Autonomous and Mobile Robotics
Final Project
Academic Year 2021/22

Comparing classic and primitive-based versions of kinodynamic RRT*

Lorenzo Mattia 1793272
Michela Proietti 1739846
Sofia Santilli 1813509

Abstract

Kinodynamic motion planning combines the search for a collision-free path with the underlying dynamics of the considered system. One of the best techniques for motion planning is RRT* (Rapidly-exploring Random Tree), because of its probabilistic completeness and asymptotic optimality properties. However, in its kinodynamic form, RRT* potentially needs multiple calls to the steering function at each iteration, which inevitably increases its computational cost. A recently introduced version of RRT* tries to overcome such computational challenge by exploiting a primitive-based approach. In particular, a catalogue of motion primitives is precomputed, by discretizing the configuration space and storing all the paths between each pair of discrete configurations before constructing the tree. In this way, the search for the optimal path just consists in finding an appropriate concatenation of such primitives. This report explains the most important implementation details and highlights the main similarities and differences between the two versions of RRT*. Simulations are performed in CoppeliaSim using a unicycle, here treated as a Dubins' vehicle.

1 Introduction

Motion planning has several applications in the robotics domain, as well as in other fields, such as computer-aided surgery, manufacturing, video games and the study of biological molecules.
The motion planning problem consists in searching for a collision-free path between an initial state and a final one. Additionally, kinodynamic planning takes into account the underlying dynamics of the system in order to guarantee feasibility of the resulting trajectories. This has a great relevance in the case of differentially constrained systems, for which motion planning may return collision-free paths that cannot be easily turned into feasible trajectories. If we need to satisfy additional requirements and we need to look for an optimal trajectory, we have to solve an optimal kinodynamic motion planning problem [2]. Kinodynamic planning can be addressed using exact or sampling-based methods.

1.1 Exact methods

Exact methods look for a solution in the continuous state space, and they are complete, because they terminate in a finite time returning a solution, if one exists, or a failure otherwise. These approaches require an explicit representation of obstacles and therefore even simpler versions of the motion planning problem are PSPACE hard. Despite this, exact methods are able to return a practical solution for problems that are characterized by a low dimensional state space.

1.2 Sampling-based methods

Sampling-based methods are based on the idea of building a roadmap in the form of a graph or a tree, sampling the continuous state space and connecting these states (nodes) with trajectories in the collision-free space. These methods are probabilistically complete, which means that the probability of finding a solution, if one exists, converges to one as the number of samples grows to infinity. Sampling-based algorithms have the advantage that they are relatively quick even in high-dimensional state spaces. Moreover, the use of a collision check module to determine the feasibility of a tentative trajectory eliminates the need for an explicit representation of obstacles. However, in all sampling-based methods, a steering function is required to design a trajectory connecting two nodes and this usually represents a computational

bottleneck.
Two popular sampling-based methods are *Probabilistic Road Maps* (PRM) and *Rapidly exploring Random Trees* (RRT). PRM [3] randomly samples nodes in the free configuration space and builds a graph connecting them by means of a local planner. Moreover, it is multiple-query, because given any two nodes we can directly look for a path between them in the graph. RRT [4], instead, incrementally builds a tree starting from a given node and returns a solution as soon as a node in the goal region is added to the tree.
Two variants of these algorithms have been then proposed, respectively PRM* and RRT*, in order to achieve asymptotic optimality, which means that the probability of finding an optimal solution, if there exists one, converges to one as the tree cardinality grows to infinity. Such property is obtained by rewiring the graph at each insertion of a new node *n*: connections with pre-existing nodes that are in a suitably defined neighborhood are tested and if a node *n'* is found such that it is closer to *n*, the edges in the graph are updated.
In optimal kinodynamic planning, there is the additional difficulty of solving a two point boundary value problem (TPBVP) to implement the steering function. In this case, PRM* is limited to symmetric costs and to systems for which the cost associated with TPBVP is conserved when the boundary pairs are swapped; therefore it isn't suitable for nonholonomic systems. However, for this type of systems, even in the case of RRT*, the presence of kinodynamic constraints makes the constrained optimization problem, that the steering function needs to solve, very hard.
In this paper, we will focus our attention on RRT* and we compare its standard version with the variant with motion primitives, showing the advantages and drawbacks of both algorithms. In our experiments, we have treated a unicycle as a Dubins' vehicle, that is a nonholonomic system for which it has been shown that the shortest path connecting two configurations must belong to a set of six possible paths. In fact, solving a TPBVP is a very complex task and the Dubins' vehicle allowed us to show the features of the two algorithms without having to face such difficulties. In the end, we present the implementation using the TPBVP as an extension to the case of a generic unicycle.

2 Problem Statement

2.1 Kinematic model of a unicycle

In both implementations of RRT* we considered a unicycle, which can be defined as a mathematical abstraction based on the *pure rolling constraint*. Such constraint can be written as:

$$\dot{x}\sin\vartheta - \dot{y}\cos\vartheta = \begin{bmatrix}\sin\vartheta & -\cos\vartheta & 0\end{bmatrix}\dot{\mathbf{q}} = 0 \tag{1}$$

Writing the unicycle's configuration as

$$\mathbf{q} = \begin{bmatrix} x \\ y \\ \vartheta \end{bmatrix} \tag{2}$$

where (x, y) is the contact point of the wheel with the ground and θ is the orientation of the wheel with respect to the x axis, it is clear that the pure rolling constraint implies that the velocity of the contact point must be zero along the orthogonal direction to the sagittal axis of the unicycle.

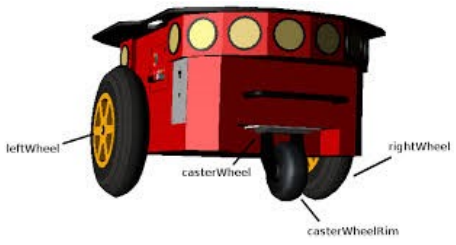

Figure 1: CoppeliaSim model of the Pioneer p3dx

The kinematic model of a unicycle can be written as:

$$\begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{\vartheta}\end{bmatrix} = \begin{bmatrix}\cos\vartheta\\ \sin\vartheta\\ 0\end{bmatrix} v + \begin{bmatrix}0\\0\\1\end{bmatrix}\omega \tag{3}$$

where v is the driving velocity and ω is the steering velocity. In all our simulations we have used the *Pioneer p3dx* model in CoppeliaSim, which can indeed be modelled as a unicycle.

2.2 Motion planning problem

Given a unicycle with the dynamic model described in the previous section and being $Q \in \mathbb{R}^d$ and $U \in \mathbb{R}^m$ respectively the state space and the input space, we denote the obstacle region as Q_{obs} and the obstacle-free space as $Q_{free} = Q \backslash Q_{obs}$. Finally, we denote by $Q_{goal} \subset Q$ the goal region. The motion planning problem consists in finding a control input $\mathbf{u} : [0, T] \rightarrow U$ that yields a feasible path $\mathbf{q}(t) \in Q_{free}$ for $t \in [0, T]$ from an initial state $\mathbf{q(0)} = \mathbf{q}_{ini}$ to the goal region $\mathbf{q}(T) \in Q_{goal}$ that obeys the system dynamics. Since we want to solve an optimal motion planning problem, we also have to consider a cost function that the resulting path needs to minimize.

2.3 Dubins' Vehicles

In our experiments, we have treated our unicycle as a Dubins' vehicle [1], that is a nonholonomic system that is only allowed to turn left with maximum angular speed (L), turn right with maximum angular speed (R), or go straight (S). It has been demonstrated that, for a Dubins' vehicle, the shortest path linking two configurations belongs to a family of six canonical paths. These are particular combinations of the previously exposed controls, namely: RSR, LSL, RSL, LSR, RLR and LRL. Writing the system's configuration as 2, the dynamics of the Dubins' vehicle is described by the following nonlinear differential equation:

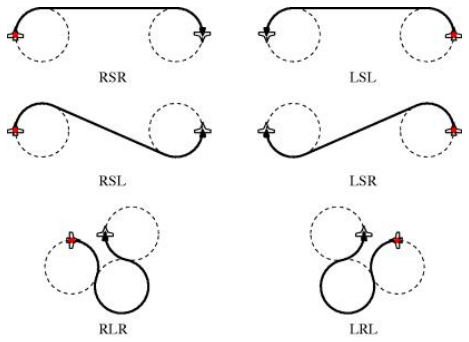

Figure 2: The six trajectories of the Dubins' vehicles

$$\begin{aligned} \dot{x} &= v \cos\theta \\ \dot{y} &= v \sin\theta \\ \dot{\theta} &= u, |u| = \frac{v}{\rho} \end{aligned} \tag{4}$$

where x and y are the Cartesian coordinates of the vehicle, θ is the heading angle, u is the steering input, v is the speed and ρ is the minimum turning radius. The speed v and the minimum turning radius ρ are assumed to be constant.

3 Methods

In this section, we will explain in details the two studied versions of RRT*, highlighting the main differences between them.

3.1 RRT*

In order to explain how the algorithm works, we will show the pseudocode and explain what each function does. By $\mathcal{T} = (V, E)$ we denote the tree built by RRT*, with V being the set of vertices belonging to Q_{free} and $E \subseteq V \times V$ being the set of connected edges between the elements of V.

