Università degli studi di Genova

RRT* for robot path planning

Alberto Remus
Simone Maccio'

March 1st, 2018

Abstract

Rapidly Random exploring Tree star is an efficient algorithm developed in order to find the optimal path from a certain source region to a goal one inside a workspace by avoiding the obstacles in it. Our objective was to use it in order to make the Baxter robot move an object grasped with one of its arm among the workspace filled with some obstacles.

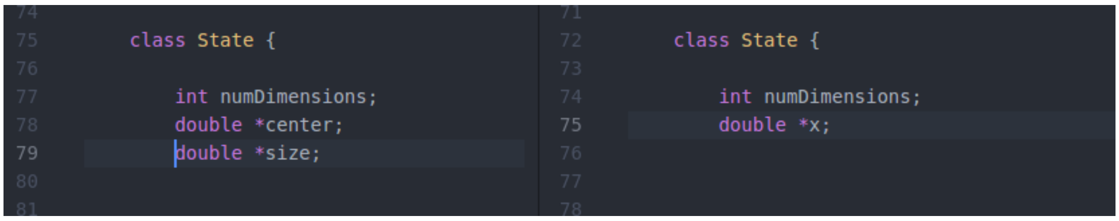
1 RRT* in brief

Basically the algorithm exploits random sampling to progressively explore the overall workspace paying attention to avoid collision and computing the optimal path once the goal region has been reached. It has been demonstrated to be asymptotically optimal from a probabilistic point of view, however its implementation is slightly more complex than its "father", the RRT.

The evolution of RRT, introduced by RRT*, consists in considering at each iteration the minimum cost path among the neighborhood, a set of vertices of the tree at a certain distance from the new vertex selected after the sampling.

2 Starting from the theory

We developed our C++ project modifying the code so that vertices of the RRT* have a size which must be kept into account by the collision avoidance routine, moreover some modifications were introduced in order to grant the sampling occurred inside the workspace. All the results discussed here have been achieved via MATLAB simulations which can be found at our repository



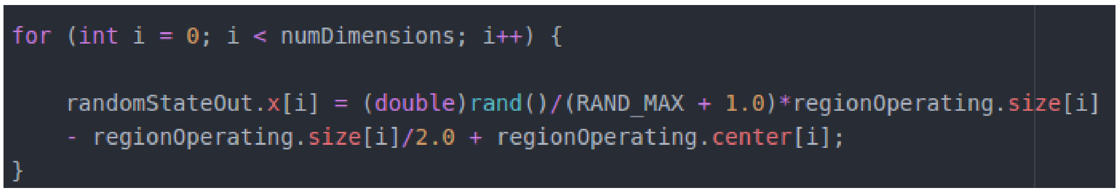
```
class State {

    int numDimensions;
    double *center;
    double *size;
```

```
class State {

    int numDimensions;
    double *x;
```

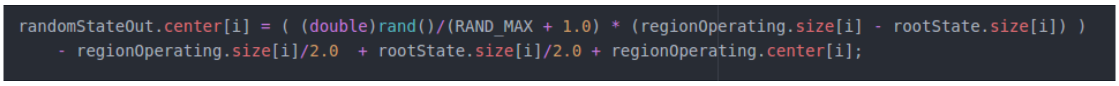
Figure 2.1: Few modifications for real application of the algorithm



```
for (int i = 0; i < numDimensions; i++) {

    randomStateOut.x[i] = (double)rand()/(RAND_MAX + 1.0)*regionOperating.size[i]
    - regionOperating.size[i]/2.0 + regionOperating.center[i];
}
```

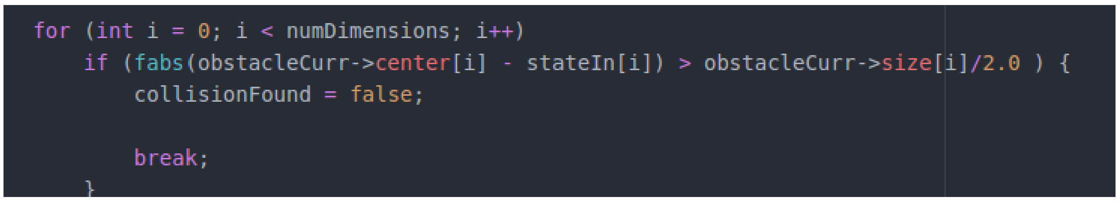
(a) Initial version of the code



```
randomStateOut.center[i] = ( (double)rand()/(RAND_MAX + 1.0) * (regionOperating.size[i] - rootState.size[i]) )
    - regionOperating.size[i]/2.0  + rootState.size[i]/2.0 + regionOperating.center[i];
```

