CYBER-PHYSICAL SYSTEMS AND ROBOTICS

Lab 3. Motion planning

# Preparation

1. The property decorator is used as a setter and getter are used in C++ or other programming languages. This allows for validation, debugging, and more readability in the Python code.
2. (x, y) = (−4, −4) (r,c) = (8,0)

(r,c) = (0,8) (x,y) = (4,4)

# Code

## Creating an admissible heuristic

In order to create an admissible heuristic, the only condition is to make an optimistic guess of how far our robot is from the goal. The easiest way to do this when dealing with a discrete distance, like in our case, is computing the Manhattan distance between any given point and the start. This is the number of steps from the starting point to a certain square without taking walls into account. The following code is a simple implementation of this idea.

def \_compute\_heuristic(self, goal: Tuple[float, float]) -> np.ndarray:  
 *"""Creates an admissible heuristic.  
  
 Args:  
 goal: Destination location in (x,y) coordinates.  
  
 Returns:  
 Admissible heuristic.  
  
 """* map\_rows, map\_cols = np.shape(self.\_map.grid\_map)  
  
  
 heuristic = np.ndarray(shape = (map\_rows, map\_cols), dtype = int)  
  
 goal\_rc = self.\_xy\_to\_rc(goal)  
  
 for i in range(0, map\_rows):  
 for j in range (0, map\_cols):  
 heuristic[i][j] = abs(goal\_rc[0] - i) + abs(goal\_rc[1] - j)  
  
 return heuristic

Code 1: Heuristic computation

As explained before, each grid space is assigned a value depending on its manhattan distance to the start. In Figure 1 we have the resulting heuristic which we will use for the A\* algorithm implemented in the following steps.



Figure 1: Resulting heuristic array

Note that the order is flipped when compared with the map, but it is symmetrical and is only a product of naming rows and columns in a different order. Once we have it we can start calculating the optimal path.

## Searching with A\*

In order to compute the A\* algorithm we followed the following steps:

First, we transform the input data into rc coordinates and obtain information such as the number of rows and columns. We also create empty lists for closed\_list, open\_list and an empty array for the ancestors.

heuristic = self.\_compute\_heuristic(goal)  
  
g = 0  
goal\_rc = self.\_xy\_to\_rc(goal)  
start\_rc = self.\_xy\_to\_rc(start)  
  
f = 0  
  
i = 1  
  
point = (start\_rc[0], start\_rc[1], heuristic[start\_rc[0], start\_rc[1]], g) #Current point  
  
open\_list = []  
closed\_list = []  
  
neighbours = []  
  
map\_rows, map\_cols = np.shape(self.\_map.grid\_map)  
  
ancestors = np.full(shape = (map\_rows, map\_cols), fill\_value = None)  
  
reconstruct\_path = []  
  
ancestors[start\_rc[0], start\_rc[1]] = (start\_rc[0], start\_rc[1], g)

Code 2: Variable initialization

Second, inside a loop that executes until we arrive to the goal, for each direction (top, right, left and bottom) we check if we’re going straight, backwards or turning. After that, depending on the old direction and the new direction, we add to a neighbours list the 4 possible movements with their respective weights depending on the type of movement they make.

while (point[0], point[1]) != goal\_rc:  
 d\_x = point[0] - ancestors[point[0], point[1]][0]  
 d\_y = point[1] - ancestors[point[0], point[1]][1]  
 direction = [d\_x,d\_y]  
  
 new\_direction = [-1,0] #RIGHT  
 a = direction[0] \* new\_direction[0] + direction[1] \* new\_direction[1]  
 c = np.dot(np.asanyarray(direction), np.asanyarray(new\_direction)) # -> cosine of the angle  
  