- **SampleRand**: it randomly samples a configuration from Q_{free} according to a uniform distribution.
- **NearestNeighbours**: given the newly randomly sampled node, it computes the nearest node in the tree according to the distance function.
- **Steer**: given two configurations $\mathbf{q_1}$ and $\mathbf{q_2}$, finds the path $\mathbf{x} : [0, T] \leftarrow Q$, such that $\mathbf{q}(0) = \mathbf{q_1}$ and $\mathbf{q}(T) = \mathbf{q_2}$ and returns $\mathbf{x}$, the control inputs $\mathbf{u}$ needed to generate $\mathbf{x}$, and the duration T of the path. In particular, in the case of Dubins' vehicle, this function first computes all the 6 trajectories joining $\mathbf{q_1}$ to $\mathbf{q_2}$, and then selects the one with minimum cost.
- **NearByVertices**: given the node $\mathbf{q_{new}}$ that is going to be added to $\mathcal{T}$ and the cardinality n of $\mathcal{T}$, this function returns the nodes in the tree that are the nearest to $\mathbf{q_{new}}$. In particular, in the case of non-Euclidean cost metrics, being $C(e_{\mathbf{q_{rand}},\mathbf{q}})$ the cost for going from $\mathbf{q_{rand}}$ to $\mathbf{q}$, this function returns all the nodes which belong to $Q_{reach} = \{\mathbf{q} \in Q : C(e_{\mathbf{q_{rand}},\mathbf{q}}) \leq l(n) \ \vee \ C(e_{\mathbf{q},\mathbf{q_{rand}}}) \leq l(n)\}$, that are the nodes $\mathbf{q}$ than can reach and can be reached from $\mathbf{q_{rand}}$ with a cost that is lower than $l(n)$, which is a threshold that decreases over iterations as $l(n) = \gamma_l((\log n)/n)$.

- **ChooseParent**: given the newly sampled node $\mathbf{q}_{\text{new}}$ and the set of near nodes computed by *NearByVertices*, it returns the parent of $\mathbf{q}_{\text{new}}$. Rather than simply choosing the nearest node to $\mathbf{q}_{\text{new}}$, it evaluates the cost of picking each node in the set as its parent and chooses the one with the lowest cost from the root.
- **AddNewNode**: given the tree $\mathcal{T} = (V, E)$, a node $\mathbf{q}_{\text{tree}} \in V$ and a new node $\mathbf{q}_{\text{new}}$, it adds $\mathbf{q}_{\text{new}}$ to V and adds the edge between the two nodes in E.
- **Rewire**: checks each node $\mathbf{q}_{\text{near}}$ belonging to the set computed by *NearByVertices*, in order to see whether reaching $\mathbf{q}_{\text{near}}$ via $\mathbf{q}_{\text{new}}$ would achieve lower cost than via its current parent. When this connection reduces the total cost associated to $\mathbf{q}_{\text{near}}$, the algorithm assigns $\mathbf{q}_{\text{new}}$ as parent of $\mathbf{q}_{\text{near}}$.

Algorithm 1: RRT*

Data: $\mathbf{q}_{\text{ini}}, \mathbf{q}_{\text{fin}}$
Result: $\mathcal{T}$
$\mathcal{T} \leftarrow InitializeTree()$;
$\mathcal{T} \leftarrow AddNewNode(\emptyset, \mathbf{q}_{\text{ini}}, \mathcal{T})$;
for *i = 1 to i = N* **do**
  $\mathbf{q}_{\text{rand}} \leftarrow SampleRand()$;
  $\mathbf{q}_{\text{nearest}} \leftarrow NearestNeighbours(\mathbf{q}_{\text{rand}}, \mathcal{T})$;
  $(\mathbf{x}_{\text{new}}, \mathbf{u}_{\text{new}}, T_{new}) \leftarrow Steer(\mathbf{q}_{\text{near}}, \mathbf{q}_{\text{rand}})$;
  $\mathbf{q}_{\text{new}} = \mathbf{q}_{\text{rand}}$;
  if $checkCollision(\boldsymbol{x}')$ **then**
    $Q_{near} \leftarrow NearByVertices(\mathbf{q}_{\text{near}}, \mathcal{T})$;
    $\mathbf{q}_{\text{min}} \leftarrow$
     $ChooseParent(Q_{near}, \mathbf{q}_{\text{nearest}}, \mathbf{q}_{\text{rand}}, \mathbf{x}')$;
    $\mathcal{T} \leftarrow AddNewNode(\mathbf{q}_{\text{min}}, \mathbf{q}_{\text{new}}, \mathcal{T})$;
    $\mathcal{T} \leftarrow Rewire(\mathcal{T}, Q_{near}, \mathbf{q}_{\text{min}}, \mathbf{q}_{\text{new}})$;
  end
end

Algorithm 2: ChooseParent

Data: $Q_{near}, \mathbf{q}_{\text{nearest}}, \mathbf{q}_{\text{new}}, \mathbf{x}_{\text{new}}$
Result: $\mathbf{q}_{\text{min}}$
$\mathbf{q}_{\text{min}} \leftarrow \mathbf{q}_{\text{nearest}}$;
$c_{min} \leftarrow Cost(\mathbf{q}_{\text{nearest}}) + c(\mathbf{x}_{\text{new}})$;
for $\boldsymbol{q}_{near} \in Q_{near}$ **do**
  $(\mathbf{x}', \mathbf{u}', T') \leftarrow Steer(\mathbf{q}_{\text{near}}, \mathbf{q}_{\text{new}})$;
  if $ObstacleFree(\boldsymbol{x}')$ *and* $\boldsymbol{x}'(T') = \boldsymbol{q}_{new}$ **then**
    $c' = Cost(\mathbf{q}_{\text{near}}) + c(x')$;
    if $c' < Cost(\boldsymbol{q}_{new})$ *and* $c' < c_{min}$ **then**
      $\mathbf{q}_{\text{min}} \leftarrow \mathbf{q}_{\text{near}}$;
      $c_{min} \leftarrow c'$;
    end
  end
end

Algorithm 3: Rewire

Data: $\mathcal{T}, Q_{near}, \mathbf{q}_{\text{min}}, \mathbf{q}_{\text{new}}$
Result: $\mathcal{T}$
for $\boldsymbol{q}_{near} \in Q_{near} \backslash \{\boldsymbol{q}_{min}\}$ **do**
  $(\mathbf{x}', \mathbf{u}', T') \leftarrow Steer(\mathbf{q}_{\text{new}}, \mathbf{q}_{\text{near}})$;
  if $ObstacleFree(\boldsymbol{x}')$ *and* $\boldsymbol{x}'(T') = \boldsymbol{q}_{near}$ *and*
   $Cost(\boldsymbol{q}_{new}) + c(\boldsymbol{x}') < Cost(Q_{near})$ **then**
    $\mathcal{T} \leftarrow ReConnect(\mathbf{q}_{\text{new}}, \mathbf{q}_{\text{near}}, \mathcal{T})$;
  end
end

3.2 RRT* with motion primitives

In RRT* multiple calls to the steering function are necessary at each iteration in order to compute the minimum cost paths. However, this is computationally challenging, especially when dealing with complex dynamics, such as for nonholonomic vehicles because of the presence of kinodynamic constraints. For this reason, an extension of RRT* has been proposed, in which a database of pre-computed motion primitives is used in order to avoid performing these computations online. This version of RRT* is based on a uniform discretization of the state space and on the computation of a finite set of motion primitives with boundary conditions on the points of the gridded space. As it has been previously done for RRT*, we will describe the algorithm by looking at the pseudocode and explaining what each function does.

- **SampleRand**: As in RRT*, this function randomly samples a configuration from Q_{free} according to a uniform distribution, but in this case the state space is discretized, so the points are taken on a grid and the orientations belong to a finite set of orientations.
- **NearByVertices**: works exactly as in RRT*.
- **FindTrajectory**: given a pair of initial and final states, it applies a translation (and if needed a rotation) so that the initial state corresponds to the null vector. Then, it looks for a trajectory connecting the two translated states in the database of motion primitives, retrieving its cost and the sequence of nodes it is composed by. Finally, the inverse of

previous translation (ad rotation) is applied to all the nodes in the trajectory so as to get the actual edge connecting the initial and final states.

- **Extend**: given the newly sampled node $\mathbf{q_{rand}}$ and the set of near nodes computed by *NearByVertices*, for each $\mathbf{q_{near}}$ it calls the function *FindTrajectory* in order to obtain the pre-computed primitive connecting it to $\mathbf{q_{rand}}$ and returns as parent of $\mathbf{q_{rand}}$ the node corresponding to the trajectory with the lowest cost.
- **Rewire**: works exactly as in RRT*, but the trajectories connecting each $\mathbf{q_{near}}$ with $\mathbf{q_{rand}}$ are simply retrieved from the database of motion primitives through the *FindTrajectory* function.

