FIELD AND SERVICE ROBOTICS (FSR) – a.y. 2023/2024

Homework N.1

Vincenzo Palomba
P38000180

University of Naples Federico II
Department of Electrical Engineering and Information Technology

Abstract

The report summarizes the completion of the initial homework assignment, consisting of three questions and two exercises. Topics covered include underactuation, degrees of freedom and implementation of motion planning algorithms. Notably, the Rapidly-exploring Random Tree (RRT) algorithm was utilized for path planning and a numerical navigation function was developed.

ATLAS: underactuated vs fully actuated

- **While standing, ATLAS is fully actuated.** *True.*
  While standing, the robot is fully actuated because it can impose any acceleration in every direction. This assertion holds true under the condition that Atlas's actuators can produce unbounded torques. This implies that for any given acceleration $\ddot{q}$, there exists a set of torque inputs τ that are linearly independent.

- **While performing a backflip, ATLAS is fully actuated.** *False.*
  During a backflip, the robot cannot impose arbitrary accelerations since it is mid-air. In such a scenario, there is no set of torque inputs that can instantaneously generate the required acceleration.

DOF of a planar and a spacial mechanism

- **Planar mechanism**.
  The mechanism is composed by $N = 6 + 1$ link plus the base, presenting $J = 9$ joints in which one of them has been counted two times, due to the fact that is in common with more than two links. Moreover we count 7 + 1 revolute joint and 1 prismatic one. For each revoute or prismatic joint we add 1 DoF to the formula.

  Applying the Grubler's formula, we obtain:

$$DoFs = 3(N - 1 - J) + \sum_{i=1}^{J} f_i \tag{1}$$

$$DoFs = 3(7 - 1 - 9) + 9 = 0 \tag{2}$$

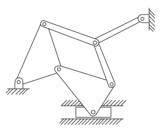
  So, the mechanism is locked and consequently the topology of his configuration space is null.

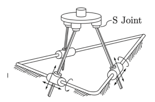
- **Spatial mechanism**.
  The mechanism is composed by $N = 4 + 1$ link plus the base, presenting $J = 6$ joints, in which we can count 3 spherical joint and 3 cylindrical plus prismatic. For each spherical joint we add 3 DoFs, 2 for the cylindrical ones and finally 1 for prismatic ones. Applying the Grubler's formula, we obtain that the DOF are 6.

$$DoFs = 6(N - 1 - J) + \sum_{i=1}^{J} f_i \tag{3}$$

$$DoFs = 6(5 - 1 - 6) + 18 = 6 \tag{4}$$

  In this case the topology of the configuration space is equal to $R^3 \times T^3$


(a) Planar Mechanism



(b) Spatial Mechanism

Underactuated or fully actuated?

- **A car with inputs the steering angle and the throttle is underactuated.** *True.*

  A vehicle controlled by the steering angle and throttle inputs is considered underactuated due to having only two control inputs, while possessing at least three degrees of freedom provided by the planar rigid body frame.

- **The KUKA youBot system is fully actuated.** *True.*

  The robot is fully actuated, in fact it has the possibility to impose arbitrarily any acceleration in every direction because the wheels of the robot are omni directional.

- **The hexarotor system with co-planar propellers is fully actuated.** *False.*

  The hexarotor can't impose arbitrarily any acceleration, because no torque τ can instantaneously generate an acceleration that is not parallel or perpendicular to the hexarotor's body.

- **The 7-DoF KUKA iiwa robot is redundant.** *True.*

  Because for the motion in space are necessary only 6 DOF while the iiwa has 7 DOFs.

RRT algorithm

To solve this problem, we'll implement the *Rapidly-exploring Random Tree (RRT) algorithm* in MATLAB to find a path for a point robot from a start point $q_{\text{start}} = [30, 125]$ to a goal point $q_{\text{goal}} = [135, 400]$ in the given map.

First of all the tree is initialized with the starting point q_i and we also set the maximum number of iterations k to a desired value, same as the stepsize distance δ.