 if c == 1:  
 neighbours.append((point[0] - 1, point[1], self.\_action\_costs[0])) #STRAIGHT  
 elif c == -1:  
 neighbours.append((point[0] - 1, point[1], self.\_action\_costs[1])) # BACKWARDS  
 elif c == 0:  
 if d\_x == 0 and d\_y != 0:  
 #Se movía en vertical  
 if d\_y > 0:  
 #Se movía aumentando y  
 if new\_direction[0] > 0:  
 #Giro a izquierda  
 neighbours.append((point[0] - 1, point[1], self.\_action\_costs[2]))  
 if new\_direction[0] < 0:  
 #Giro a derecha  
 neighbours.append((point[0] - 1, point[1], self.\_action\_costs[3]))  
 if d\_y > 0:  
 #Se movía disminuyendo y  
 if new\_direction[0] > 0:  
 #Giro a derecha  
 neighbours.append((point[0] - 1, point[1], self.\_action\_costs[3]))  
 if new\_direction[0] < 0:  
 #Giro a izquierda  
 neighbours.append((point[0] - 1, point[1], self.\_action\_costs[2]))  
 if d\_y == 0 and d\_x != 0:  
 #Se movía en horizontal  
 if d\_x > 0:  
 #Se movía aumentando x  
 if new\_direction[1] > 0:  
 #Giro a izquierda  
 neighbours.append((point[0] - 1, point[1], self.\_action\_costs[2]))  
 if new\_direction[1] < 0:  
 #Giro a derecha  
 neighbours.append((point[0] - 1, point[1], self.\_action\_costs[3]))  
 if d\_x < 0:  
 #Se movía aumentando x  
 if new\_direction[1] > 0:  
 #Giro a derecha  
 neighbours.append((point[0] - 1, point[1], self.\_action\_costs[3]))  
 if new\_direction[1] < 0:  
 #Giro a izquierda  
 neighbours.append((point[0] - 1, point[1], self.\_action\_costs[2]))  
 else:  
 neighbours.append((point[0] - 1, point[1], 1)) #FIRST STEP

Code 3: Direction computation and neighbour appending

Then, for every neighbor we check if they are reasonable data (they have real and sensible coordinates) and if they’re on top of an obstacle. If they aren’t, we calculate the new g with the one of the current point and the g corresponding to each neighbor and if that point hasn’t been added to the ancestors matrix or if they’re already in the ancestor matrix but now a new path has been found with a lower g, we take them into the open list and add them to the ancestor list. The second case is there just in the remote case that happens.

for neighbour in neighbours:  
 try:  
 if self.\_map.grid\_map[neighbour[0], neighbour[1]] != 1 and np.sign(neighbour[0]) != -1 and np.sign(neighbour[1]) != -1 and neighbour[0] < map\_rows and neighbour [1] < map\_cols:  
 g = point[3] + neighbour[2]  
 if ancestors[neighbour[0], neighbour[1]] is None or ancestors[neighbour[0], neighbour[1]][2] > g:  
 open\_list.append((neighbour[0], neighbour[1], heuristic[neighbour[0], neighbour[1]] + g, g))  
 ancestors[neighbour[0],neighbour[1]] = (point[0], point[1], g)  
 except:  
 print("Point not in map")

Code 4: Adding neighbours to the open list

Then, we sort the open list depending on the lower f, and after that we move that point to the closed list and our new current point will be that particular point.

neighbours.clear()  
  
open\_list.sort(key = lambda r: (r[2]))  
  
point = open\_list[0]  
  
closed\_list.append(open\_list[0][:2])  
  
print("Number of steps: " + str(i))  
  
i += 1  
  
del open\_list[0]

Code 5: Adding the point to the closed list

When we have exited the loop, we call the reconstruct\_path function and reverse the result obtained from the previous call, because it’s given from goal to start instead of from start to goal.

reconstruct\_path = self.\_reconstruct\_path(start\_rc, goal\_rc, ancestors = ancestors)  
  
reconstruct\_path.reverse()  
  
return reconstruct\_path

Code 6: Calliong reconstruct\_path to obtain the path

## Returning the optimal path

Since we are using the ancestors matrix, returning the optimal path is really simple and easy. We start in the goal point and we find which point discovered the goal, and we add that point to the path. We continue this algorithm until we get to the start point. That’s why we reversed the path in the A\* function.