Algorithm 4: RRT*_MotionPrimitives

Data: $\mathbf{q_{ini}}, \mathbf{q_{fin}}$
Result: $\mathcal{T}$
$\mathcal{T} \leftarrow InitializeTree()$;
$\mathcal{T} \leftarrow AddNewNode(\emptyset, \mathbf{q_{ini}}, \mathcal{T})$;
for *i = 1 to i = N* **do**
  $\mathbf{q_{rand}} \leftarrow SampleRand()$;
  $Q_{near} \leftarrow NearByVertices(\mathbf{q_{rand}}, \mathcal{T})$;
  $\mathbf{q_{best}} \leftarrow Extend(Q_{near}, \mathbf{q_{rand}})$;
  if $\boldsymbol{q_{rand}} \notin V \wedge \boldsymbol{q_{best}} \neq \emptyset$ **then**
    $V \leftarrow V \cup \mathbf{q_{rand}}$;
    $E \leftarrow E \cup (\mathbf{q_{best}}, \mathbf{q_{rand}})$;
    $\mathcal{T} \leftarrow Rewire(\mathcal{T}, \mathbf{q_{rand}}, Q_{near})$;
  else
    $\mathbf{q_{prev}} \leftarrow Parent(\mathbf{q_{rand}})$;
    if $\boldsymbol{q_{best}} \neq \boldsymbol{q_{prev}}$ **then**
      $E \leftarrow (E \backslash (\mathbf{q_{prev}}, \mathbf{q_{rand}})) \cup \mathbf{q_{best}}, \mathbf{q_{rand}}$;
      $\mathcal{T} \leftarrow Rewire(\mathcal{T}, \mathbf{q_{rand}}, Q_{near})$;
    end
  end
end

Algorithm 5: Extend

Data: $Q_{near}, \mathbf{q_{rand}}$
Result: $\mathbf{q_{best}}$
$\mathbf{q_{best}} \leftarrow \emptyset$;
$e_{best} \leftarrow \emptyset$;
$c_{best} \leftarrow \infty$;
for $\boldsymbol{q_{near}} \in Q_{near}$ **do**
  $(\mathbf{x'}, \mathbf{u'}, T') \leftarrow$
   $FindTrajectory(\mathbf{q_{near}}, \mathbf{q_{rand}})$;
  if $c(\boldsymbol{x'}) < c_{best}$ **then**
    if $ObstacleFree(\boldsymbol{x'})$ **then**
      $\mathbf{q_{best}} \leftarrow \mathbf{q_{near}}$;
      $e_{best} \leftarrow \mathbf{x'}$;
      $c_{best} \leftarrow c(\mathbf{x'})$;
    end
  end
end

Algorithm 6: Rewire

Data: $\mathcal{T}, Q_{near}, \mathbf{q_{rand}}$
Result: $\mathcal{T}$
for $\boldsymbol{q_{near}} \in Q_{near}$ **do**
  $(\mathbf{x'}, \mathbf{u'}, T') \leftarrow$
   $FindTrajectory(\mathbf{q_{rand}}, \mathbf{q_{near}})$;
  if $Cost(\boldsymbol{q_{rand}}) + c(x') < Cost(\boldsymbol{q_{near}})$ **then**
    if $ObstacleFree(\boldsymbol{x'})$ **then**
      $\mathcal{T} \leftarrow ReConnect(\mathbf{q_{rand}}, \mathbf{q_{near}}, \mathcal{T})$;
    end
  end
end

4 Implementation Details

The two versions of RRT* have been developed in an Ubuntu 18.04 environment using C++ and performing simulations in CoppeliaSim 4.0 EDU. In the following sections, we explain how the most relevant functions and data structures are implemented.

4.0.1 Tree structure

In both versions of RRT* the tree, despite being built in different ways, shares the same implementation. It consists of a vector of structures named $VERTEX$. Each vertex structure is made up of:

- an *Eigen::VectorXd* **z**, that represents a node of the tree structured as (x, y, θ, idx, idx_{parent}, $cost$), where (x, y, θ) are respectively the Cartesian coordinates of the contact point and the orientation of the wheel, idx is the index of the node in the vector of $VERTEX$ structures representing the tree, idx_{parent} is the index of its parent and $cost$ is the cost needed to reach the node from the root.

- a *std::vector<Eigen::VectorXd>* **parent_to_node**, that is the sequence of nodes constituting the path that reaches the considered node starting from its parent.

4.0.2 Cost Function

Given a path between two states, described as a sequence of configurations, we compute its cost as the sum of the distances between each pair of configurations. The distance between two configurations $\mathbf{q_1}$ and $\mathbf{q_2}$ takes into account also the difference in orientation and it is computed as:

$$d(\mathbf{q}_1, \mathbf{q}_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} + w|\theta_1 - \theta_2| \tag{5}$$

where the weight w has been set to 0.2 during the experiments, so that the difference in orientation between the two configurations does not overly increase the cost.

4.0.3 *NearByVertices* function

The threshold $l(n)$ proposed in the original paper becomes lower as the tree grows (i.e. as n increases) and therefore the number of nodes that are added to the tree at each iteration decreases. For this reason, a large number of iterations are needed in order to find a solution, given that there is one. To address this problem, in our implementation we have considered a simplified version of the *NearByVertices* function, that returns the set containing the k nearest nodes to $\mathbf{q_{new}}$ identified using the distance function described in 5, where we considered k=5.

4.0.4 Exploration-exploitation

In order to increase the algorithms' performances, especially in terms of the time needed to find the goal, we have implemented the *exploration vs exploitation* strategy. Therefore, at each iteration, the random node is extracted inside the goal region with a chosen probability r, which has been set to r = 0.2 during the experiments.

4.0.5 Primitives' dictionary

In RRT* with motion primitives, a key point is the creation of the primitives' dictionary. In our implementation, it consists of a text file in which each row is in the format $(final\_node; cost; path;)$, to simplify the retrieval of the needed trajectory. When it is not possible to find a path to a node, the final node is still written in the file, without of course reporting the path and its cost. In this way, as soon as the empty primitive is found the algorithm will skip to the next iteration, thus avoiding scrolling through the whole file.

In order to keep the size of the dataset as small as possible, we have decided to save only primitives that start in the origin $(0, 0)$ with orientation $0°$ and whose end points lie either in the first and third quadrants. All other cases can be obtained by translating and rotating the trajectories stored in the dictionary. In order to explain how we look for a certain primitive inside the dictionary, let us consider two configurations $\mathbf{q_A} = (x, y, \theta)$ and $\mathbf{q_B} = (x', y', \theta')$. First of all, both the configurations are translated by $-x$ and $-y$ in order to place $\mathbf{q_A}$, which is the starting point, in the origin. Then, they are both rotated of $-\theta$ to set the orientation of $\mathbf{q_A}$ to $0°$. At this point, we have obtained a new final configuration $\mathbf{q_{B\_rt}} = (x' - x, y' - y, \theta' - \theta)$, which is $\mathbf{q_B}$ translated and rotated. If $\mathbf{q_{B\_rt}}$ lays in the second or fourth quadrant, its y and θ coordinates are inverted by sign, thus obtaining a new configuration, $\mathbf{q_{B\_fin}}$, that lies in the first or third quadrant and for which we have a pre-computed primitive in the dictionary that we are going to look for. Once the desired primitive has been found, all the configurations it contains are rotated and translated back to their initial pose. This procedure is very important and it is exploited both in the *Extend* and in the *Rewire* functions.

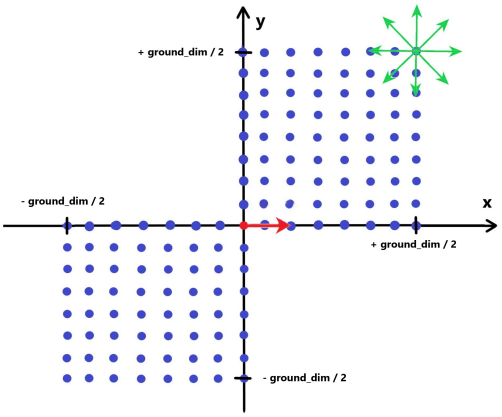

Figure 3: Configurations considered for computing the primitives in a scene of size $ground\_dim$ x $ground\_dim$. The red point is the configuration (0, 0, 0), starting from which all primitives are computed. For each blue point, we consider eight different orientations between $[-\pi, \pi)$, every $\pi/4$. Different experiments have been done also considering a reduced number of orientations.

4.0.6 Termination condition

When a configuration $\mathbf{q_{rand}}$ is sampled very close to the goal, with a certain tolerance, it is saved as a temporary goal and then the algorithm continues. Note that with the exploration/exploitation mechanism we are able to reach exactly the desired configuration most of the times. Successively, if another point in the goal region is extracted and the path to reach it is cheaper, the goal node gets updated. The algorithm terminates when the maximum number of iterations is reached.

5 Planning Experiments

The experiments have been performed using four increasingly challenging simulation environments, that are static and surrounded by walls that prevent the robot from exiting the floor while going from one state to another. Concerning RRT* with motion primitives, we have considered three dictionaries by changing the discrete grid resolution and the number of possible orientations, in order to see how the number of primitives affects the performance of the algorithm.
In the following sections, we show the environments' setups and we describe and compare the results obtained with the two different versions of the RRT* algorithm. The tables containing the statistics for each type of simulation that are present in the next sections have been computed by averaging the results obtained from 10 different trials. In the column "n.primitives", the number in brackets represents the maximum possible number of primitives, due to the chosen type of gridding. However, some of these primitives may not be feasible, therefore we report the number of the feasible ones, that will be used by the algorithm to connect the configurations in the tree. For the experiments on each scene, we have also inserted consecutive snapshots from which the movement of the robot along the computed path can be observed.

5.1 Scene 1

The first scene is the simplest one. It consists of a 5x5 [m] floor on which we have placed four obstacles of two different types. The robot starts from the configuration $(2.0, 1.5, \pi)$ and needs to reach the goal region, which is centered in $(-1.5, -1.5)$, with desired orientation $-\pi/2$.
The three used grids are:

- 1x1 grid, with 4 possible orientations for each intersection point, resulting in a dictionary of 63 feasible motion primitives;
- 1x1 grid, with 8 possible orientations for each intersection point, resulting in a dictionary of 126 feasible motion primitives;
- 0.5x0.5 grid, with 8 possible orientations for each intersection point, resulting in a dictionary of 370 feasible motion primitives.