(b) New version of the code

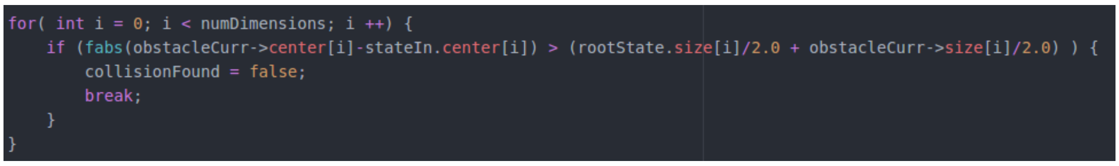
Figure 2.2: Comparison for sampling code



```
for (int i = 0; i < numDimensions; i++)
    if (fabs(obstacleCurr->center[i] - stateIn[i]) > obstacleCurr->size[i]/2.0 ) {
        collisionFound = false;

        break;
    }
```

(a) Initial version of the code



```
for( int i = 0; i < numDimensions; i ++) {
    if (fabs(obstacleCurr->center[i]-stateIn.center[i]) > (rootState.size[i]/2.0 + obstacleCurr->size[i]/2.0) ) {
        collisionFound = false;
        break;
    }
}
```

(b) New version of the code

Figure 2.3: Comparison for collision-checking code

The comparison made between the original code and the one which keeps into account the size showed an increment of 25-30 percent over 20000 samplings on the latter version due to very few lines of code for checking collision with non zero size objects.

Most of the loss of time is due to the introduction of the size of the object, however the overhead in sampleState is negligible compared to the one in extendTo, the routine devoted to interpolate two by two all vertices of the optimal trajectory path which requires many calls to the IsInCollision routine.
After this preliminar work, we analyzed the impact of using RRT* with non zero dimension nodes, finding out some interesting linear relationships by tuning size of the grasped object, the size of the obstacle and the number of random obstacles in the workspace.

We started analysing the relationship between the theoretical probability of collision and the collision rate deriving from the ratio numberOfCollisions over numberOfSamplings finding out how those two quantities are stricly related, in the plots at the next page we show how they linearly depends from the volume of the object, even if we increase the size of the obstacle, in fact we used one obstacle centered in the origin as the 20x20x20 workspace is.