The iterative part of the algorithm relies on the following steps:

Generation of a random configuration point q_r as:

$$q_r = \begin{bmatrix} x_r \\ y_r \end{bmatrix} \tag{5}$$

using the MATLAB command *randi*:

```
qr = [randi(size(map, 2)), randi(size(map,
     1))];
```

The next step consists into selecting the nearest point q_{near} from the tree to q_r. This is accomplished by calculating the norm between each point in the tree and the current random point, and then selecting the tree point with the minimum norm as the nearest configuration.

```
for i = 1 : size(G(:,1))
    normVector(i) = norm(G(i,:) - qr);
end
% finding the minimum norm
[~, minIndex] = min(normVector);
% the closest point is the one with the
    same index of the minimum norm
% in the norm vector
qNear = G(minIndex, :);
```

Now, let's compute the new configuration point q_{new}, that is a point taken at distance δ from q_{near} orientated in his direction. Consider:

$$q_{near} = \begin{bmatrix} x_{near} \\ y_{near} \end{bmatrix} \tag{6}$$

With the following relation let's compute the orientation β:

$$\beta = atan_2(y_r - y_{near}, x_r - x_{near}); \tag{7}$$

From which it yields:

$$q_{new} = q_{near} + \delta \begin{bmatrix} cos(\beta) & sin(\beta) \end{bmatrix} \tag{8}$$

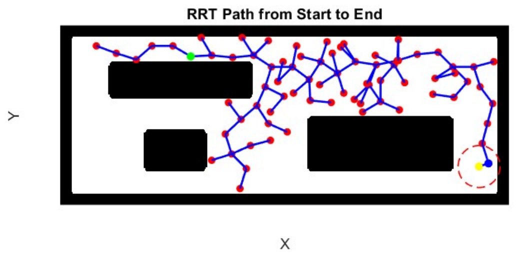
Finally we can add the new point to the tree, if and only if it belongs to the map and there are no collision points in the rectangular area between q_{near} and q_{new}. The *collision check* relies on the concept that, considering the grid nature of the map, it is not always possible to define a straight sequence of cell between two point, moreover ensuring that the rectangle area between q_{near} and q_{new} is free of obstacles allow to select whatever path we want in the same area.

```
% Checking if all the blocks inside the
    rectangular area between qNear
% and qNew are 1, so without obstacles
collisionFree = all(map(min(qNew(2), qNear
    (2)):max(qNew(2), qNear(2)),...
    min(qNew(1), qNear(1)):max(qNew(1),
        qNear(1))) == 1, 'all');
```

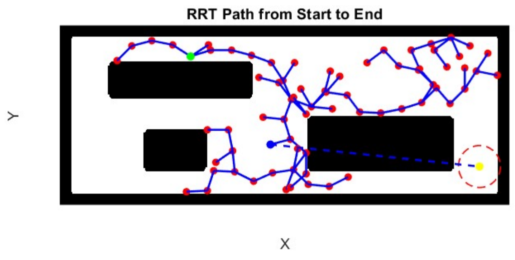
In this step we perform also a check whether or not the newest point is close enough to the goal q_g or not.

The simulation was performed by varying the number of maximum iterations and the distance. Now, let's take the following data as examples: iteration $k = 150$ $\delta = 20$, iteration $k = 300$ $\delta = 10$, iteration $k = 50$ $\delta = 50$.

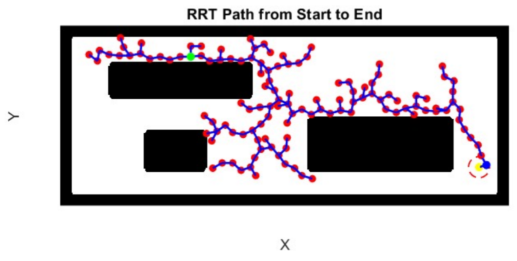
Increasing the number of iterations appears to positively correlate with goal achievement, as increasing the distance δ usually result also in an increased probability of reach the goal. Further investigation using Monte Carlo simulations should be made in order to estimate the correlation factors. One significant drawback of this algorithm is its dependency on randomly distributed points, which results in a lack of directional feedback along the path.



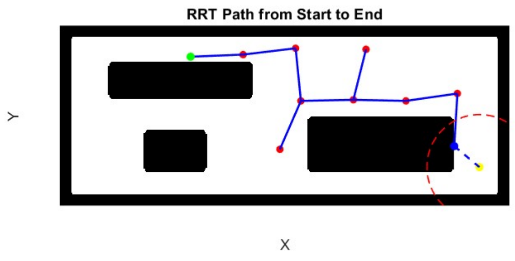
(a) Goal Reached $k = 150$ and $\delta = 20$



