2

   Initial and target points:
A=punto()
B=punto()
B.x=8
B.y=3

   Catenary:
   def catenaria(A,B,L,t):
      try:
         def f(z):
            f = math.sinh(z)/z-math.sqrt(L**2-(B.y-A.y)**2)/(B.x-A.x)
            return f
         z=scipy.optimize.bisect(f, 0.1, 100)
         a=(B.x-A.x)/2/z
         p=(A.x+B.x-a*math.log((L+B.y-A.y)/(L-B.y+A.y)))/2
         q=(B.y+A.y-L*math.cosh(z)/math.sinh(z))/2
         try:
            return a*math.cosh((t-p)/a)+q
         except OverflowError:
            return "Fallo"
      except ValueError or ZeroDivisionError:
         return "Fallo"
```

```
    Obstacles:

    N=16 Number of obstacles between points

    class obs():
        pos=0 ||Position points matrix
        sit=0 ||Land or Aerial
        pol=0 ||Polynomial

    def creaobs(x):
        obsx=obs()
        if x=="Aereo":
            obsx.pos=np.array([np.linspace(A.x,B.x,N),(np.random.rand(N)*7+1)*(B.y/3)])
            obsx.pos[1,N-1]=B.y+1
            obsx.sit="Aereo"
        else:
            obsx.pos=np.array([np.linspace(A.x,B.x,N),np.random.rand(N)-0.5])
            obsx.sit="Terrestre"
            obsx.pol=interp1d(obsx.pos[0,],obsx.pos[1,],kind='linear')
        return obsx

    Ground:
Suelo=creaobs("Terrestre")
Suelo.pos=np.array([np.linspace(A.x,B.x,N),np.zeros((N,))])

    ||Class Plane: Two obstacles and ground
    class plano():        obs1=creaobs("Aereo")
        obs2=creaobs("Terrestre")
        suelo=Suelo
        grado=1
        dmin=10000 Solution

    def dibujaobs(obs):
        plot(obs.pos[0,],obs.pos[1,],'o-')

    def dibujacat(A,B,L):
        i=A.x
        while i<=B.x:
            plot(i, catenaria(A, B, L, i), 'o-')
            i=i+0.01*(B.x-A.x)/8

    Collision function:

    def choque(Obs,A,B,L):

    try:
        k=0
        if Obs.sit == "Aereo":
            for i in Obs.pos[0,]:
                if i>=A.x and i<B.x:
                    if catenaria(A,B,L,i)-Obs.pos[1,k] > 0:
                        return 1
                k+=1
            return 0
        else:
            k=0
            for i in Obs.pos[0,]:
```

```
            if i>=A.x and i<=B.x:
                if catenaria(A,B,L,i)-Obs.pos[1,k] < 0:
                    return 1
            k+=1
        return 0
except np.core.exceptions.UFuncTypeError:
    return 1

def vectbisect(x,y):
    z=[]
    for i in range(len(x)):
        z += [x[i]]
        z += [y[i]]
    return z
```

Auxiliary algorithm:

```
def calculo(A,B,L,plano):
    try:
        obs2=plano.obs1
        obs4=plano.obs2
        suelo=plano.suelo
        Zx=obs2.pos[0,np.where(obs2.pos[1,]==np.amin(obs2.pos[1,]))[0]]
        Zy=np.amin(obs2.pos[1,])
        C1=-2*Zx
        C2=(Zy**2)+(Zx**2)-(L**2)+2*L*math.sqrt(((B.y-Zy)**2)+((B.x-Zx)**2))-((B.y-Zy)**2)-((B.x-Zx)**2)
        xm1=(-C1-math.sqrt((C1**2)-4*C2))/2
        xm2=B.x-math.sqrt((L**2)-((B.y-A.y)**2))
        xmin=max(xm1,xm2,A.x)
        lmax=L
        for A.x in np.linspace(xmin, B.x-((B.x-xmin)/N), N-1):
            lmin=math.sqrt(((B.x-A.x)**2)+((B.y-A.y)**2))
            lmax=(3*L+lmin)/4 An aproximation
            Pint=int((2+N*((B.x-A.x)/(B.x-xmin)))/2)
            Pmed=0.9981*(B.x-A.x)-0.0017*N+2.5090
            x=np.linspace(Pmed,lmax, Pint)
            y=np.linspace(Pmed-1/Pint,lmin, Pint)
            for l in vectbisect(x,y):
                if(obs4.pol(A.x)<=0 and choque(obs4,A,B,l)==0 and choque(suelo,A,B,l)==0 and choque(obs2,A,B,l)==0):
                    plano.dmin=B.x-A.x
                    return l
        return "Fallo"
    except ValueError:
        return "Fallo"
```

Main algorithm:

```
def nonlinear(t):
    x, y, z = t[0], t[1], t[2]
    f1 = math.sin(x)-y/D
    f2 = math.cos(x)-z/D
    f3 = math.tan(x)-y/z
    return [f1, f2, f3]

Pizq=-pi/2
Pder=pi/2
k=0
```

```
r=0
m=0
m2="No value"
Da=100000
W=0
Lista=[]
while(r<4): This number can be changed
  x=np.linspace(int(Pizq*360/(2*pi)),int((Pder+Pizq)/2),int((Pder-Pizq)*M/(4*pi)))
  y=np.linspace(int(Pder*360/(2*pi)),int((Pder+Pizq)/2),int((Pder-Pizq)*M/(4*pi)))
  Gxs=vectbisect(x,y)
  for grado in Gxs: ||Between -90º and 90º
    if grado < 0:
      i=int((grado+360)*M/360)
    else:
      i=int(grado*M/360)
    print(i)
    if i in Lista:
      r+=1
      print("Lista")
    else:
      if W==0:
        i=0
        grado=0
        print("At 0º plane")          Lista += [i]
      if grado < Pizq*360/(2*pi) or grado > Pder*360/(2*pi):
        Lista += [i]
        break
      A=punto()
      calculo(A, B, L, planos[i])
      if planos[i].dmin <=L:
        D = math.sqrt((B.x - (planos[i].dmin) * math.cos(planos[i].grado * 2 * pi / 360)) ** 2 + (
        (planos[i].dmin) * math.sin(planos[i].grado * 2 * pi / 360)) ** 2)
        if D<=Da:
          m = i
          m2 = D
          x, y, z = scipy.optimize.fsolve(nonlinear, [1,1,1])
          print("P1= ", y,",",z)
          if k>0:
            if math.asin(y / L)<Pder:
              Pder = math.asin(y / math.sqrt(L**2-B.y**2))
              r=0
            else:
              r+=1
          else:
            Pder = math.asin(y / math.sqrt(L**2-B.y**2))
          x, y, z = scipy.optimize.fsolve(nonlinear, [-1, -1, 1])
          print("P2= ", y, ",", z)
          if k > 0:
            if math.asin(y / math.sqrt(L**2-B.y**2)) > Pizq:
              Pizq = math.asin(y / math.sqrt(L**2-B.y**2))
        else:
          Pizq = math.asin(y / math.sqrt(L**2-B.y**2))
          print("Pder=",int(Pder*360/(2*pi)))
          print("Pizq=", int(Pizq*360/(2*pi))+360)
          print("Distancia=",B.x-planos[i].dmin)
          print("Grado=",planos[i].grado)
          k+=1
```

```
            Da=D
            if W!=0:
                break
        r+=1
    if W==0:
        W=1
        break
```

8.2 Visualisation

```
fig = matplotlib.pyplot.figure()
ax = fig.add-subplot(111, projection="polar")
for i in range(M):
    if planos[i].dmin <= L and planos[i].dmin >= -L:
        ax.plot(i*(360/M)*2*pi/360,planos[i].dmin,'o',mfc='black')

ax.plot(0,B.x,'s',mfc='blue')
ax.plot((m*2*pi)/360,B.x-m2,'D',mfc='red')
plt.show()
A.x=B.x-planos[m].dmin
dibujacat(A,B,calculo(A,B,L,planos[m]))
dibujaobs(planos[m].obs1)
dibujaobs(planos[m].obs2)
dibujaobs(Suelo)
show()
```

8.3 Auxiliary algorithm with movement at the target point. Version 1

```
def calculo(A,B,L,plano,xmin,pos,pos2):
    try:
        obs2=plano.obs1
        obs4=plano.obs2
        suelo=plano.suelo
        lmax=L
        B.y=B.y+pos2
        for A.y in np.linspace(0.2,pos, 100):
            flag=0
            lmin=math.sqrt(((B.x-A.x)**2)+((B.y-A.y)**2))
            Pint=int((2+N*((B.x-A.x)/(B.x-xmin)))/2)
            x=np.linspace(lmax, (lmax+lmin)/2, Pint)
            y=np.linspace(lmin,(lmax+lmin)/2, Pint)
            for l in vectbisect(x,y):
                if(flag==0 and choque(suelo,A,B,l)==0):
                    lmax=l
                    flag=1
                elif(obs4.pol(A.x)<=0 and choque(obs4,A,B,l)==0 and choque(suelo,A,B,l)==0 and choque(obs2,A,B,l)==0):
                    plano.dmin=B.x-A.x
                    return l
        return "Fallo"
    except ValueError:
        return "Fallo"
```

8.4 Auxiliary algorithm with movement at the target point. Version 2