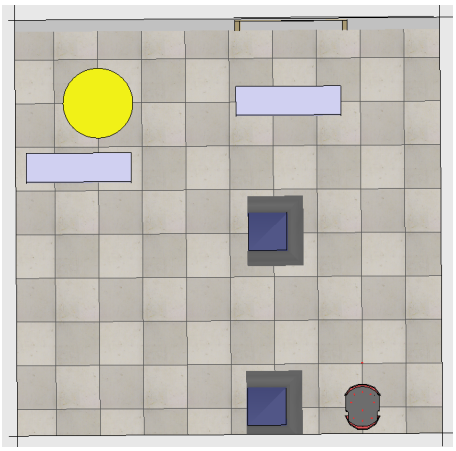

Figure 4: *Scene 1* - CoppeliaSim

Table 1 shows the results obtained on Scene 1 with the two different versions of RRT* and the different dictionaries of primitives.
As the number of iterations increases, the tree size noticeably grows. In the case of RRT* with motion primitives, also the success rate, expressing the percentage of times the algorithm finds a feasible path to reach the goal, improves accordingly. On the contrary, the standard version of the algorithm, which has an infinite number of configurations that can be sampled at each iteration, is always able to find a solution, even with only 100 iterations. Also the primitive-based algorithm is able to always reach the goal when running for a lot of iterations, namely 10.000. However, while using the smallest or the largest dictionaries, even with only 100 iterations it failed just 20% of the times. Surprisingly, using the intermediate dictionary the results are worse than with the smallest one. This is probably due to the fact that keeping the same grid resolution and increasing the number of possible orientations, since the size of the floor is quite small, it happens more often that we sample very close configurations with inconvenient orientations and this results in unfeasible paths.
Another important thing that must be noted is that, with the increase of the number of iterations, the path found by both algorithms gets progressively refined and therefore the cost of the solution decreases. In fact, thanks to the rewiring procedure, as more nodes are added to the tree, the paths for reaching nodes that were already present are eventually updated with less costly ones. For this scene, the lowest costs have been obtained through the primitive-based RRT* exploiting the smallest dictionary. This could be due to the fact that the scene is quite small and the grid-points and their respective orientations are placed in a comfortable way with respect to the initial position of the robot and goal area. For

this reason, considering more orientations in the other dictionaries results in having more inconvenient configurations among which to choose, and therefore in paths with higher costs. Despite the interesting observations that can be done regarding solution costs and success rate, a very relevant aspect when comparing the two methods is their computational time. In fact, it must be highlighted that the execution time is much lower in the case of RRT* with motion primitives. This is not surprising if we consider that most of the time is spent computing trajectories with the steering function and that this is not needed in the presence of the dictionary of pre-computed primitives. Looking at the 100 and 1000 iterations in standard RRT*, we notice that more than 80% of the execution time is owed to the steering function, while in RRT* with motion primitives, it takes less that 1 second to look for the needed trajectories inside the dictionary.

SCENE 1	**n.primitives**	**n.iters**	**tot.time(s)**	**steering time(s)**	**tree size**	**sol.cost**	**success rate(%)**
Standard RRT*	—	100	8.5	7.5	53	11.1	100
		1.000	102.3	82.7	465.3	8.49	100
		10.000	860	587.9	5562.6	8.43	100
Primitive -based RRT*	63 (64)	100	3.9	< 1	22.5	8.61	80
		1.000	40.6	< 1	44.4	8.35	100
		10.000	179.4	< 1	46	8.16	100
	126 (128)	100	1.9	< 1	21.6	14.83	20
		1.000	24.5	< 1	73.1	12.59	80
		10.000	245.4	113.7	81.4	12.01	100
	370 (384)	100	3.1	< 1	31.1	11.37	80
		1.000	41.9	< 1	198.2	11.21	100
		10.000	472.2	238.2	301.9	11.12	100

Table 1: Results obtained by applying the two algorithms to *Scene 1*.

The figures below show the solutions found respectively with the standard and primitive-based versions of RRT*. As we underlined before, the lowest cost solutions have been obtained using the smallest dictionary in the primitive-based algorithm, and this is reflected in the smooth path, that after avoiding the armchair goes straight to the goal. On the contrary, the path obtained using standard RRT* goes a little too far towards the left and then turns and goes back to avoid the dresser. Therefore, the observations aroused from the numerical values are confirmed by the visual results.

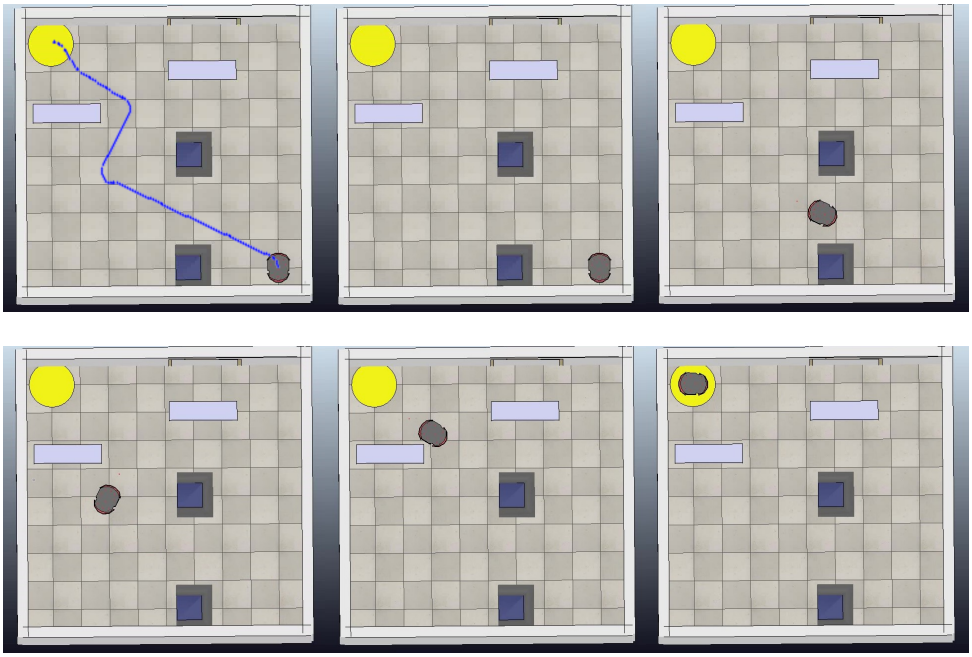

Figure 5: Optimal path found in Scene 1 with standard RRT* version. This solution has cost 8.19.

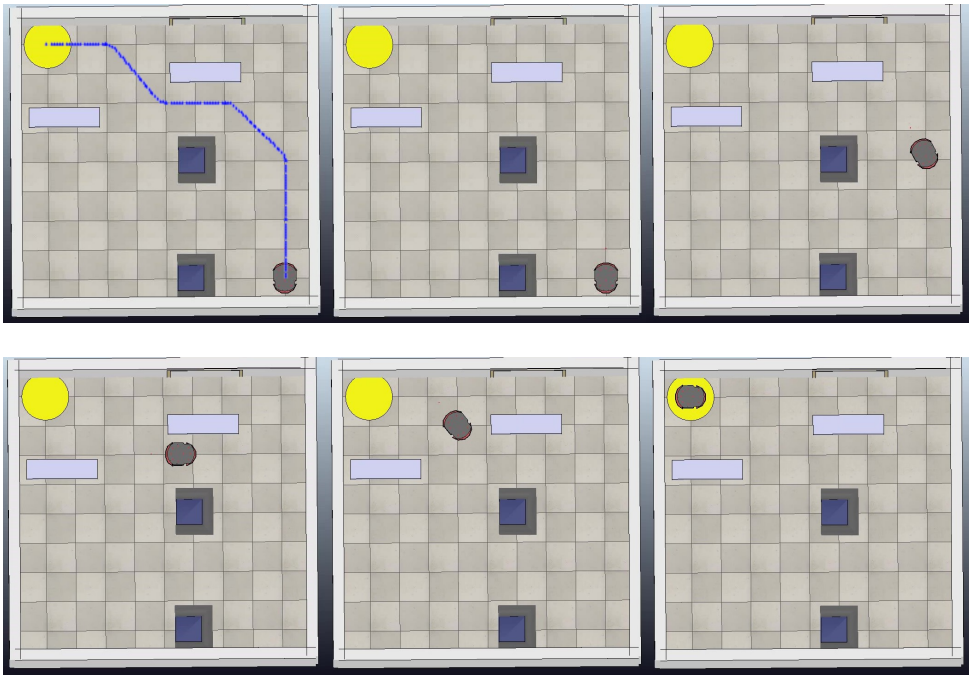

Figure 6: Optimal path found in Scene 1 with motion-primitive based RRT* version, using the dictionary with grid resolution 1.0 and 4 possible orientations. This solution has cost 7.52.