(b) Goal not Reached $k = 150$ and $\delta = 20$



(c) Goal Reached $k = 300$ and $\delta = 10$

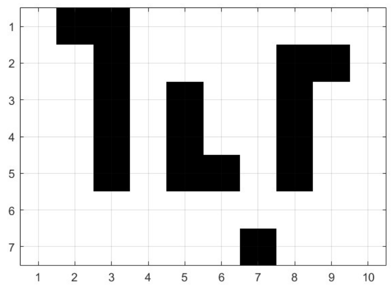


(d) Goal Reached $k = 50$ and $\delta = 50$

Figure 1: RRT path simulations

Numerical navigation function

First of all it has been developed a *"flooding algorithm"* that calculate the cost of reaching each cell from the goal cell, populating the empty map containing only obstacles, represented below:



Using an approach based on the *Breadth first search algorithm*, the map was explored in its entirety, allowing cell-by-cell entry of *costs* (expressed by positive integers), both for the case with 4 adjacent cells and for 8 ones. The choice of using such an algorithm is due to the fact that it starts at the map root and explores all cells at the present depth prior to moving on to the cells at the next depth level.

At the beginning of the algorithm we initialize the queue with the goal cell q_g. At each iteration we *dequeue* the first element that is not yet explored, finding the list of its 4 (or 8) adjacent cells. Then after we update the map values of the adjacent cells, we *enqueue* one new unexplored cell.

```
while size(queue,1) > 0
    % queue pop
    row = queue(1, 1);
    col = queue(1, 2);
    queue(1,:) = [];
    % marking the point as visited
    visited = [visited; [row, col]];
    % searching the four adjacent cells
    list = fourNeighbors(emptyMap, row,
        col);
    % for every cell in list modify the
        respective value in the map
    for i = 1 : size(list,1)
        % if the cell was not yet visited
        if (~ismember(visited(:,1),list(i
            ,1))) | (~ismember(visited
            (:,2),list(i,2)))
            % floading the 4 adjacent
                cells
            emptyMap(list(i,1),list(i,2))
                = emptyMap(row, col) + 1;
            visited = [visited; list(i,1),
                list(i,2)]; % adding the
                cells to visited
            queue = [queue; [list(i,1),
                list(i,2)]]; % update the
                queue
        end
    end
```

```
end
```

Map cost values are calculated if and only if they belong to the map and have not been previously filled.

For the 4 adjacent cells algorithm it has been used the functions *breadth4explorer* and *fourNeighbors* (or for the 8 adjacent cells algorithm the functions *breadth8explorer* and *eightNeighbors*).

```
list = [];
% position of south cell
if row + 1 <= size(map,1) && map(row + 1,
    col) ~= -1
    list = [list; [row + 1, col]];
end
% position of north cell
if row - 1 >= 1 && map(row - 1, col) ~= -1
    list = [list; [row - 1, col]];
end
% position of east cell
if col + 1 <= size(map,2) && map(row, col
    + 1) ~= -1
    list = [list; [row, col + 1]];
end
% position of west cell
if col - 1 >= 1 && map(row, col - 1) ~= -1
    list = [list; [row, col - 1]];
end
```

As for the path finding algorithm, a similar approach was used, based on *depth first* logic. In this case, once a stack is initialized with the starting point q_s, the algorithm iteratively pick the last element entered in the stack (*pop*) and explores his adjacent cells, if they are not already present in the visited list, until the goal q_g has been reached. From the above valid elements, the cell is *pushed* on top of the stack.

```
%until we reach the goal
while (row ~= qg(1) || col ~= qg(1))
    % pop
    row = stack(end, 1);
    col = stack(end, 2);
    stack(end,:) = [];
    % marking the point as visited
    visited = [visited; [row, col]];
    % searching the 8 adjacent cells with
        min value
    list = minNeighbors(map, row, col);
    % for each valid cell update the stack
    for i = 1 : size(list,1)
        if (~ismember(visited(:,1),list(i
            ,1))) | (~ismember(visited
            (:,2),list(i,2)))
            stack = [stack; [list(i,1),
                list(i,2)]];
        end
    end
end
```

Two slightly different strategies were adopted when the stack should be updated: at first, the cell with the lowest cost is chosen so as to minimize the number of hops (function *dfs*). This ensures to choose always the cell with the lowest cost from the adjacent ones, but not always the shortest path. The second strategy (function *depthFirstHeuristic*) involves the concept of heuristic decision, in the sense that at each step the adjacent list is sorted in a descend order: this ensure to add more cells to the stack and chose always the in a sorted way between the lowest cost element not yet visited.

```
    % marking the point as visited
    visited = [visited; [row, col]];

    % searching the 8 adjacent cells with
    list = neighbors(map, row, col, -1);
    % using the heuristic to sort the
        stack in order to pick first the
    % desend order list
    heuristic = heuristicSorting(map, list
        );
    % for each valid cell update the stack
    for i = 1 : size(list,1)
        if (~ismember(visited(:,1),
            heuristic(i,1))) | (~ismember(
            visited(:,2),heuristic(i,2)))
            stack = [stack; [heuristic(i
                ,1), heuristic(i,2)]];
        end
    end

for i = 1 :size(list,1)
    valueList(i) = map(list(i,1), list(i
        ,2));
end
[~, index] = sort(valueList, 'descend');
for i = 1 :size(list,1)
    heuristic(i,:) = [list(index(i),1),
        list(index(i),2)];
end
```