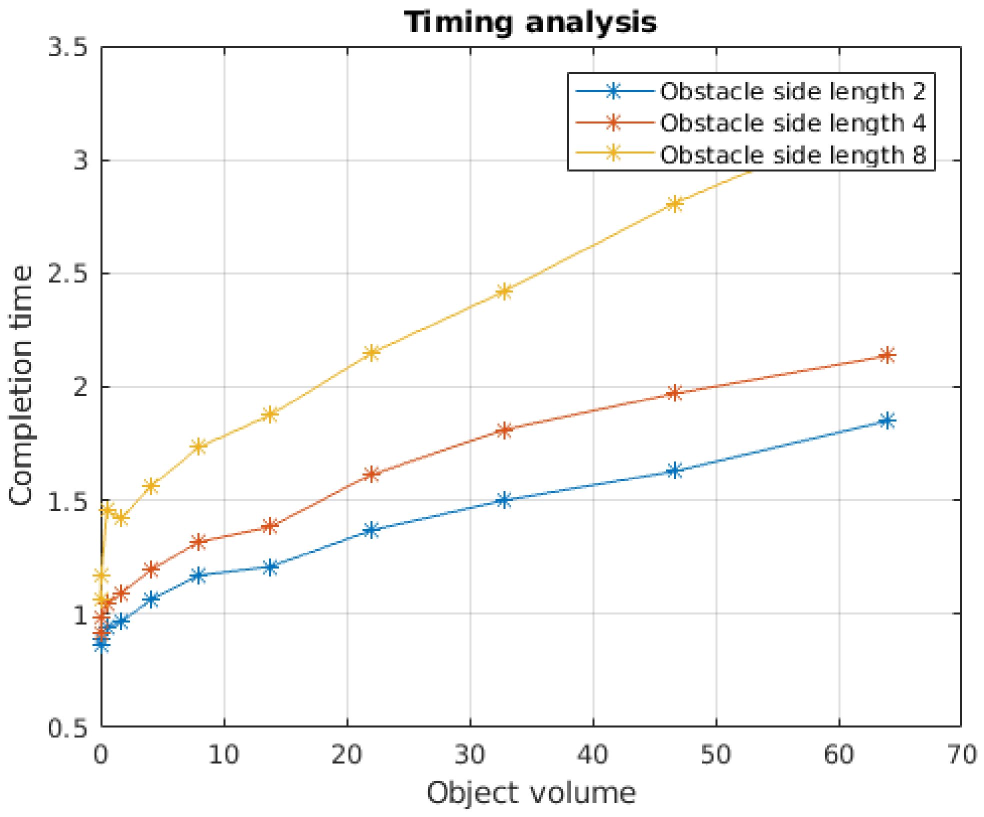


Figure 2.4: Impact of the dimension of the object on time performance

In figure 2.4 we can appreciate how the size of the grasped object affects the time expense for computing an optimal trajectory from the source to the goal position, it results to be linear with respect to the volume of the object and getting worse as the size of the obstacle increases.

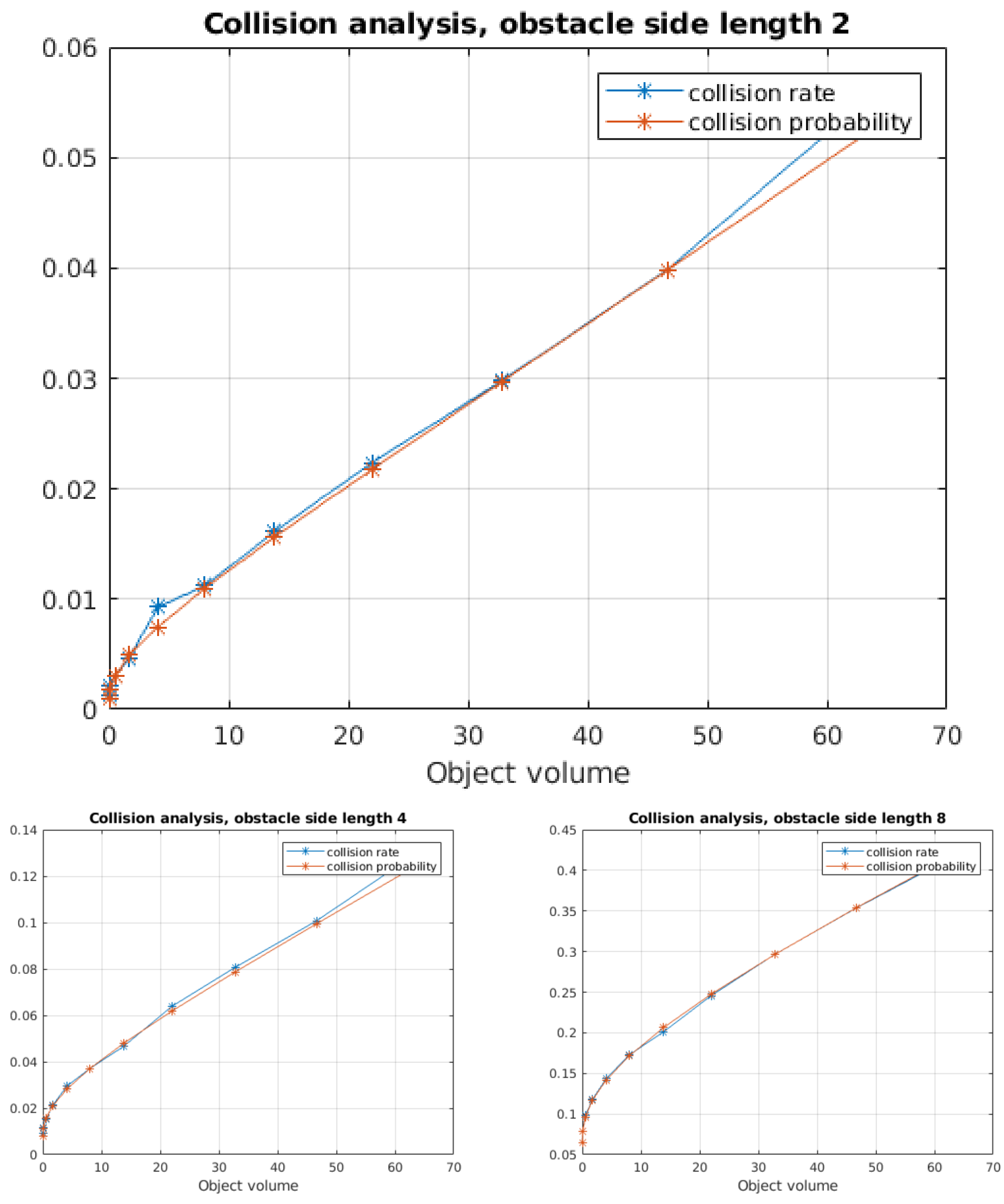
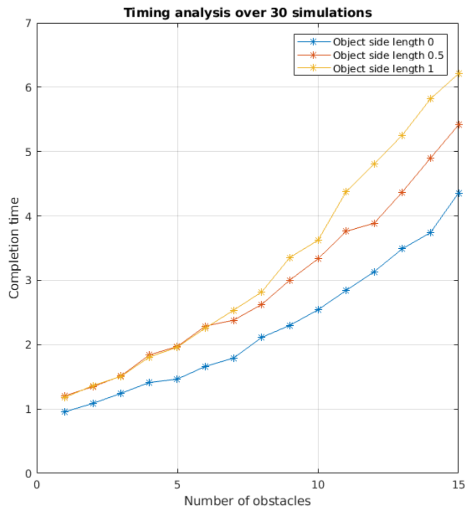


