MOTION PLANNING IN C-SPACE

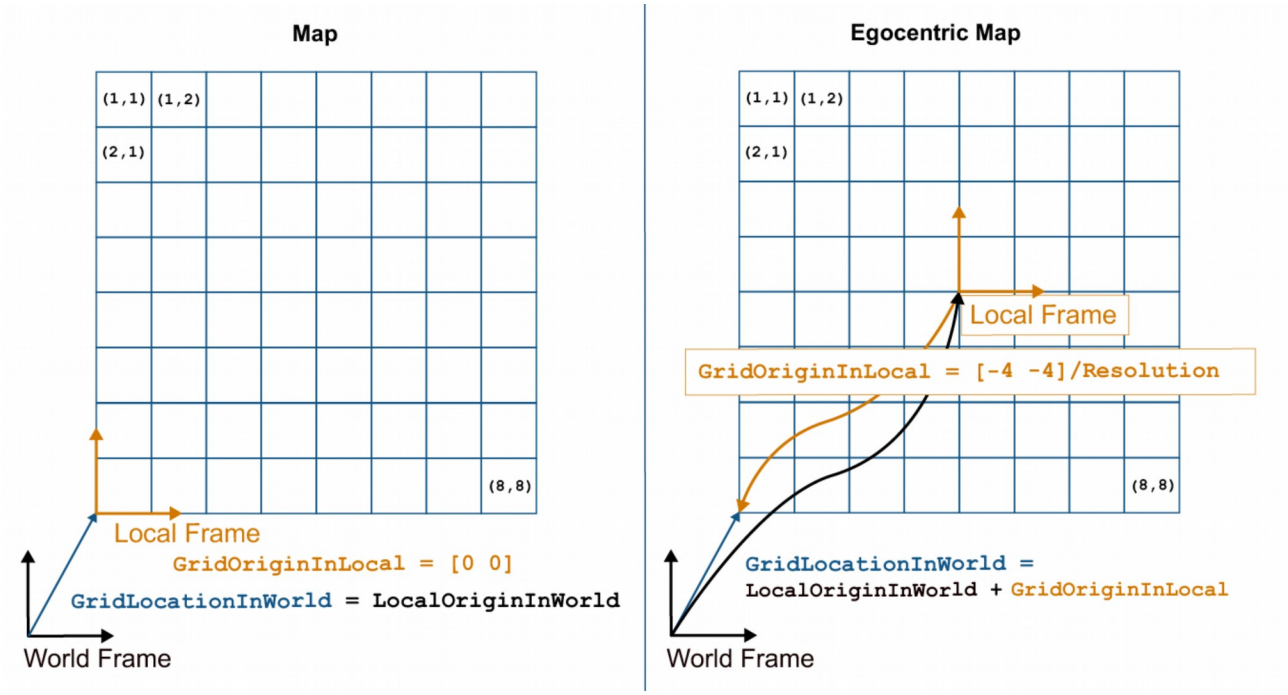
INTRODUCTION

In this exercise, the goal is to perform path planning for the 2 arm manipulator in the Configuration Space created in exercise 5.2. To do so, it is first necessary to convert the C-space into a Binary Map. In this map, the yellow region (which is the region where for that values of joints position q1, q2 , the manipulator and the obstacle are in collision) must be converted into an obstacle in order to perform RRT algorithm in the same way as previously done in Exercise 5.1.
After this conversion, the RRT algorithm can be applied in order to find a Path from start position to goal position, avoiding the forbidden region (the obstacle).
Once a path is found, we have at disposal a tree where a sequence of joints position values (q1, q2) is defined. This sequence is the one that the robot shall follow in order to perform the motion. However, since it is defined in the joints space and not in the 2D cartesian space, a final conversion need to be done.
The path of the manipulator in the X-Y plane can be easily derive substituting in the direct kinematics expression for the robot the values of (q1(i), q2(i)) stored in the sequence of the Shortest Path.
Finally, a graphical simulation of the Path in the X-Y space of the Manipulator is presented.