5.2 Scene 2

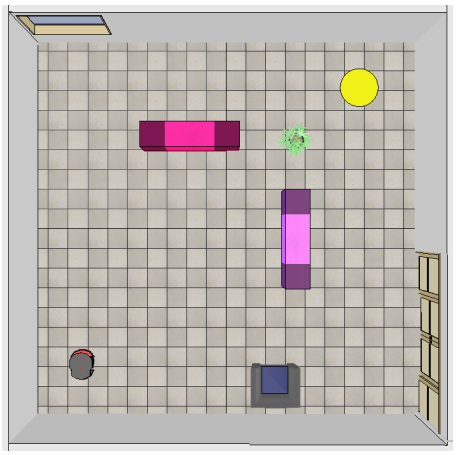

Figure 7: *Scene 2* - CoppeliaSim

The second scene is a 10x10 [m] floor with four obstacles of different dimensions (a plant, two dressers, a sofa) distributed in the space. The initial position of the robot is $(3.0, -4.0, \pi)$. The center of the goal region is $(-4.0, 3.0)$ and the desired orientation is π; The three used grids are:

- 4x4 grid, with 4 possible orientations for each intersection point, resulting in a dictionary of 14 feasible motion primitives;
- 1x1 grid, with 8 possible orientations for each intersection point, resulting in a dictionary of 325 feasible motion primitives
- 0.5x0.5 grid, with 8 possible; orientations for each intersection point, resulting in a dictionary of 1220 feasible motion primitives.

Table 2 shows the results obtained on scene 2 with the two different versions of RRT* and the different dictionaries of primitives. Differently from what we have seen for scene 1, in this case neither with standard RRT* we are able to achieve a success rate of 100% with 100 iterations, due to the larger dimension of the scene. Another difference with respect to the results obtained in scene 1 is that the lowest success rates in the case of the primitive-based algorithm are obtained with the smallest dictionary, and not with the intermediate one. This is probably due to the fact that since the scene is larger, having more orientations allows the robot to move more freely in the environment and to approach quite narrow passages, like the one that the robot passes through in Figure 8 and Figure 9, in such a way that it is also possible to traverse them. Similarly to what we have highlighted for scene 1, instead, with 10.000 iterations both algorithms succeed all the times. An additional difference between these results and those obtained for scene 1 concerns the costs of the solutions. In fact, while for scene 1 the lowest cost was obtained with the smallest dictionary in the primitive-based version, in this case it is given by standard RRT*, as expected. Moreover, the costs obtained with the largest dictionary are higher than those we get with the other two. This could be due to the fact that having a finer

resolution, we have much more points to sample and therefore there is a lower probability of picking the only two points between the plant and the dressers, that allow to go straight to the goal. For this reason, the robot tends to take more external paths that are longer and more expensive. Finally, the same observations we have done for scene 1 concerning the execution time still hold, because the primitive-based version has still lower results, due to the fact that it does not have to repeatedly compute the trajectories connecting the newly sampled nodes to the tree.

SCENE 2	n.primitives	n.iters	tot.time(s)	steering time(s)	tree size	sol.cost	success rate(%)
Standard RRT*	—	100	31.9	25.9	60.2	14.31	90
		1.000	253.5	211.8	655.7	14.43	100
		10.000	2998.2	2187.6	5727.9	14.18	100
Primitive -based RRT*	14 (24)	100	< 1	< 1	1.5	//	0
		1.000	3.7	< 1	8.9	19.52	50
		10.000	172.8	< 1	42	16.56	100
	325 (560)	100	4.3	< 1	34	18.27	70
		1.000	48.4	35.5	275.7	18.25	100
		10.000	518.1	270.5	495.2	17.79	100
	1220 (1920)	100	8.9	7	40.1	21.72	80
		1.000	104.7	70	429.8	20.06	100
		10.000	1045.5	620	1703.1	18.11	100

Table 2: Results obtained by applying the two algorithms to *Scene 2*.

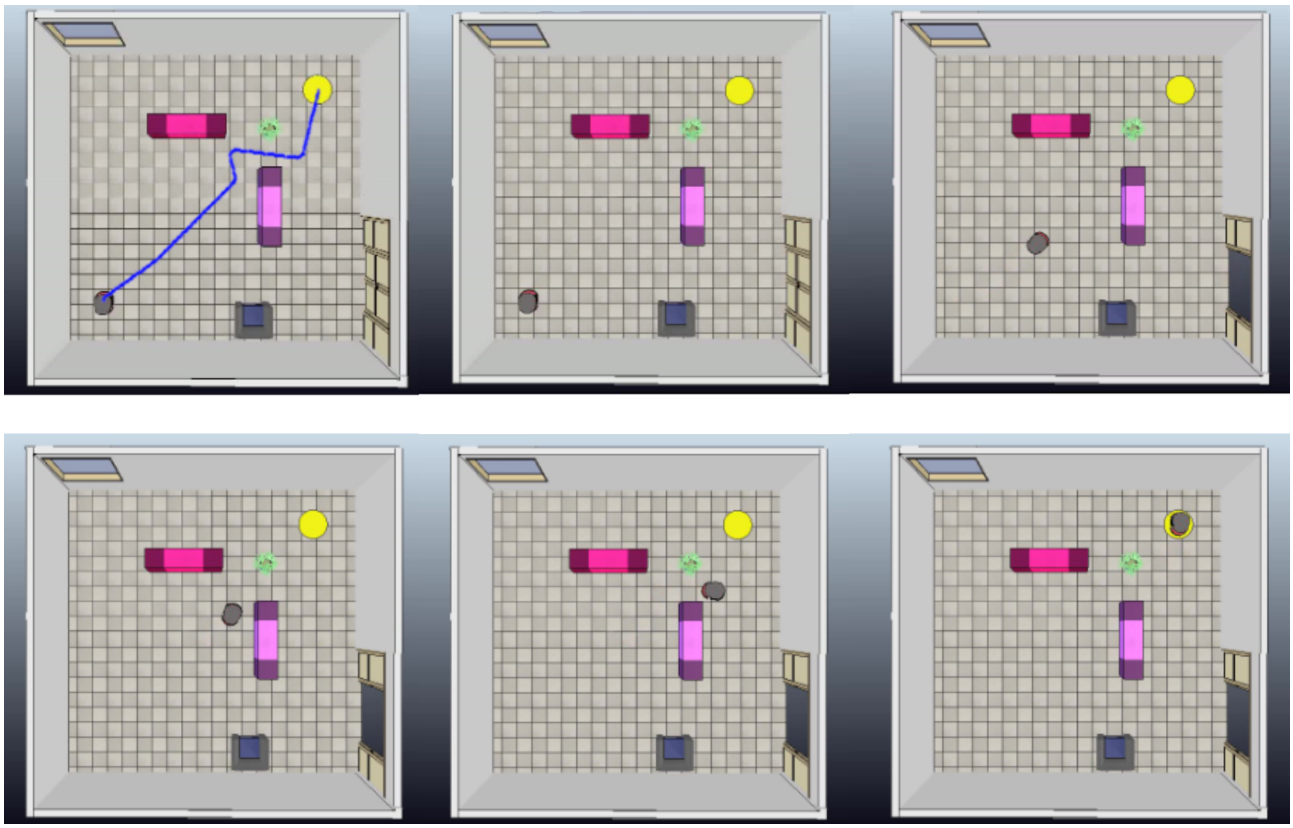

Figure 8: Optimal path found in Scene 2 with standard RRT* version. This solution has cost 12.59.

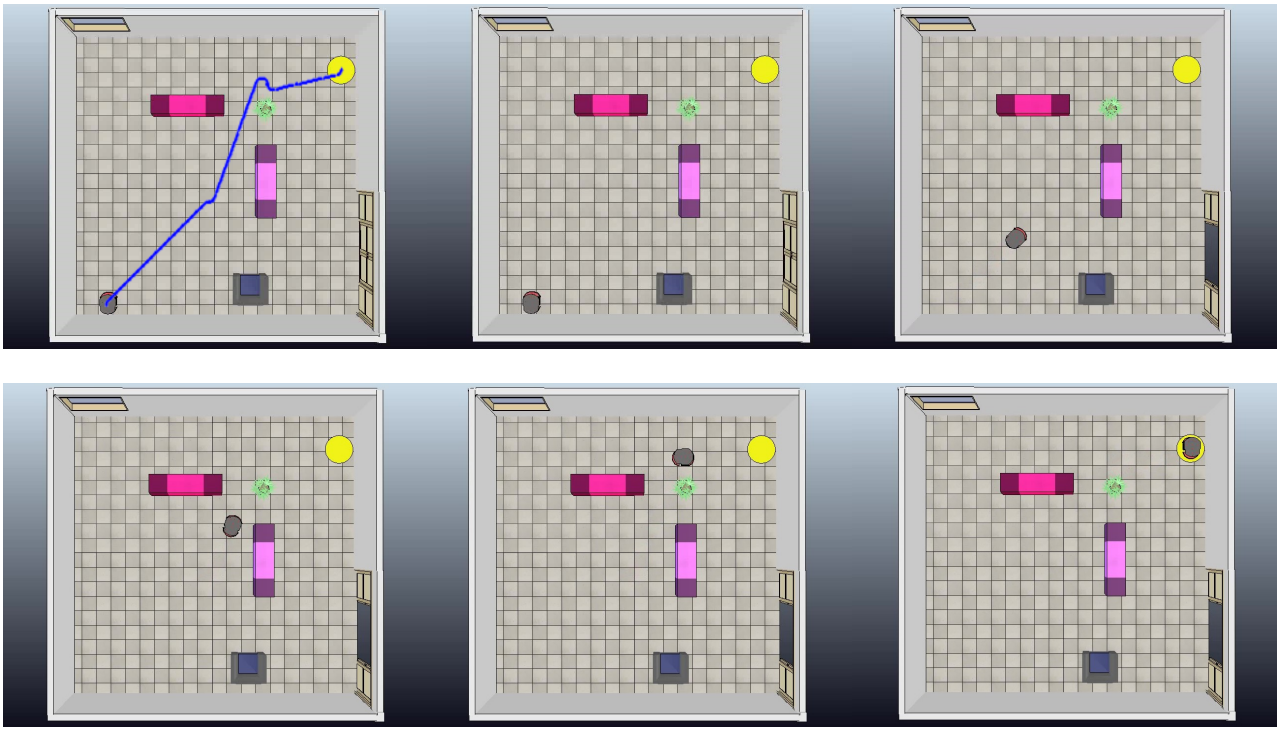

Figure 9: Optimal path found in Scene 2 with motion-primitive based RRT* version, using the dictionary with grid resolution 1.0 and 8 possible orientations. This solution has cost 15.79.