The path is transformed to xy coordinates because our path smoothing algorithm can work with those kind of coordinates.

def \_reconstruct\_path(self, start: Tuple[float, float], goal: Tuple[float, float], ancestors: np.ndarray) -> \  
 List[Tuple[float, float]]:  
 *"""Computes the trajectory from the start to the goal location given the ancestors of a search algorithm.  
  
 Args:  
 start: Initial location in (x, y) format.  
 goal: Goal location in (x, y) format.  
 ancestors: Matrix that contains for every cell, None or the (x, y) ancestor from which it was opened.  
  
 Returns: Path to the goal (start location first) in (x, y) format.  
  
 """* path = []  
 current = goal  
  
 path.append(self.\_rc\_to\_xy(goal))  
  
 while current != start:  
 current = (ancestors[current[0], current[1]][0], ancestors[current[0], current[1]][1])  
 path.append(self.\_rc\_to\_xy(current))  
  
 return path

Code 7: Reconstructing the optimal path

## From Manhattan to Montecarlo

Once the optimal route has been calculated with the Manhattan distance, it is possible to account for a more realistic movement of the robot. One way it to smooth the obtained path using the following alogorith:

def smooth\_path(path, data\_weight: float = 0.5, smooth\_weight: float = 0.1, tolerance: float = 1e-9) -> \  
 List[Tuple[float, float]]:  
  
 *"""Computes a smooth trajectory from a Manhattan-like path.  
  
 Args:  
 path: Non-smoothed path to the goal (start location first).  
 data\_weight: The larger, the more similar the output will be to the original path.  
 smooth\_weight: The larger, the smoother the output path will be.  
 tolerance: The algorithm will stop when after an iteration the smoothed path changes less than this value.  
  
 Returns: Smoothed path (initial location first) in (x, y) format.  
  
 """* new\_path = [[0 for col in range(len(path[0]))] for row in range(len(path))]  
  
 for i in range(len(path)):  
 for j in range(len(path[0])):  
 new\_path[i][j] = path[i][j]  
  
 change = 1  
 while change > tolerance:  
 change = 0  
 for i in range(1, len(path) - 1):  
 for j in range(len(path[0])):  
  
 old\_path = new\_path[i][j]  
 new\_path[i][j] = new\_path[i][j] + data\_weight \* (path[i][j] - new\_path[i][j])  
 new\_path[i][j] = new\_path[i][j] + smooth\_weight \* (new\_path[i + 1][j] + new\_path[i - 1][j] - 2 \* new\_path[i][j])  
 change += abs(old\_path - new\_path[i][j])  
  
 return new\_path

Code 8: Smoothing the path

The idea is to generate a new series of coordinates which is a balance between:

* Fidelity to the original path
* Optimality in terms of distance

First, the algorithm starts by copying the original path. Then, for every point of the path i, for each coordinate j, the algorithm does 2 things:

1. the coordinate is brought closer to the original point with a weight data\_weight
2. the coordinate is brought closer to the optimal position between the points i-1 y i+1, to reduce the distance of the new path, with a weight smooth\_weight

This whole process is repeated until changes in the new paths between two iterations are smaller than a given value.

Below are the two extreme examples. On the left, the data\_weight = 0 which means nothing is made to stick to the original route. The result is a straight line which minimizes the distance of the new path. On the right (smooth\_weight = 0), nothing forces the path to be smoothed and one obtains the original path.