CODE DESCRIPTION

PART 1: Create the Binary Occupancy Grid

As for the previous exercise, the code start with the robot initialization through the Robotics System Toolbox. Without repetition (explanation of this initialization has already been defined in report for Exercise 5.2 – C-space), the next step is to create a Binary Map corresponding to the C-space.
In a similar way as in the previous exercise, two arrays theta_0 and theta_1 are initialized to create grid points of the C-space (step is 0.05).
Then, a map is initialized with the binaryOccupancyMap function. This function receives in input map dimensions and resolution, that for coherent conversion must have a value equal to 1/step.
This function belongs to the Navigation Toolbox, and for a robot moving in this map three different reference coordinate system exists: World Reference Frame, Grid Reference Frame and Local Reference Frame, as illustrated in the following image:



To properly initialize the Map, it is necessary to define the Grid origin with respect to Local Reference Frame. In our case, this is shifted of [-pi, -pi]. The created Map has the following properties:

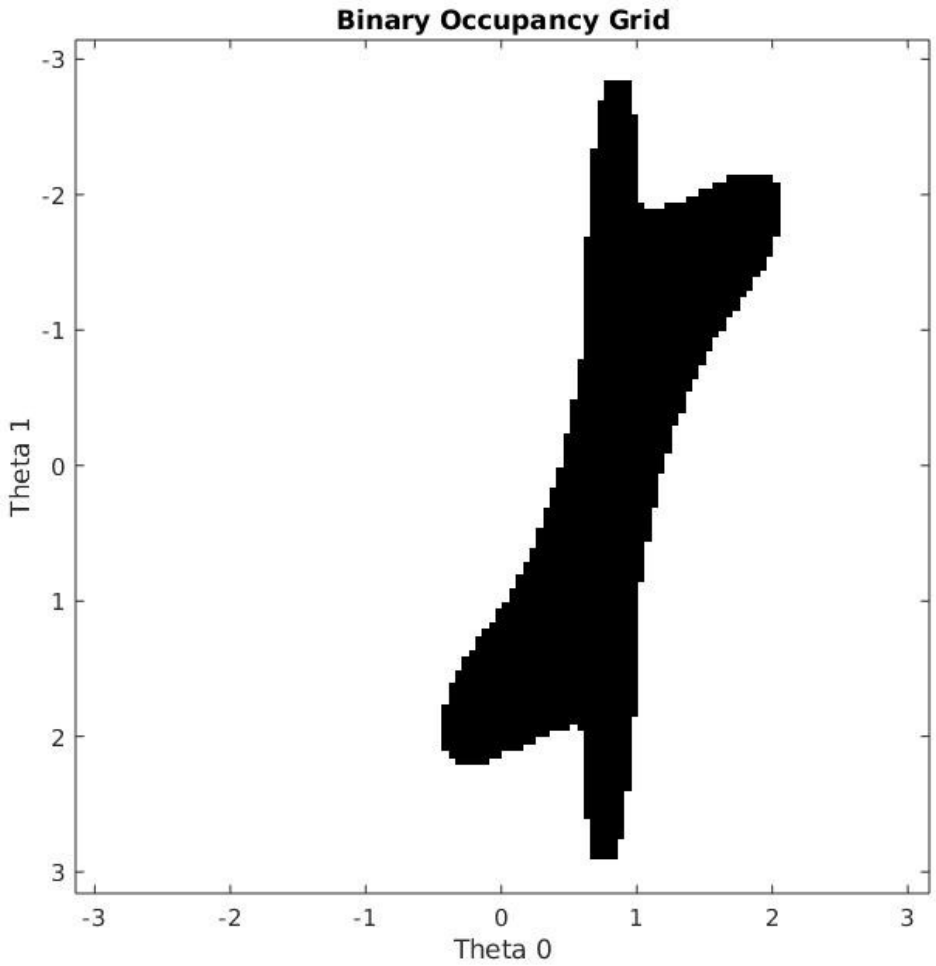
```
map =

  binaryOccupancyMap with properties:


    GridLocationInWorld: [-3.1416 -3.1416]
           XWorldLimits: [-3.1416 3.1584]
           YWorldLimits: [-3.1416 3.1584]
```

```
            DataType: 'logical'
        DefaultValue: 0
          Resolution: 20
            GridSize: [126 126]
        XLocalLimits: [-3.1416 3.1416]
        YLocalLimits: [-3.1416 3.1416]
    GridOriginInLocal: [-3.1416 -3.1416]
LocalOriginInWorld: [0 0]
```