5.3 Scene 3

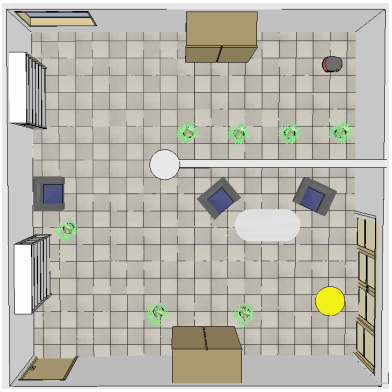

Figure 10: *Scene 3* - CoppeliaSim

The third scene is a 10x10 [m] floor, where many obstacles were distributed, creating the plant of an office. The initial position of the robot is $(-4.0, 4.0, -\pi/2)$, the center of the goal region is $(3.0, 4.0)$ and the desired final orientation is 0. The dictionaries that have been used are the same that were deployed for scene 2.
Table 3 shows the results obtained on scene 3 with the two different versions of RRT* and the different dictionaries of primitives. As it is possible to notice, in this case, the standard RRT* version is able to find a feasible path to the goal in all the simulations, with a consequent 100% of success rate. This result is in line with those obtained previously on scene 1 and 2. In fact, even in those simulations, this version of the algorithm was always able to reach the goal, except for the 100 iterations on scene 2. For what concerns the primitive-based version of the algorithm, an important aspect that is worth mentioning is how the success rate is decreased with respect to scene 1 and 2. This happens with all the three different dictionaries and is probably due to the scene's structure, which makes it much more difficult to find feasible paths to the goal. As it was predictable, also looking at the results obtained with the other scenes, all the simulations run with 100 iterations have a very low success rate, exceeding 50% only with the largest dictionary. As in scene 2, the worst results are obtained with the smallest dictionary. Talking about solution's costs, in this case we have obtained different results with respect to scene 2, since the algorithm has found lower cost solutions with the growing of the dictionaries' size, due to the absence of narrow passages. Considerations about computational time made previously for the other scenes still hold for this one. In fact, even this time the primitive-based version is less time-consuming, but at the cost of slightly more expensive solutions.

SCENE 3	n.primitives	n.iters	tot.time(s)	steering time(s)	tree size	sol.cost	success rate(%)
Standard RRT*	—	100	14.6	8.4	51.8	24.26	100
		1.000	90.4	48.6	611.4	22.74	100
		10.000	874.2	419.4	6180.8	22.66	100
Primitive -based RRT*	14 (24)	100	< 1	< 1	1	//	0
		1.000	1	< 1	6.1	32.46	10
		10.000	99	< 1	43.1	30.3	100
	325 (560)	100	2.5	$<= 1$	21.1	26.78	50
		1.000	22.7	5.7	246.4	26.35	100
		10.000	162	50.5	433.1	25.28	100
	1220 (1920)	100	3.4	1.4	24.3	25.08	60
		1.000	37.8	18	383.5	24.67	100
		10.000	287.3	91.7	1485.9	23.95	100

Table 3: Results obtained by applying the two algorithms to *Scene 3*.

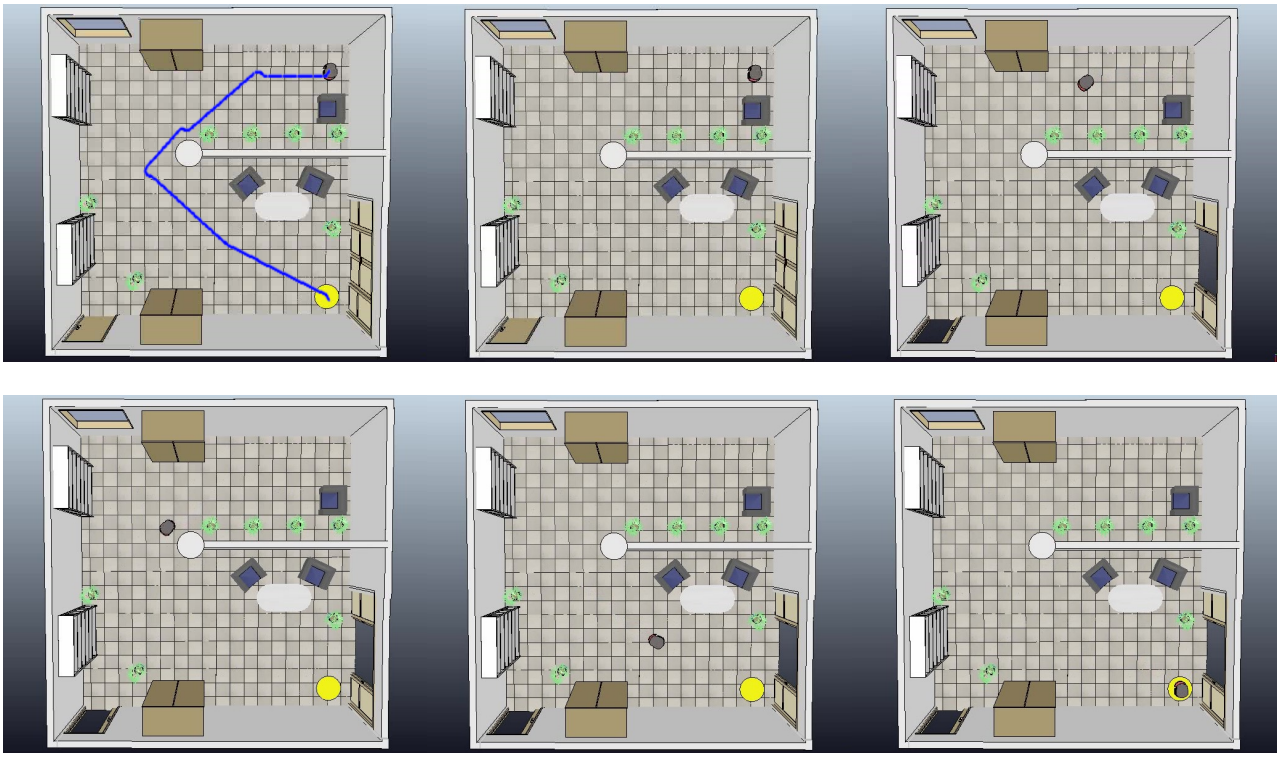

Figure 11: Optimal path found in Scene 3 with standard RRT* version. This solution has cost 21.30.

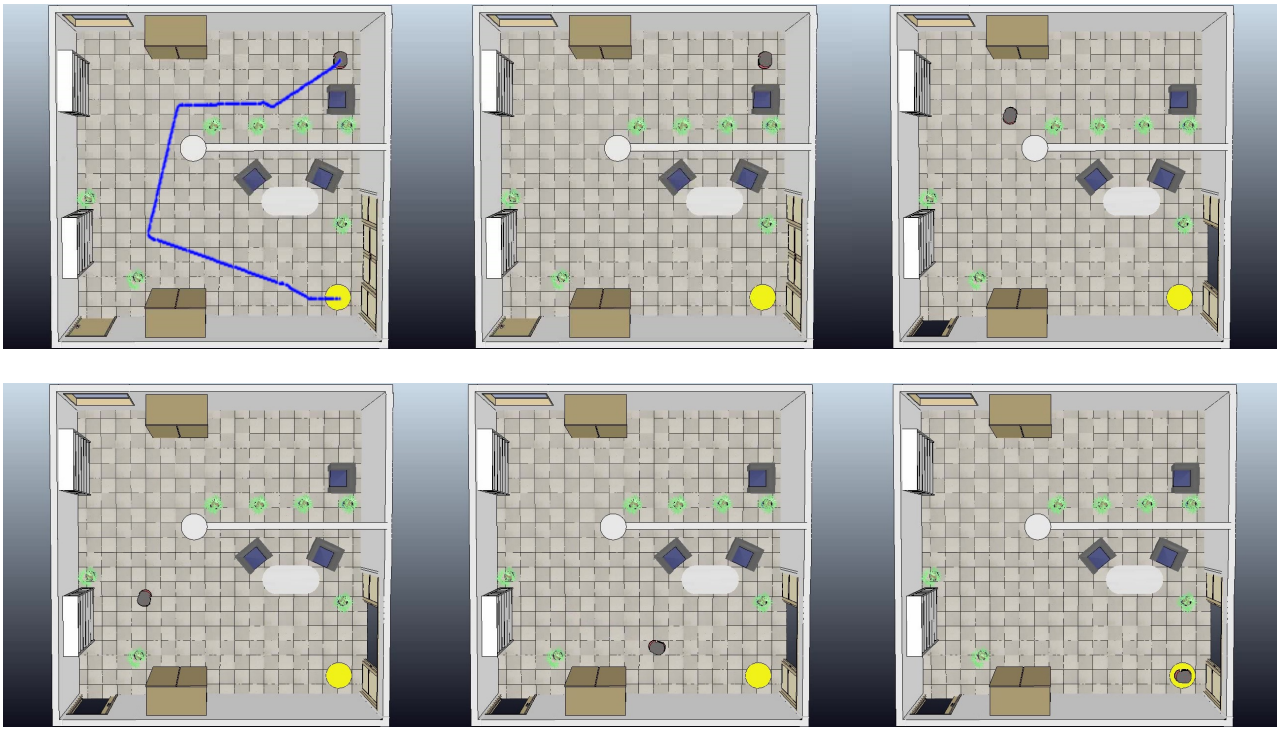

Figure 12: Optimal path found in Scene 3 with motion-primitive based RRT* version, using the dictionary with grid resolution 0.5 and 8 possible orientations. This solution has cost 19.41.