Une image contenant texte, carte

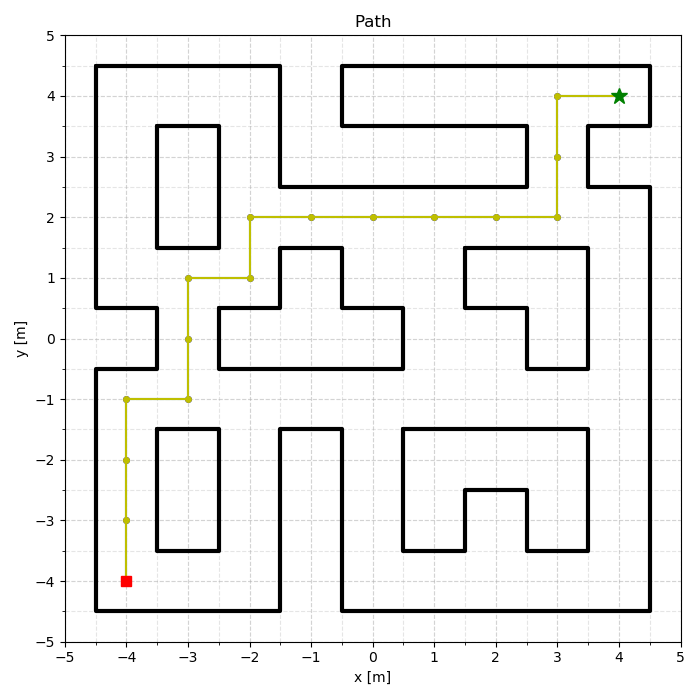
Description générée avec un niveau de confiance très élevé

Figure 2: Results of smoothing the path with the two extreme weights

Of course, the idea is to use a set a parameter which is a trade-off. With these maze’s dimensions (it depends on the width of the corridors), a satisfactory set is data\_weigth = 0.1, smooth\_weight = 0.05. The limit to smoothing is to avoid the corners. It gives the following result:

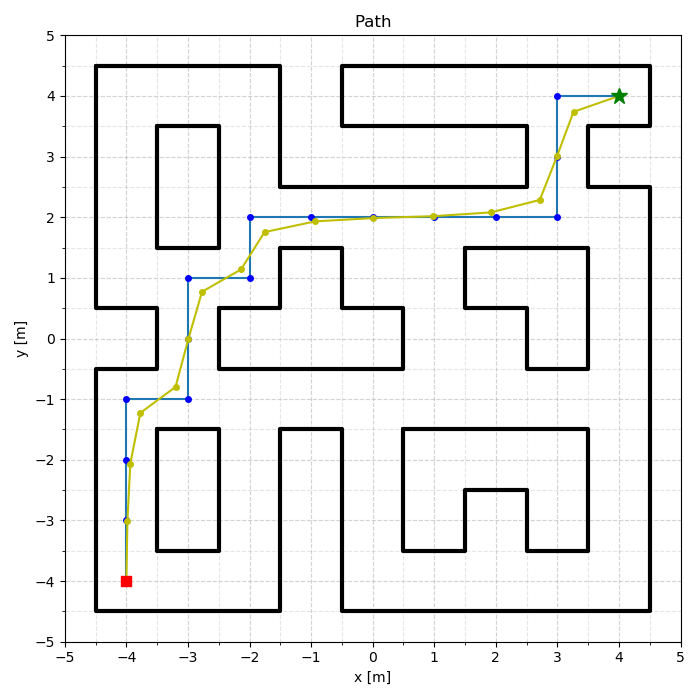


Figure 3: Smoothing the path with a good set of parameters

## 5. A-star was born naïve

A heuristic allows to choose preferentially to expand some points before others, knowing there are closer to the goal. With no heuristic, the algorithm will naively expand points one after the other, until finding the goal. This possibility was implemented thanks to an additional attribute to the Planning class:

for i in range(0, map\_rows):  
 for j in range (0, map\_cols):  
 if self.\_naive:  
 heuristic[i][j] = 0  
 else:  
 heuristic[i][j] = abs(goal\_rc[0] - i) + abs(goal\_rc[1] - j)

Code 9: Allowing for a naïve approach

The path shown before was found in 35 steps using the Manhattan distance heuristic. On the other hand, with no heuristic, it took 51 steps to find the same path, for a total of 55 points to discover! Almost every point had to be expanded, including the ones far away from the objective.