The created map is initialized as empty. In order to add the obstacle to the map, two nested for loops are used exactly at the same way as implemented in Exercise 5.2: for every point in the grid (for every possible position q1, q2) it is checked if collision between robot in corresponding configuration and obstacle occurs. Instead of creating a C-space, in this script if there is collision, through the setOccupancy function, the value of that point in the map is set to 1 (1 correspond to 100% probability to have obstacle). At the end of the loop, the obstacle is completely defined. Through the inflate function, which enlarges the obstacle by an amount defined by the user (0.001 in this case), an additional small threshold is added to the obstacle: this is a common way to make path planning more robust to obstacle collision. The resulting obtained map is the following (notice that for coherence with previous C-space, even in this case the image has been flipped):

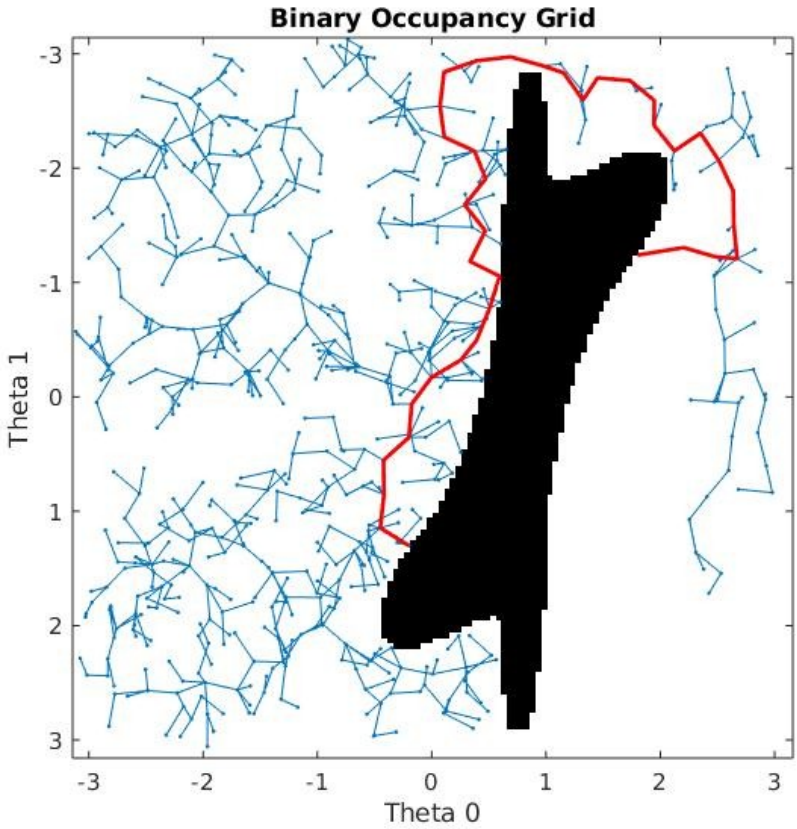


PART 2: Find the Shortest Path in C-Space (RRTStar Algorithm)

The very next step is the implementation of the RRT algorithm. In this case, it was possible to use the already implemented function RRTStar, that without going into too much details, it is an optimized version of basic RRT star that keeps track of nodes distances (cost) at every iteration. To use this function, start position and goal position are defined as three dimensional vectors. 2 dimensions are for the position within the plane, 1 additional dimension is the rotation and even if is not relevant in this application, it is fundamental to define it because RRT does not suppose that is dealing with points but with rigid bodies. Note also that start and goal position have been defined in order to simulate the given image from lecture notes: the value of the joints position corresponding to start and goal have been obtained using the provided twoLinkRobotArmInteractive exploiting inverse kinematics for desired x-y position.