5.4 Scene 4

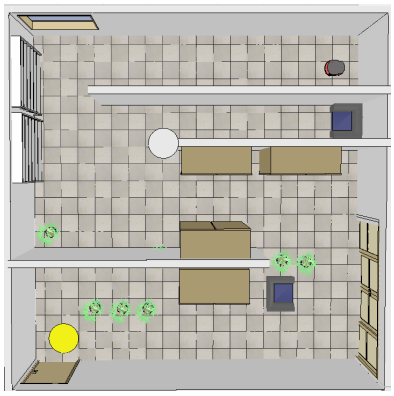

Figure 13: *Scene 4* - CoppeliaSim

The fourth scene is a 10x10 [m] floor, where many obstacles were distributed, and in particular walls were used to create several narrow passages, through which a feasible path is more difficult to find. The initial position of the robot is $(-4.0, 4.0, -\pi/2)$, while the desired goal configuration is $(4.0, -4.0, \pi)$. Even in this case, we have used the same dictionaries as in scenes 2 and 3, with 14, 325 and 1220 feasible motion primitives.
Table 4 shows the results obtained on scene 4 with the two different versions of RRT* and the different dictionaries of primitives.
The first thing that stands out is that the presence of many walls and obstacles make it impossible for the algorithm to find a solution path when running for few iterations. In fact, with 100 iterations neither algorithm succeeds in finding a path from the initial position to the goal. When exploiting the smallest dictionary of primitives, also the 1.000 iterations gets unsatisfactory results, with only one successful run over ten trials. Therefore, even the resulting values are not reliable for making comparisons. In all other cases, both implementations working with 1.000 iterations obtain almost similar and satisfactory success rates. Finally, 10.000 iterations turn out to be enough for always finding a solution with both methods, independently from the dictionary that is used.
Looking at the solution costs, the standard RRT* expectedly obtains better results, since it can sample points from the whole configuration space, allowing to obtain paths in which the robot remains closer to the walls and obstacles when taking the bends. Also the costs obtained with the second and third dictionaries of the primitive-based RRT* are the expected ones: with more primitives the resulting path is shorter because more refined. But surprisingly, when working with 10.000 iterations, the smallest dictionary of primitives allows to get the lowest costs. Actually, such result was predictable: in the first dictionary, although the feasible primitives are only 14, they are enough to find a short path, mainly due to two reasons. Firstly, the scene was created in such a way that with the smallest dictionary only a path was practically available, that passed for all the nine points of the grid. In fact, by skipping one of such points it is impossible to find a solution,

since there does not exist any primitive between two non-adjacent points, because of the presence of walls and obstacles. Secondly, there are only four orientations that can be sampled at each point of the grid. Therefore, in 10.000 iterations the algorithm will almost certainly find the best possible path, avoiding all the useless rotations. In fact, nearly all the paths obtained in the ten performed simulations are the same. On the contrary, when we are able to sample more points on the grid with different orientations, the path will contain more rotations, and this causes an increase of its length.
The observations made for the previous scene regarding the size of the tree and the computational time hold also for the fourth scene.

SCENE 4	n.primitives	n.iters	tot.time(s)	steering time(s)	tree size	sol.cost	success rate(%)
Standard RRT*	—	100	38.4	29.9	18.6	//	0
		1.000	274.7	218.8	293.1	39.32	70
		10.000	3031.1	2232.2	3712.2	39.35	100
Primitive -based RRT*	14 (24)	100	< 1	< 1	1.4	//	0
		1.000	2.2	0.8	6.7	46.62	10
		10.000	151	7.1	49.5	39.66	100
	325 (560)	100	1	< 1	3.2	//	0
		1.000	44.3	13.25	164.7	48.7	90
		10.000	393.2	131.7	368.5	46.12	100
	1220 (1920)	100	2.7	1.15	7.2	//	0
		1.000	60.4	31.2	211.4	46.8	80
		10.000	978.8	485.1	1233.6	45.86	100

Table 4: Results obtained by applying the two algorithms to *Scene 4*.

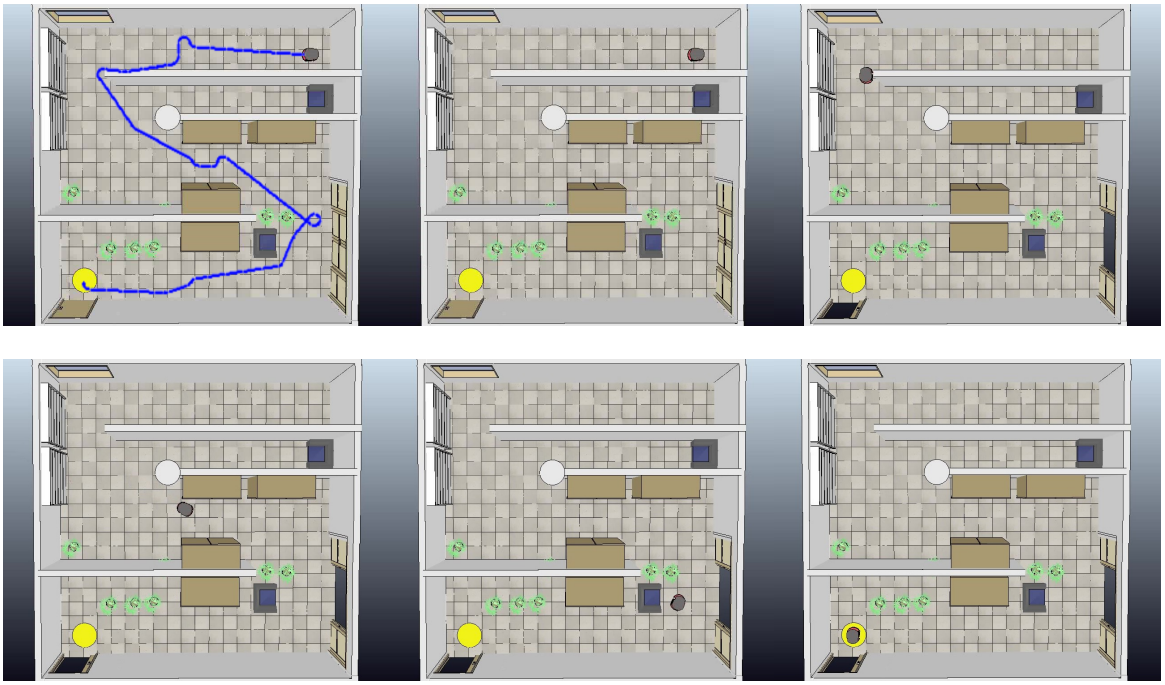

Figure 14: Optimal path found in Scene 4 with standard RRT* version. This solution has cost 35.41.

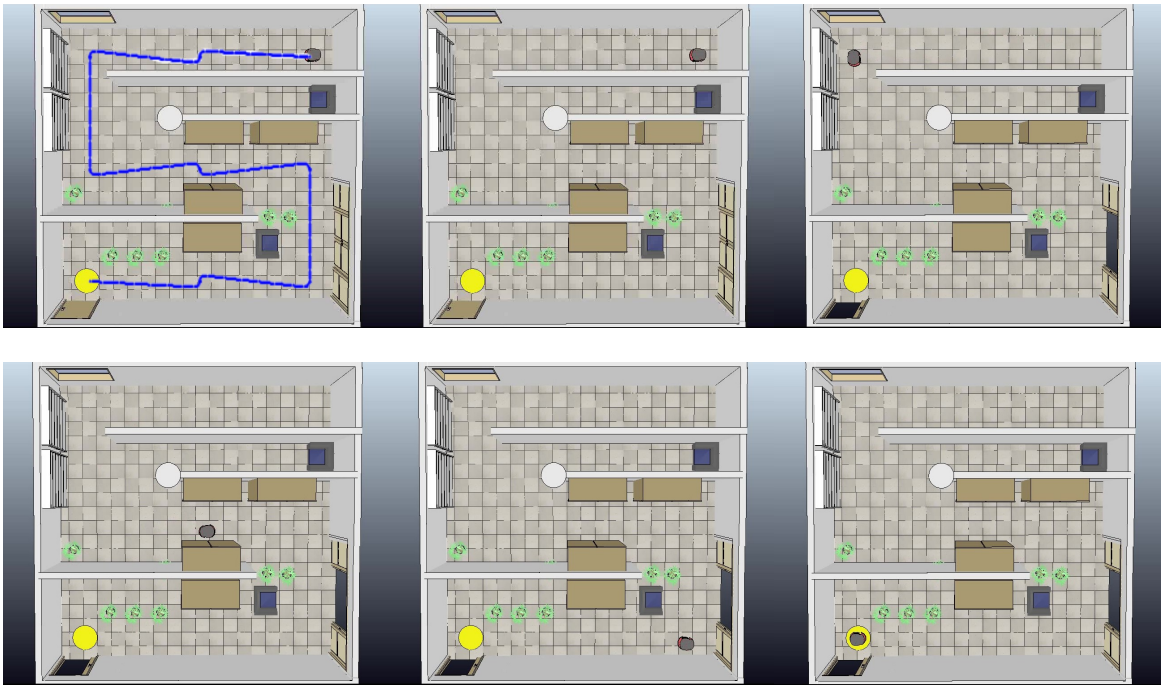

Figure 15: Optimal path found in Scene 4 with motion-primitive based RRT* version, using the dictionary with grid resolution 4.0 and 4 possible orientations. This solution has cost 36.84.