## Sensitivity analyses

First, let’s look at the influence of changing the costs of the 4 actions: going forward, backward, left, right. In the following figure, an equal cost has been specified to every movement. The optimal path is also the shortest one, of cost 6.

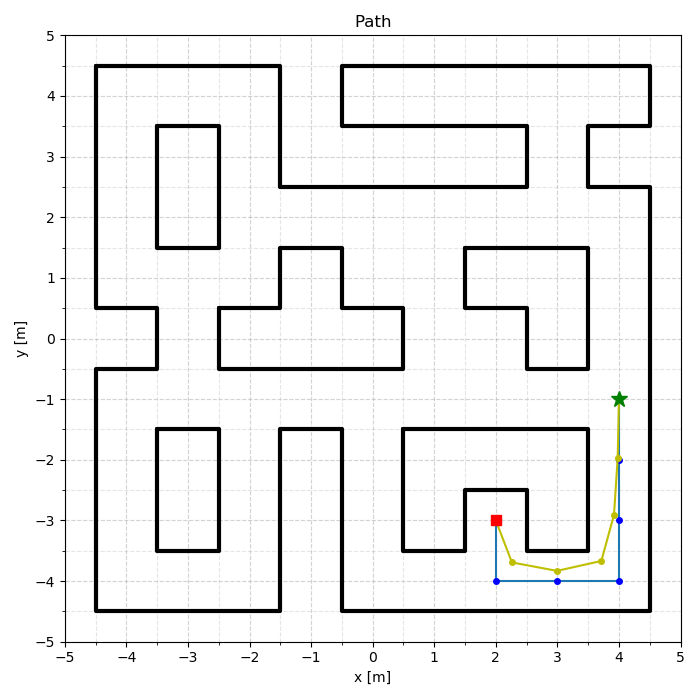


Figure 4: Optimal path with unitary cost for every movement

On the other hand, if we decide to penalize the action of turning left, necessary to follow the above path, the result will no longer be the shortest one:

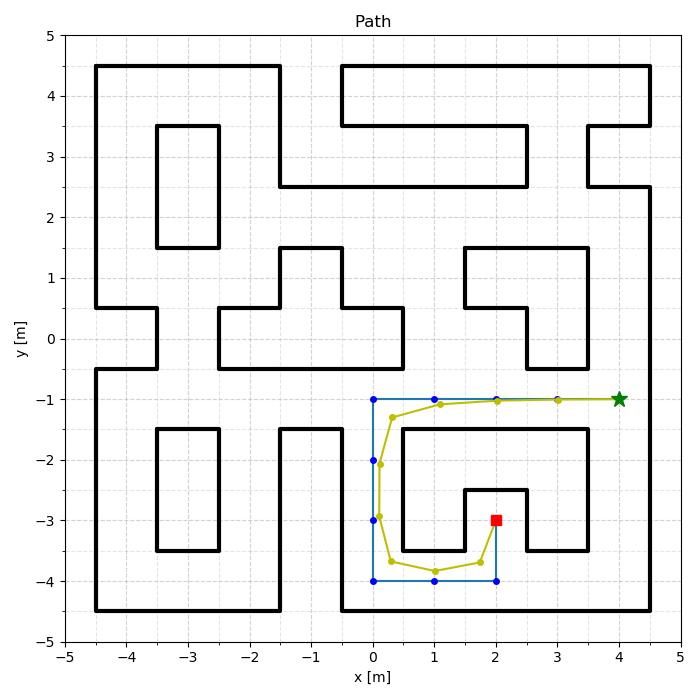
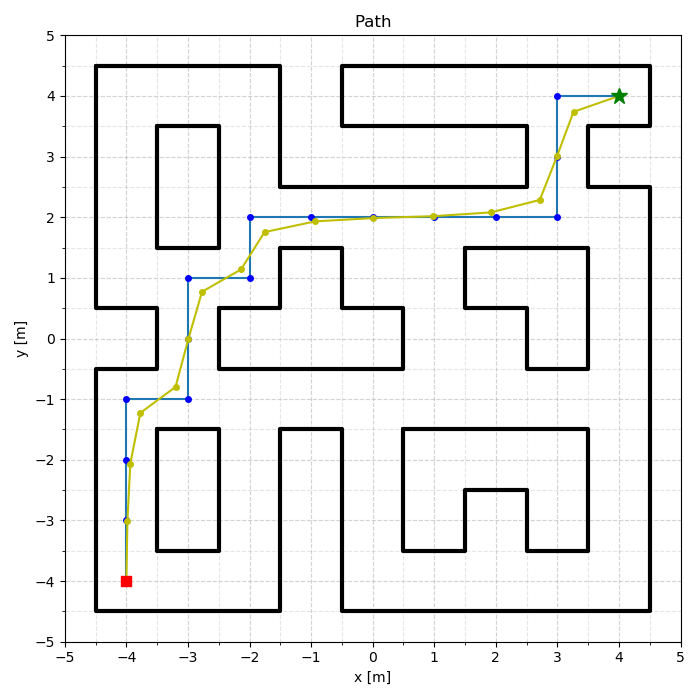