To use RRTStar, it is first necessary to create a state space with the function “stateSpaceSE2” and then a space validator with the function “validatorOccupancyMap()”. After that, some parameters must be passed to the function (ValidationDIstance, the map and the Bounds for the state of the variable).

Finally, the actual RRTStar algorithm can be called with the function “plannerRRTStar(ss,sv)”. In order to make the algorithm works, a MaxIterations value must be defined, together with MaxConnectionDIstance (stepsize) and MaxNumTreeNodes. The optional feature “ContinueAfterGoalReached” can be set to true or to false. In the first case, the algorithm iterates as many times as the maximum iterations value and at the end it returns the Shortest Path from start to goal; if instead it is set to false, th algorithm immediately returns the first found Path, like in the following example:



At the end, a struct pthObj is returned containing the nodes that form the sequence of the ShortestPath. An array nodes containing all the nodes from start to goal is defined and finally the Path, together with all the other nodes and edges of the tree, is plot on the Binary Grid as shown above.

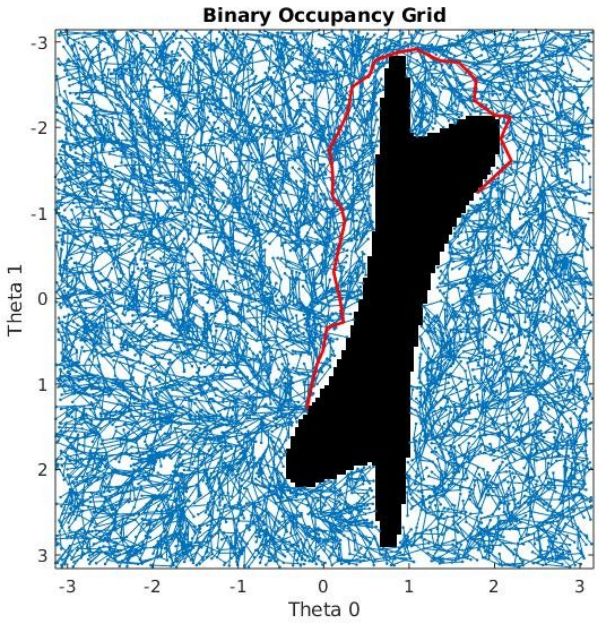
PART 3: Show corresponding Path in x-y Space

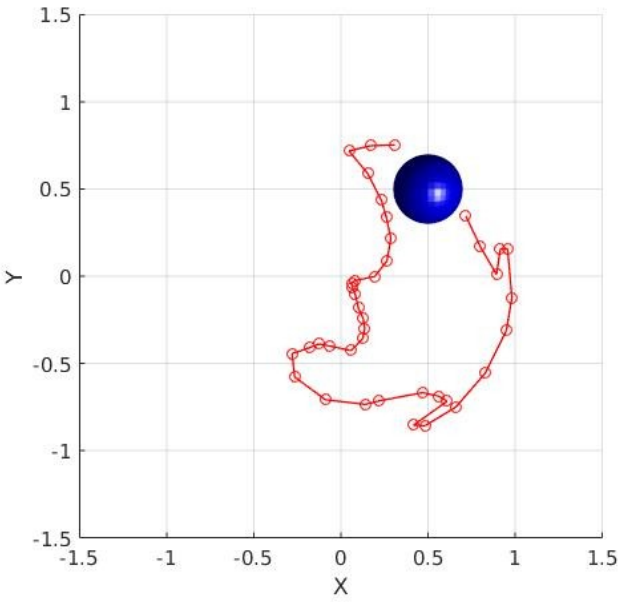
At this point, we have a vector containing the sequence of the states (q1,q2) corresponding to positions that the joints shall assume to perform the motion. Each node, as explained above, has three components, but we are interested only in the first two. Pose_vector is then initialize with all these values, together with two vectors x and y that have the same length of pose_vector. This two vectors are the one were x and y position of every node of the Path will be stored.
To convert joints coordinates into x-y coordinates (the reference is the end effector of the robotic arm) a for loop (with as many iterations as the length of pose_vector is

