A* and 3D Configuration Space

Part I: Star Algorithm to build 3D Configuration Space

I have implemented again the Star Algorithm to compute the 3-dimensional configuration space $Q$ starting from the workspace $W$ and the robot shape $R$. Since the robot can now translate and rotate, the $Q$ space is $SE(2)$ or $\mathbb{R}^2 \times S^1$. The obstacles $QO_i$ in the configuration space are given by the following relationship:

$$QO_i = \{ q \in Q \mid R(q) \cap WO_i \neq \emptyset \}$$

where the configuration $q$ is a vector containing the two coordinates $x$ and $y$ of a reference point on the robot and its rotation angle $\alpha$ about this point.

If we assume that the robot and the obstacles in the workspace are convex polygons, the Star Algorithm computes the obstacles in the workspace using the normal vectors of each side of the polygons and a given robot’s rotation angle $\alpha$. I have used the Matlab function `norm = c_norm(tang, dir)` that returns the normal vector for each side of a polygon. The function is the same used in the Homework #3.

The algorithm sorts the sides of the robot and of one obstacle by increasing angles of normal vectors. Thereafter it builds the new convex polygon adding all the sides of the robot and of the obstacle in the calculated sequence. In order to define the position of the new obstacle in the configuration space for each robot’s rotation angle $\alpha$, I define the vertex with the higher $x$-value in the non rotated configuration as reference point on the robot. This reference points does not change during the rotation of the robot.

In order two define the obstacle reference point, I search for the obstacle vertex that the robot’s reference point can touch for a given rotation angle $\alpha$. I have implemented a
Matlab function \texttt{inside = vertex\_check(Xr, Yr, Xo, Yo)} that performs this check. After finding the correct reference point on the obstacle, I can transform an obstacle of the workspace in the correspondent obstacle in the configuration space using the modified Matlab function \texttt{[X, Y, num] = q\_obstacle(robot\_X, robot\_Y, robot\_num, robot\_n, obstacle\_X, obstacle\_Y, obstacle\_num, obstacle\_n, R\_ref\_i, O\_ref\_i)}. The new parameter \texttt{R\_ref\_i} and \texttt{O\_ref\_i} are the index of the reference point on the robot and on the obstacle. In order to understand which vertex of the new obstacle is reference point, that means must be on the same position of the reference point of the old obstacle, when I insert the sides to build the new polygon I memorize the position of the reference point of the old obstacle and of the robot. For each of the two points, I will have two possible positions on the new obstacle. Then:

- \textbf{if} the two positions corresponding to the reference point of the old obstacle are the same, the new reference point is this position;
- \textbf{elseif} the two positions corresponding to the reference point of the robot are the same, the new reference point is this position;
- \textbf{elseif} the two positions corresponding to the reference point of the old obstacle are different, but one of these is the same of one of the two positions corresponding to the reference point of the robot, the new reference point is this position;

I repeat the whole procedure for each possible robot’s rotation angle \( \alpha \). In the following picture a representation of the Q-obstacle.
When the program starts, you can push the button “Draw Obstacle” to draw an obstacle and set with the left mouse button the vertices of the obstacles within the white area (robot workspace $W$). To end the process you can click with the right mouse button somewhere. To draw the robot you can push the button “Draw Robot” and set with the left mouse button the vertices of the robot within the white area. To end the process, click somewhere with the right mouse button. A black point will be placed on the reference vertex of the robot. To set the end configuration, push the button “End Configuration”, and set with the left mouse button the point. Than insert the value of the final rotation alpha in the text box “Angle”. To calculate the 3D-configuration space and the path, push the button “Calculate Path”.
Part II: A* Algorithm

In order to compute the path within the configuration space, I have implemented the A* algorithm. The problem by a 3-D grid search is the huge computational cost. In fact at each step we have 26 neighbors to take into account. Therefore I have based my implementation on a multiscale pyramid of bitmap arrays.

We define the following functions:

- \( \text{star}(n) \): represents the set of node which are adjacent to \( n \); I have implemented a Matlab function \( \text{star}_x = \text{expand}(x, i_{\text{max}}, j_{\text{max}}, k_{\text{m}}) \) that takes into account that the third coordinate is defined over \( S^1 \);
- \( c(n_1, n_2) \): represents the length of the edge connecting \( n_1 \) and \( n_2 \); I have implemented a Matlab function \( c = \text{eval}_c(x_1, x_2) \) that takes into account that the third coordinate is defined over \( S^1 \);
- \( g(n) \): represents the total length of back pointer from \( n \) to \( q_{\text{start}} \);
- \( h(n) \): represents the heuristic cost function which returns the estimated cost of shortest path from \( n \) to \( q_{\text{goal}} \); I have implemented two Matlab functions. The first \( H = \text{init}_\text{dist}(q_{\text{goal}}, H_{\text{in}}, i_{\text{max}}, j_{\text{max}}, k_{\text{max}}) \) computates a wave front expansion of \( q_{\text{goal}} \) using the classic non-Euclidean distance of A*.

However, since the computational cost of the previous method is huge, I have
used the Euclidean distance \( d_{euc}(X, Y, k_{max}) \). Both functions take into account that the third coordinate is defined over \( S^1 \).

- \( f(n) = g(n) + h(n) \): is the estimated cost of shortest path from \( q_{start} \) to \( q_{goal} \) via \( n \).

The Algorithm is the follow:

```
repeat
    Pop \( n_{best} \) from \( O \) such \( f(n_{best}) \leq f(n), \forall n \in O \) and add it to \( C \). I have implemented the Matlab function \([O, C] = pop(O, C, x)\);

    If \( n_{best} = q_{goal} \) EXIT

    Expand \( n_{best} \). For all \( x \in star(n_{best}) \) that are not in \( C \):

    if \( x \notin O \) then
        add \( x \) to \( O \)
    else if \( g(n_{best}) + c(n_{best}, x) < g(x) \) then
        update \( x \)’s backpointer to point to \( n_{best} \)
    end if
```

until \( O \) is empty

The common sampling factor used for the coordinate and the angle is 10. I have also calculated the path in higher resolution levels, but the computation time was up to an hour. Therefore in the movies the path is not a smooth curve. However, it is easy to smooth the calculated path in a post processing operation.

I have implemented this algorithm because I was more interesting start working on a three dimensional configuration space. I think that this is closer to the real task of path planning rather than continuing working in two dimensional configuration space.