Figure 2.5: Relationships between collision rate and probability

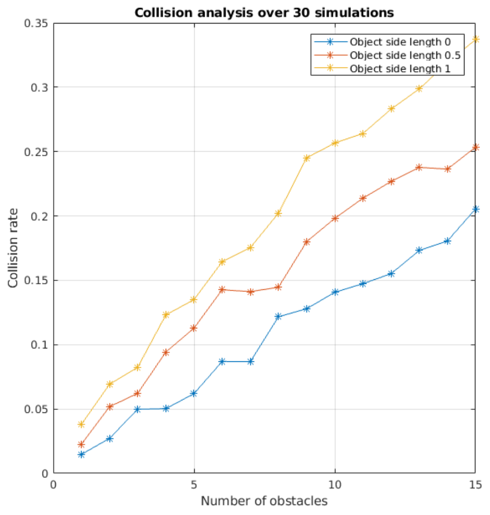
In figure 2.5 we want to put into evidence the tight relationship between theoretical collision probability and real collision rate, the first one is given by $(obstacleSize + objectSize)^3/(workspaceSize - objectSize)^3$ while the second one by $\frac{numberOfCollisions}{numberOfSamplings}$, the obstacle size plays the role of increasing the probability of collision as we compare the 3 graphs, note that with *size* we mean the length of one side of the cube.

Another important analysis reguards the number of obstacles in the workspace which linearly affects the computational time and the collision rate, in both the situation we have an object of size 0, 0.5, 1 in three different colors in order to focus only on the impact of obstacles in the workspace and contemporarily highlighting the impact of object size somehow reversing the analysis performed at figure 2.4.

In figure 2.6a an average value is provided as well as in 2.6b and in the latter case we can notice how the random sampling over a random workspace can affect the linear trend, much more than in the timing analysis.