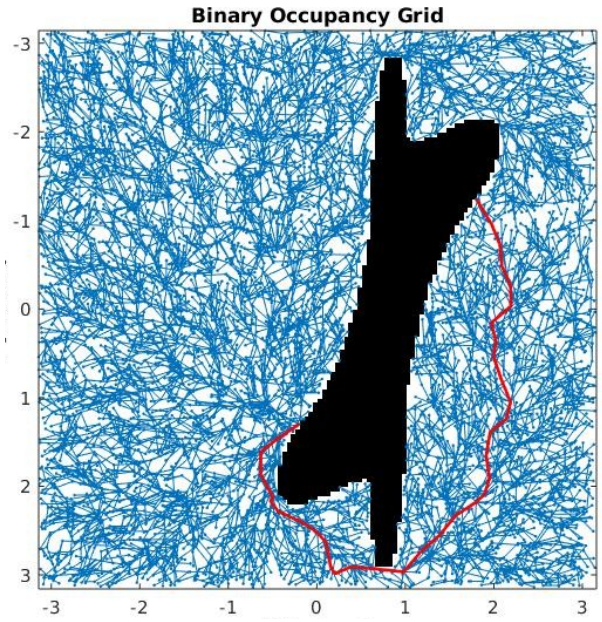
used). Inside, vector q is initialized at every iterations as the value of pose_vector(i); from q1 and q2, it is possible to compute x(i) and y(i) position with the simple formula of direct kinematics for the 2 link robot arm:

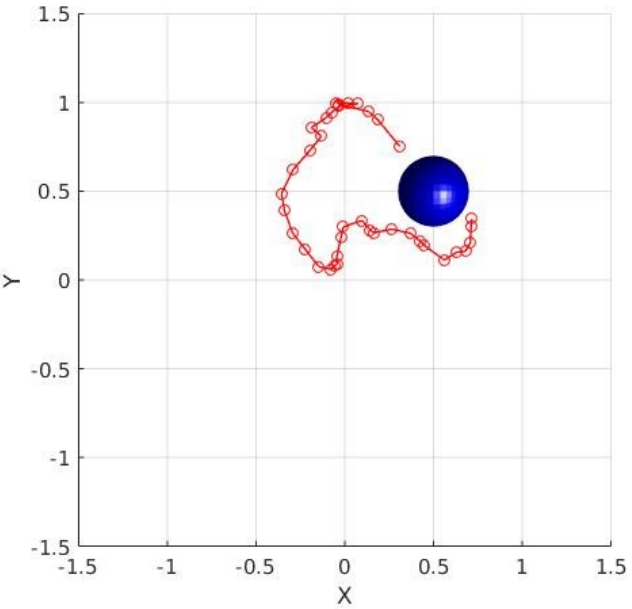
```
x(i) = L1*cos(q1)+L2*cos(q1+q2);
y(i) = L1*sin(q1)+L2*sin(q1+q2);
```

This procedure is repeated two times to display two different figures. The first one represents the corresponding x-y path without the manipulator. Two examples of this figure are reported here, where the first one correspond to a Path where in the C-space the manipulator avoids the obstacle from the top, and the second one from the