Figure 5: Optimal path with cost of turning left equal to 6

Here, the cost of the path is 10 but it’s smaller than what would have costed the first one (16)

As we have shown before, in 2.4, tweaking the data\_weight and smooth\_weight allow to switch between a closer to the original or a shorter path.

One interesting thing is that it is not only the ratio between the two weight which is important. The absolute value is also critical. Below, smooth\_weight = data\_weight / 2 on the 2 figures. Yet, the result is notably different:

Une image contenant texte, carte

Description générée avec un niveau de confiance très élevé

Figure 6: Comparison with weights (0.1 ; 0.05) and (1 ; 0.5)

The reason for that is simple: the changes are applied sequentially, ending with the distance optimization phase. Even if first we choose to set the new point were the original is (data\_weight = 1), setting afterwards the point to 50% of the optimal position in terms of distance is huge !

After a closer look, when keeping the ratio data/smoothing to 2, the result is quite stable until reaching weight\_smooth = 0.4. Then the effect of smoothing is too strong. And we saw that in the algorithm it had the latest word !

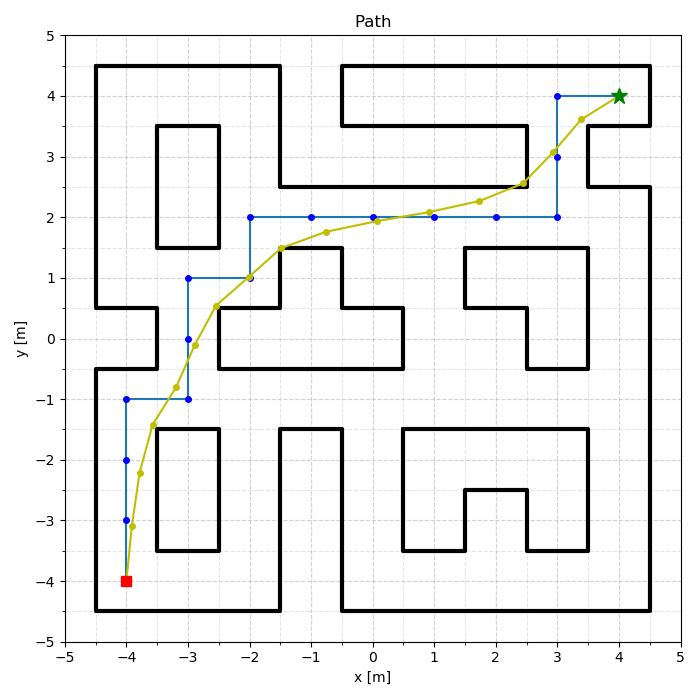


Figure 7: Keeping the same ratio but increasing too much the smoothing quickly modifies the path