The algorithm aims to be a easiest version of the *dijkstra algorithm* or *A* algorithm*, in order to minimize the path cost.

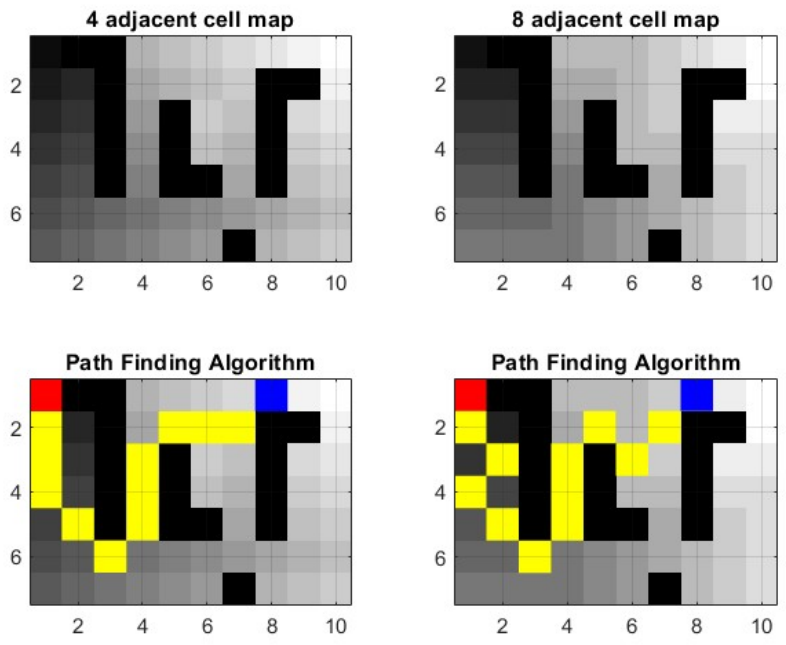
Regarding the 4 cell adjacent map, the following result were accomplish:

algorithm	total path cost	cells traversed
depthFirstHeuristic	107	13
dfs	116	14

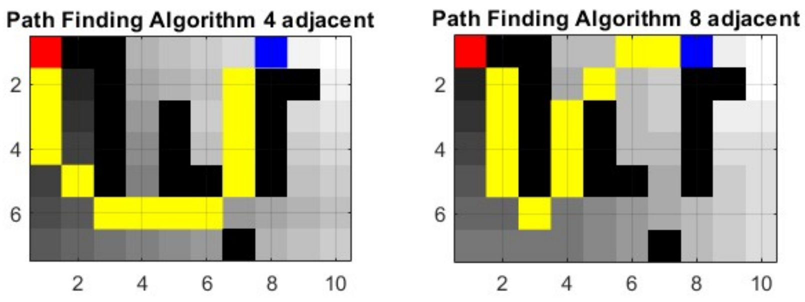
Regarding the 8 cell adjacent map, the following result were accomplish:

algorithm	total path cost	cells traversed
depthFirstHeuristic	78	13
dfs	78	13

As can be seen, the use of map with 4 or 8 adjacent cells is irrelevant in terms of the number of cells crossed. However, there is a difference in terms of the total cost: in fact, it can be seen that the least costly path is the one obtained on the map with 8 adjacent cells. A substantial difference is also made by the algorithm *depthFirstHeuristic* in the 4-adjacent-cell map.



depthFirstHeuristic: 4 adjacent cells (left) and 8 adjacent cells (right)


Path Finding Algorithm 4 adjacent
Path Finding Algorithm 8 adjacent

dfs path: 4 adjacent cells (left) and dfs path 8 adjacent cells (right)

Figure 2: depthFirstHeuristic path vs dfs path

Code instructions

The RRT algorithm code is in the file *"RRT.m"*, while the Numerical Navigation algorithm code is in the file *"numericalNavigationFunction.m"*. Code main and function are in one file for both the algorithm, so you can run the code without problem if you have newer version of MATLAB (from 2016B).