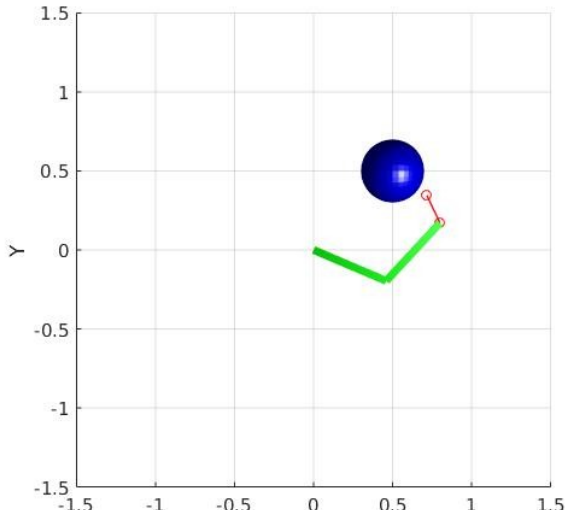


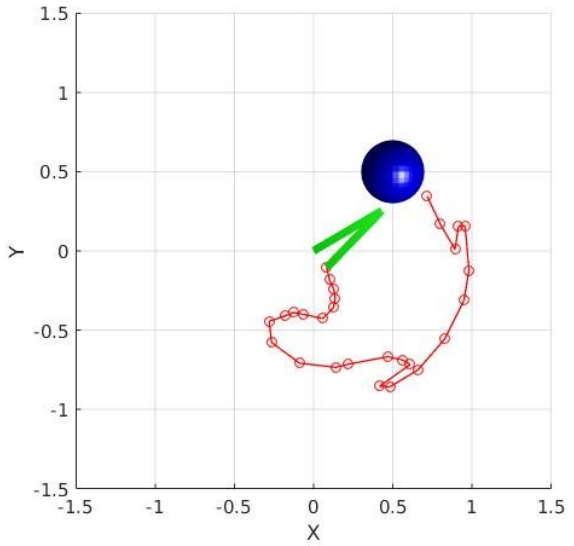


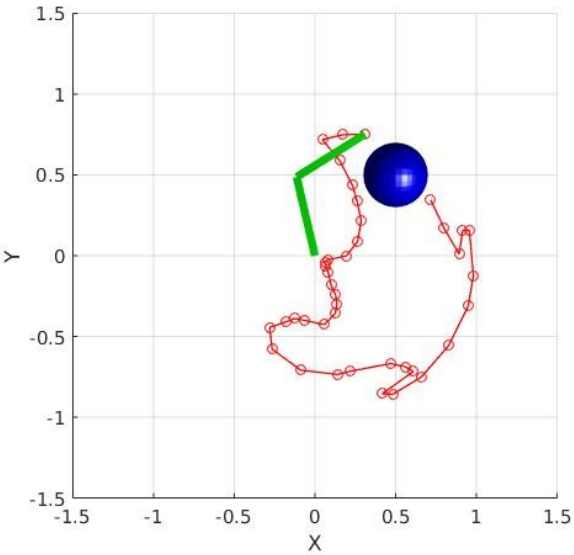


bottom:

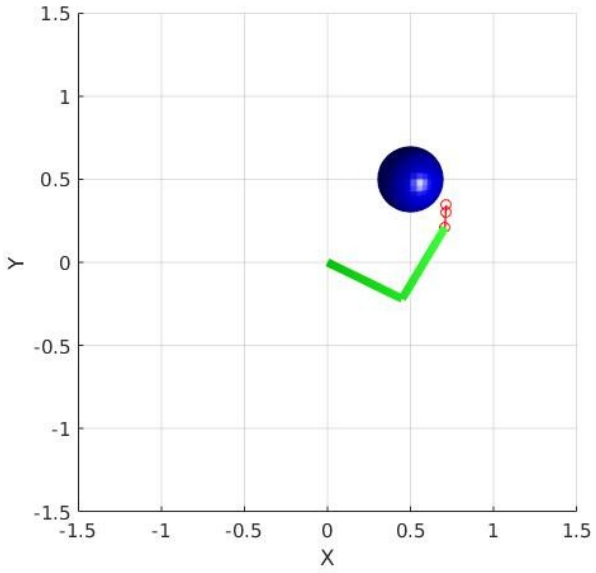
The second figure is instead an animation representing the evolution of the arm position from goal to start. Here in the report I cannot insert the whole animation, but three images are reported that corresponds to the path in the previous images:

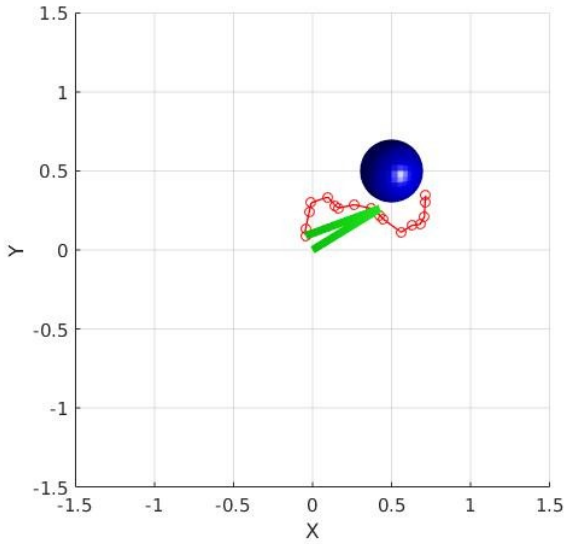


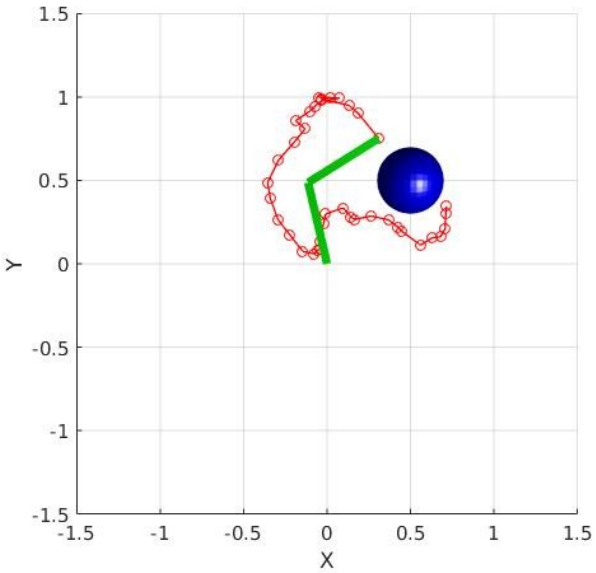




And for the for the other Path:







ANALYSIS OF RESULTS AND FINAL CONSIDERATIONS

In this exercise, even if it is based on a very simple and basic robot arm manipulator, all the fundamental concepts to generate path planning are highlighted and some important considerations can be extracted:

1) As you can see from the resulting images, the robot arm in both Shortest Paths has to pass from a “via point” with a particular configuration, where the end effector is pointing towards joint 0 position. The position in this point is very particular because here the robot is in a configuration close to a kinematics singularity (a configuration where the determinant of the manipulator Jacobian is zero, that means momentaneous loss of 1 degree of freedom). The alternative to this via point would be to avoid the sphere with an anticlockwise rotation of joint 0. However, this solution is not found in this implementation because it is not the Shortest Path to reach desired Goal position.

2) The inflate function used for the definition of the object is particularly useful in cases like this, where the arm has to pass very close to the obstacle.

3) The Path found can be further improved, increasing the number of iterations of RRTStar algorithm. In fact, you can see that there are some not required oscillations of the arm that could be avoided

4) The found procedure can be easily extended to cases where robot has more than two joints and the space is no more in 2D but in 3D. Of course, computational effort will increase, but validity of the basic steps of this procedure is proved.

ADDITIONAL NOTES

As you can see, at the beginning of the script I added a comment because there seem to be some bugs with the Robotics Package. In fact, this Release is very recent and one of the problem that I have encountered is that if I run all the script the figures generated with show(robot) sometimes are divided into two figures. This might by my fault, but I tried to fix it in many ways and still the problem persists.

Instead, if the code is run section after section, no problem occurs.