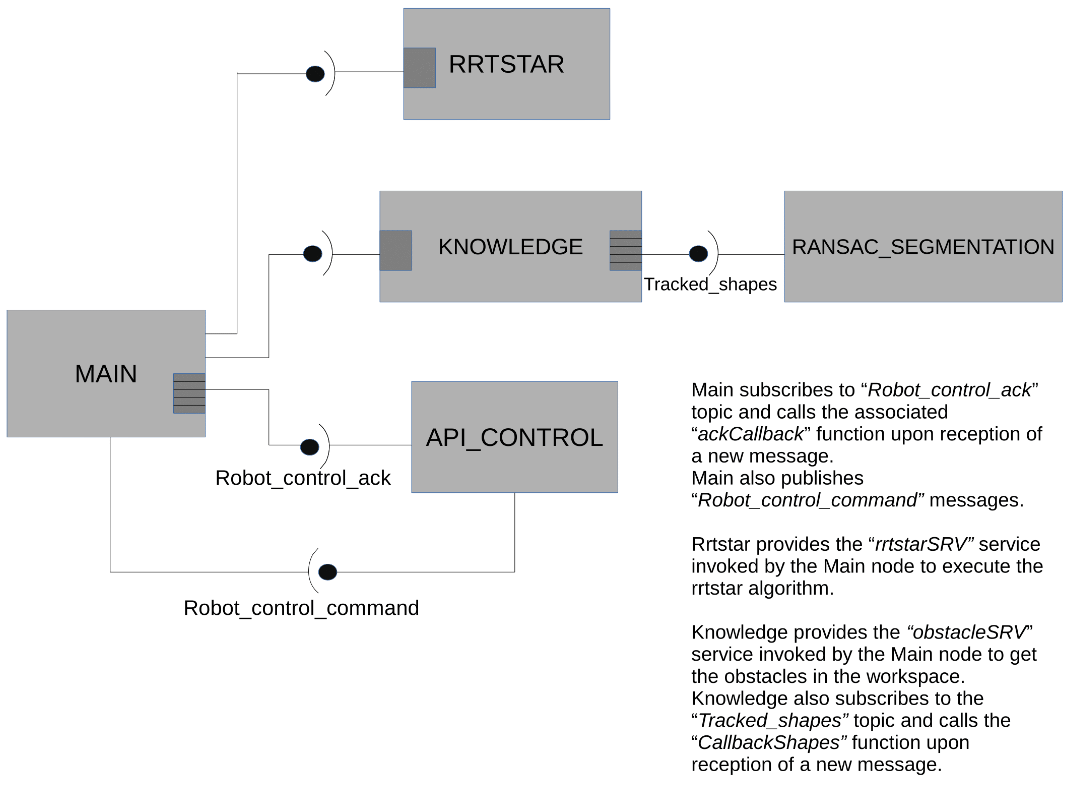
(a) Timing analysis



(b) Collision analysis

Figure 2.6: Timing and collision analysis with respect to number of obstacles

3 From theory to practise



Main subscribes to "*Robot_control_ack*" topic and calls the associated "*ackCallback*" function upon reception of a new message.
Main also publishes "*Robot_control_command*" messages.

Rrtstar provides the "*rrtstarSRV*" service invoked by the Main node to execute the rrtstar algorithm.

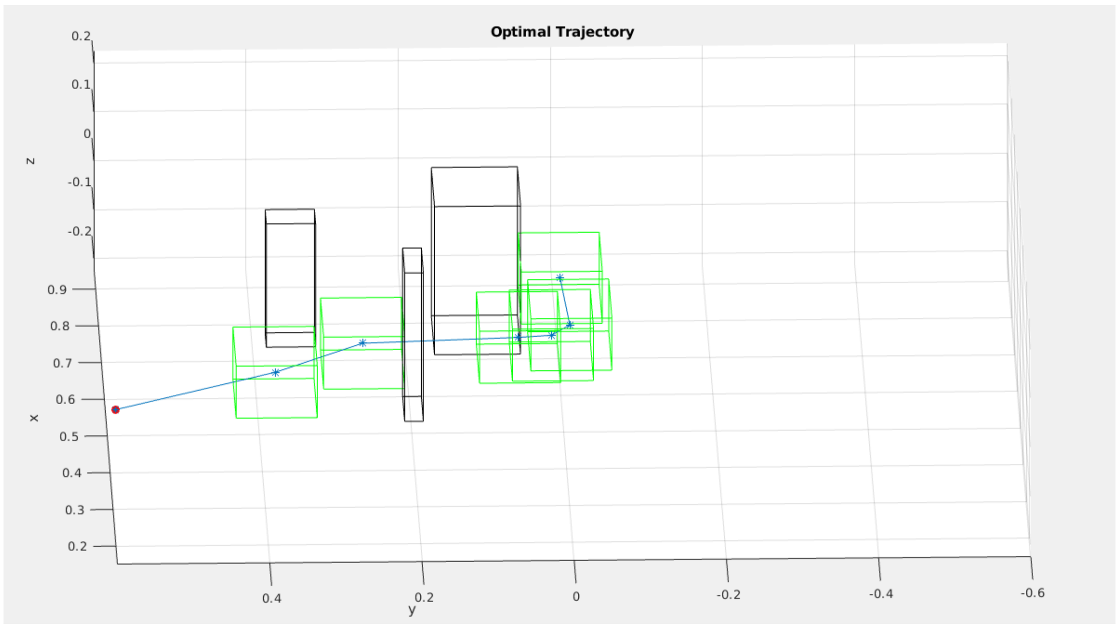
Knowledge provides the *"obstacleSRV*" service invoked by the Main node to get the obstacles in the workspace.
Knowledge also subscribes to the "*Tracked_shapes*" topic and calls the "*CallbackShapes*" function upon reception of a new message.