```
def calculo(A,B,L,plano,pos,pos2):
    try:
        obs2=plano.obs1
        obs4=plano.obs2
        suelo=plano.suelo
        B.y=B.y+pos2
        Zx=obs2.pos[0,np.where(obs2.pos[1,]==np.amin(obs2.pos[1,]))[0]]
        Zy=np.amin(obs2.pos[1,])
        C1=-2*Zx
        C2=(Zy**2)+(Zx**2)-(L**2)+2*L*math.sqrt(((B.y-Zy)**2)+((B.x-Zx)**2))-((B.y-Zy)**2)-((B.x-Zx)**2)
        xm1=(-C1-math.sqrt((C1**2)-4*C2))/2
        xm2=B.x-math.sqrt((L**2)-((B.y-A.y)**2))
        xmin=max(xm1,xm2,A.x)
        for A.y in np.linspace(0.2, pos, 100):
            for A.x in np.linspace(xmin, B.x - ((B.x - xmin) / N), N - 1):
                lmax = L
                lmin = math.sqrt(((B.x - A.x) ** 2) + ((B.y - A.y) ** 2))
                Pint = int((2 + N * ((B.x - A.x) / (B.x - xmin))) / 2)
                for l in np.linspace(lmin, lmax, Pint * 2):
                    if(obs4.pol(A.x)<=0 and choque(obs4,A,B,l)==0 and choque(suelo,A,B,l)==0 and choque(obs2,A,B,l)==0):
                        plano.dmin=B.x-A.x
                        return l
        return "Fallo"
    except ValueError:
        return "Fallo"
```

8.5 Algorithms with floating obstacles

Creation of floating obstacles:

```
def creaobsflot():
obsx=obsflot()
a=B.x/2
a=a-2+4*np.random.rand(1)
b=B.y/3
b = b - 0.5 + 1 * np.random.rand(1)
obsx.pos=np.array([[a+a/10,a+a/6,a+a/4,a+a/10],[b+b/10,b+b/3,b+b/6,b+b/10]])
obsx.sit="Flotante"
return obsx


class plano():
    obs1=creaobs("Aereo")
    obs2=creaobs("Terrestre")
    suelo=Suelo
    obsf=creaobsflot()
    obsf2=creaobsflot()
    obsf3=creaobsflot()
    grado=1
    dmin=10000
```

Collision function:

```
def choque(Obs,A,B,L):
    try:
        k=0
        if Obs.sit == "Aereo":
            for i in Obs.pos[0,]:
                if i>=A.x and i<B.x:
                    if catenaria(A,B,L,i)-Obs.pos[1,k] > 0:
                        return 1
                k+=1
            return 0
        elif Obs.sit == "Terrestre":
            k=0
            for i in Obs.pos[0,]:
                if i>=A.x and i<=B.x:
                    if catenaria(A,B,L,i)-Obs.pos[1,k] < 0:
                        return 1
                k+=1
            return 0
        elif Obs.sit == "Flotante":
            k=0
        i=Obs.pos[0,0]
            if catenaria(A,B,L,i)-Obs.pos[1,k] > 0:
                m=1
            else:
                m=2
            if m==1:
                for i in Obs.pos[0,]:
                    if i >= A.x and i < B.x:
                        if catenaria(A, B, L, i) - Obs.pos[1, k] < 0:
                            return 1
                    k += 1
                return 0
            else:
                for i in Obs.pos[0,]:
                    if i >= A.x and i < B.x:
                        if catenaria(A, B, L, i) - Obs.pos[1, k] > 0:
                            return 1
                    k += 1
                return 0
    except np.core.-exceptions.UFuncTypeError:
        return 1
```

Auxiliary algortihm:

```
def calculo(A,B,L,plano):
try:
    obs2=plano.obs1
    obs4=plano.obs2
    obsf=plano.obsf
    obsf2=plano.obsf2
    obsf3 = plano.obsf3
    suelo=plano.suelo
    Zx=obs2.pos[0,np.where(obs2.pos[1,]==np.amin(obs2.pos[1,]))[0]]
    Zy=np.amin(obs2.pos[1,])
    C1=-2*Zx
    C2=(Zy**2)+(Zx**2)-(L**2)+2*L*math.sqrt(((B.y-Zy)**2)+((B.x-Zx)**2))-((B.y-Zy)**2)-((B.x-Zx)**2)
```

```
        xm1=(-C1-math.sqrt((C1**2)-4*C2))/2
        xm2=B.x-math.sqrt((L**2)-((B.y-A.y)**2))
        xp1=B.x-B.y*((B.x-obsf.pos[0,2])/(B.y-obsf.pos[1,2]))
        xp2=B.x-B.y*((B.x-obsf2.pos[0,2])/(B.y-obsf2.pos[1,2]))
        xp3 = B.x - B.y * ((B.x - obsf3.pos[0, 2]) / (B.y - obsf3.pos[1, 2]))
        xmin=max(xm1,xm2,A.x,(xp1+xp2+xp3)/3)
        lmax=L
        for A.x in np.linspace(xmin, B.x-((B.x-xmin)/N), N-1):
            lmin=math.sqrt(((B.x-A.x)**2)+((B.y-A.y)**2))
            lmax=(3*L+lmin)/4
            Pint=int((2+N*((B.x-A.x)/(B.x-xmin)))/2)
            Pmed = 0.9981 * (B.x - A.x) - 0.0017 * N + 2.5090
            x=np.linspace(lmax,Pmed, Pint)
            y=np.linspace(lmin,Pmed-1/Pint, Pint)
            for l in vectbisect(x,y):
                if(obs4.pol(A.x)<=0 and choque(obs4,A,B,l)==0 and choque(suelo,A,B,l)==0 and choque(obs2,A,B,l)==0
                and choque(obsf,A,B,l)==0 and choque(obsf2,A,B,l)==0 and choque(obsf3,A,B,l)==0):
                    plano.dmin=B.x-A.x
                    return l
        return "Fallo"
    except ValueError:
        return "Fallo"
```

8.6 Auxiliary algorithm with floating obstacles. Version 2

```
def calculo(A,B,L,plano):
    try:
        obs2=plano.obs1
        obs4=plano.obs2
        obsf=plano.obsf
        obsf2=plano.obsf2
        obsf3 = plano.obsf3
        suelo=plano.suelo
        Zx=obs2.pos[0,np.where(obs2.pos[1,]==np.amin(obs2.pos[1,]))[0]]
        Zy=np.amin(obs2.pos[1,])
        C1=-2*Zx
        C2=(Zy**2)+(Zx**2)-(L**2)+2*L*math.sqrt(((B.y-Zy)**2)+((B.x-Zx)**2))-((B.y-Zy)**2)-((B.x-Zx)**2)
        xm1=(-C1-math.sqrt((C1**2)-4*C2))/2
        xm2=B.x-math.sqrt((L**2)-((B.y-A.y)**2))
        xp1=B.x-B.y*((B.x-obsf.pos[0,2])/(B.y-obsf.pos[1,2]))
        xp2=B.x-B.y*((B.x-obsf2.pos[0,2])/(B.y-obsf2.pos[1,2]))
        xp3 = B.x - B.y * ((B.x - obsf3.pos[0, 2]) / (B.y - obsf3.pos[1, 2]))
        xmin=max(xm1,xm2,A.x,(xp1+xp2)/2,(xp2+xp3)/2,(xp1+xp3)/2) //New
        lmax=L
        for A.x in np.linspace(xmin, B.x-((B.x-xmin)/N), N-1):
            lmin=math.sqrt(((B.x-A.x)**2)+((B.y-A.y)**2))
            lmax=(3*L+lmin)/4
            Pint=int((2+N*((B.x-A.x)/(B.x-xmin)))/2)
            Pmed = 0.9981 * (B.x - A.x) - 0.0017 * N + 2.5090
            x=np.linspace(lmax,Pmed, Pint)
            y=np.linspace(lmin,Pmed-1/Pint, Pint)
            for l in vectbisect(x,y):
                if(obs4.pol(A.x)<=0 and choque(obs4,A,B,l)==0 and choque(suelo,A,B,l)==0 and choque(obs2,A,B,l)==0
                and choque(obsf,A,B,l)==0 and choque(obsf2,A,B,l)==0 and choque(obsf3,A,B,l)==0):
                    plano.dmin=B.x-A.x
                return l
        return "Fallo"
    except ValueError:
        return "Fallo"
```

8.7 Creation of planes