Figure 3.1: Our ros architecture

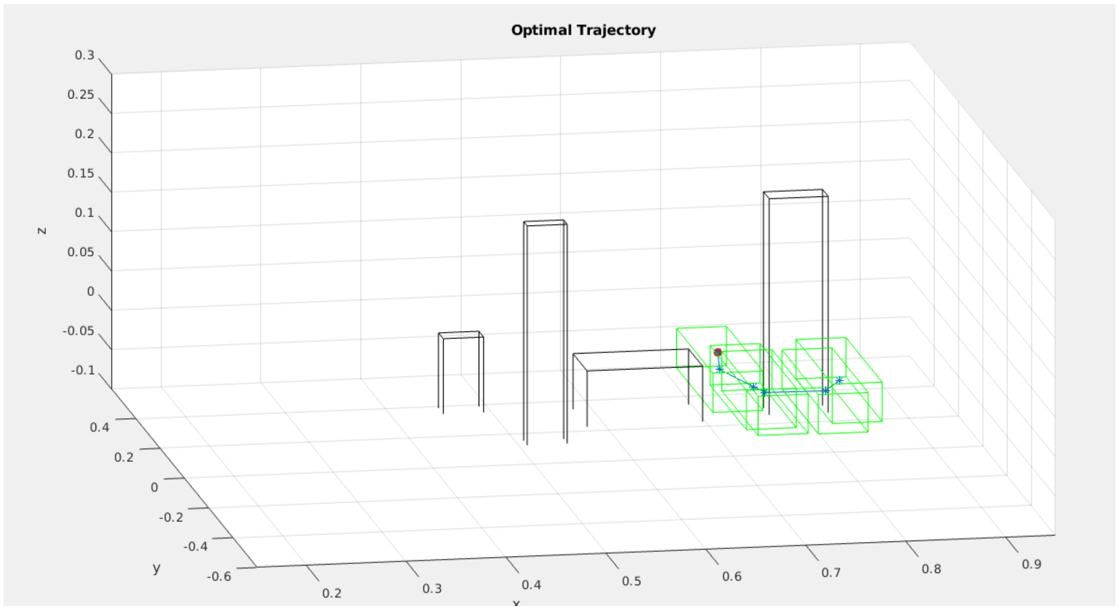
Although the results provided by the simulations have been interesting the standalone version is not enough to make the Baxter robot moving over the workspace performing collision avoidance, that's now that ROS comes into play.

We implemented the RRT* algorithm as a service, what we needed was to introduce a new node whose objective is to provide all the inputs necessary for the service extracting the optimal path trajectory as output to be sent to the control node responsible for moving the Baxter, in these last pages you can have a look to a matlab simulation inside some real workspaces and our software architecture.

In figure 3.1 you can notice the ransac segmentation node, which is responsible for publishing the tracked shapes through the kinect mounted on the Baxter which locates a point cloud, the knowledge node is responsible for the reconstruction of the objects in the workspace.



(a) Workspace with 3 cylinders



(b) Workspace with 3 cylinders and 1 sphere

Figure 3.2: The Baxter passing through the obstacles with different workspaces

In figure 3.2 we can figure out the trajectory of the baxter arm in various situations, note that the green bounding is larger then the real object according to a safety factor you can find in our code.

4 What's next

So far our obstacle avoidance algorithm is performed only on Baxter without changing the orientation, actually our work is conceived to be extended for other kind of robots with little modifications

In real world we cannot think not to exploit the degrees of freedom that a human or robotic arm provides, therefore even if the algorithm gets more complicated, the capability of changing orientation of the arm can lead to more cost-effective trajectories, not trivial things once our work is extended to quadricopters or mobile robots in which energy consumption matters.

5 References

Our work can be found in the github repository:
`https://github.com/AlbertBit/rrtstar_remus_maccio`

Videos of the Baxter in action can be found at:
`https://drive.google.com/drive/folders/1xuVRzpmkaUUe9EbzCr-s1qGLoGC1TvXX?usp=sharing`

Our reference material has been the paper Sampling-based algorithms for optimal motion planning - Sertac Karaman and Emilio Frazzoli

A special thank to our supervisor Kourosh Darvish.