```
M=1000 //Number of planes
planos=[]
for i in range(M):
    planos+=[plano()]
    planos[i].obs1=creaobs("Aereo")
    planos[i].obs2=creaobs("Terrestre")
    planos[i].obsf1 = creaobsflot()
    planos[i].obsf2 = creaobsflot()
    planos[i].obsf3 = creaobsflot()
    planos[i].grado=i*360/M
```

8.8 Edge detection algorithm

```
aa=0
points=[]
gs=[]
for i in range(90):
    ls=[1000]
    for j in range(N-2):
        if (planos[i].obs2.pos[1,j]==0 and planos[i].obs2.pos[1,j+1]>0 and planos[i].obs2.pos[1,j+2]==0):
            ls2 = [planos[i].obs2.pos[0, j + 1]]
            ls += ls2
    a=min(ls)
    if a - aa > (B.x / N + 0.2) and i!=0 and i!=1 and a!=1000 and aa!=1000:
        points+=[a]
        gs+=[i]
    elif a - aa < (-B.x / N - 0.2) and i!=0 and i!=1 and a!=1000 and aa!=1000:
        points+=[aa]
        gs += [i-1]
    aa = a

    for i in range(90):
    i=i+270
    ls=[1000]
    for j in range(N-2):
        if (planos[i].obs2.pos[1,j]==0 and planos[i].obs2.pos[1,j+1]>0 and planos[i].obs2.pos[1,j+2]==0):
            ls2=[planos[i].obs2.pos[0, j+1]]
            ls += ls2
    a = min(ls)
    if a - aa > (B.x / N + 0.2) and i!=270 and i!=271 and a!=1000 and aa!=1000:
        points+=[a]
        gs += [i]
    elif a-aa<(-B.x/N-0.2) and i!=270 and i!=271 and a!=1000 and aa!=1000:
        points+=[aa]
        gs += [i-1]
    aa=a
    P=np.array(points)
    P=np.amax(P)
    G=np.array(gs)
    G=G[np.where(points==np.amax(points))[0]]
    d=B.x-P
    Ps=np.array([d*math.sin(G*pi/180),B.x-(B.x-d)*math.cos(G*pi/180)])
```

8.9 Edge detection algorithm. Version 2

```
aa=0
points=[]
gs=[]
sm=0
for i in range(90):
    ls=[1000]
    for j in range(N-2):
        j=N-1-j
        if (planos[i].obs2.pos[1,j]>0 and planos[i].obs2.pos[1,j+1]==0 ):
            ls2 = [planos[i].obs2.pos[0, j + 1]]
            ls += ls2
    sm=0
    for j in range(N):
        sm+=planos[i].obs2.pos[1,j]
    if sm==0:
        ls2 = [B.x]
        ls += ls2
    a=min(ls)
    if a - aa > (B.x / N + 0.2) and i!=0 and i!=1 and a!=1000 and aa!=1000:
        points+=[aa]
        gs+=[i-1]
    elif a - aa < (-B.x / N - 0.2) and i!=0 and i!=1 and a!=1000 and aa!=1000:
        points+=[a]
        gs += [i]
    aa = a

    for i in range(90):
    i=i+270
    ls=[1000]
    for j in range(N-2):
        j=N-1-j
        if (planos[i].obs2.pos[1, j] > 0 and planos[i].obs2.pos[1, j + 1] == 0 ):
            ls2 = [planos[i].obs2.pos[0, j + 1]]
            ls += ls2
    sm=0
    for j in range(N):
        sm += planos[i].obs2.pos[1, j]
    if sm == 0:
        ls2 = [B.x]
        ls += ls2
    a = min(ls)
    if a - aa > (B.x / N + 0.2) and i!=270 and i!=271 and a!=1000 and aa!=1000:
        points+=[aa]
        gs += [i-1]
    elif a-aa<(-B.x/N-0.2) and i!=270 and i!=271 and a!=1000 and aa!=1000:
        points+=[a]
        gs += [i]
    aa=a

    try:
    px=points.index(min(points))
    P=(points[px])
    G =(gs[px])
    Gs=G
    d = B.x - P
    Ps = np.array([d * math.sin(G * pi / 180), B.x - (B.x - d) * math.cos(G * pi / 180)])
except ValueError:
    Ps=np.array([0,0])
    Gs=0
```