6 Extension to generic unicycle

As previously explained, the final version of this project has been carried out modelling the unicycle as a Dubins' vehicle. Another version was developed using the classical dynamics of the unicycle (able to go back and forth in the space) and the Two Point Boundary Value Problem in order to optimally connect two configurations. The algorithm was the same except for the steering function, that exploited the TPBVP solution, instead of the Dubins' curves. In this case, the steering is more difficult to compute and there exist approximations due to the need of dividing the total time for moving between two configurations into an arbitrary number of time intervals, during which the control inputs are piece-wise constant. These approximations cause errors in the reconstruction of the robot's trajectory, resulting in the robot to slightly slip along the path. This happened when interpolation was exploited in order to find the intermediate configurations between each two points returned by the TPBVP. To remove the slipping issue, interpolation was replaced by exact integration. However, it causes the robot not to arrive exactly in the end point. Therefore, a final manoeuvre has been applied to bring the robot in the exact final configuration, but this made the solution sub-optimal.
In this section, we present a more detailed explanation of the TPBVP version, which represents an extension of our project to a generic unicycle.

6.1 Two point boundary value problem (TPBVP)

Given a state tuple $\mathbf{q} \in Q$ and a boundary value pair $(\mathbf{q}_0, \mathbf{q}_f)$, a TPBVP is a constrained optimization problem in the form:

$$\begin{aligned} \underset{\mathbf{u}(.),\tau}{\text{minimize}} \quad & \int_0^\tau g(\mathbf{q}(t), \mathbf{u}(t))dt \\ \text{subject to} \quad & \dot{\mathbf{q}} = f(\mathbf{q}(t), \mathbf{u}(t)), \\ & \mathbf{u}(t) \in U, t \in [0, \tau], \\ & \mathbf{q}(t) \in Q_{free}, t \in [0, \tau], \\ & \mathbf{q}(0) = \mathbf{q}_0, \mathbf{q}(\tau) = \mathbf{q}_f \end{aligned} \tag{6}$$

in which we minimize a given cost function with respect to the inputs $\mathbf{u}$ and the time duration, subject to some constraints that are given respectively by the kynematic model of the unicycle, the bounds on the inputs, and the initial and final states

that we want to connect.

6.2 Implementation Details

In both versions of the algorithm we have solved in the *Steer* function the TPBVPs needed to determine the optimal path between two given configurations. To this end, we have used *ifopt*, a light-weight, Eigen-based C++ interface to Nonlinear Programming solvers, among which we have chosen *ipopt*. As anticipated, the development of the TPBVP is based on a discretization of the optimization problem described in the previous section. In particular, a constant and arbitrary number N of intervals has been considered, in which the control inputs are constant. This implies that minimizing the total duration of the path corresponds to minimizing the single intervals' durations δ_i.
The decision variables are the configurations, the control inputs and the time, repeated for each interval:

$$\mathbf{X} = (x_0, ..., x_N)\ ,\ \mathbf{Y} = (y_0, ..., y_N)\ ,\ \mathbf{\Theta} = (\theta_0, ..., \theta_N)$$
$$\mathbf{V} = (v_0, ..., v_{N-1})\ ,\ \mathbf{\Omega} = (\omega_0, ..., \omega_{N-1})$$
$$\mathbf{\Delta} = (\delta_0, ..., \delta_{N-1})$$

In the TPBVP statement, we have defined the following cost function:

$$\min_{\mathbf{X},\mathbf{Y},\mathbf{\Theta},\mathbf{V},\mathbf{\Omega},\mathbf{\Delta}} 1 + 0.5\,\mathbf{V} + 0.5\,\mathbf{\Omega} + 0.5\,\mathbf{\Delta} \tag{7}$$

that is minimized for each interval, considering the corresponding decision variables. A number of constraints were also imposed to the problem: the pure rolling constraint for the unicycle *(2)* and the bounds on the maximum and minimum values of the control inputs and on the initial and final configurations.
The configurations (x, y, θ) returned by the solver were connected through the exact integration:

$$\begin{aligned} x_{k+1} &= x_k + \frac{v_k}{w_k}(\sin\theta_{k+1} - \sin\theta_k), \\ y_{k+1} &= y_k - \frac{v_k}{w_k}(\cos\theta_{k+1} - \cos\theta_k), \\ \theta_{k+1} &= \theta_k + w_k T_s. \end{aligned} \tag{8}$$

which exploits the velocity inputs returned by the TPBVP.
Because of approximation errors, the use of exact integration caused the robot to end a path between two configurations in the tree not exactly in the end point. For this reason, since the final path from q_{ini} to q_{fin} is a concatenation of the paths connecting two adjacent nodes in the tree, the robot jumped from the wrong end point of the path to the initial configuration of the next path.

In order to solve this problem, we added to each path returned by a TPBVP a final manoeuvre made of three steps, which are necessary to connect the reached point and the real final point of the path:

- the robot is rotated by the pointing angle, which is the angle between its current orientation and the orientation of the line connecting its current position to the desired one (1);
- the robot moves on a straight line to reach the x, y coordinates of the final configuration (2);
- the robot is rotated to match the angle of the desired configuration (3).

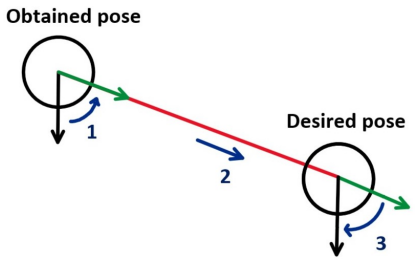

Figure 16: Graphical representation of the final manoeuvre

6.3 Planning Experiments

In order to validate our work, some results obtained by applying this method on scene 3 are shown in Table 5, with both the standard implementation and the primitive-based implementation exploiting the intermediate grid-resolution.
It can be easily noticed that most of the claims done previously still hold. The main aspects to be highlighted are the following:

- The path costs decrease with the growing number of epochs in both the algorithm's versions.
- The primitive-based RRT* computes higher cost solutions with respect to the standard RRT*, with the advantage of a lower computational time.
- The solution costs obtained by exploiting the Dubins' curves are lower in all experiments, since the TPBVP causes the solution to be sub-optimal because of the final adjustment maneuver.

SCENE 3	n.primitives	n.iters	tot.time(s)	steering time(s)	tree size	sol.cost	success rate(%)
Standard RRT*	—	100	79	36	37	34.2	50
		1.000	767	389	493	28	100
		10.000	8705	4238	6043	26.6	100
Primitive -based RRT*	325 (560)	100	36	< 1	41	34.5	60
		1.000	221	52.7	255	29.1	100
		10.000	1709	423.3	430	27.3	100

Table 5: Results obtained by applying the two algorithms to *Scene 3* with the TPBVP exploited in the steering function.

7 Conclusions

In this paper we have implemented two versions of a very popular algorithm for motion planning, that is RRT*. The first one is RRT* in its original form, while the second one uses a database of pre-computed motion primitives to reduce the execution time, avoiding to repeatedly compute the trajectories needed for connecting the nodes in the tree. In fact, with this second version, the steering function simply retrieves the pre-computed primitives from the database, and therefore it is much quicker. The results obtained in our experiments confirmed this claim, since the execution times we have reported are much shorter in the case of the primitive-based algorithm. Moreover, in the standard version it can be easily noticed that the computation of the steering function takes most of the time.
In order to make further comparisons, we have considered three databases of motion primitives, using several grid resolutions and different numbers of possible orientations. The increase in speed comes at the price of a rise in the cost of the solution and a reduction in the success rate since, due to the discretization of the state space, it becomes more difficult to find feasible paths. In our scenes, because of the placements of the obstacles and walls, differently from what we had thought, we managed to obtain better results, both in terms of path cost and execution time, using the smallest dictionary, as long as sufficiently high number of iterations was considered. In fact, the largest dictionaries determine an increase in the number of configurations that do not belong to the optimal paths going through narrow passages, and therefore the probability of picking the optimal points (for example in scene 2 the points between the plant and the dressers) becomes lower. On a different scene, in which the points in the narrow passages do not fall on the discrete grid, the primitive-based algorithm would not be able to find a solution, so a finer resolution should be considered. To support these observations, we bring to the reader's attention that in scene 3, in which we do not have narrow passages, the path cost reduces at the increase of the number of primitives.
A similarity between the two versions of RRT* consists in the fact that the path costs become lower at the increase of the number of iterations, since the tree is rewired every time a node is added and the paths gets refined. Concerning the primitive-based algorithm, it must be noted that, exception made for scene 1, it is not able to find a solution in 100 iterations while using the smallest dictionary, and in the most complex scenes even with the larger ones.
In conclusion, both versions have advantages and drawbacks. Therefore, the choice must be done evaluating the needs related to the considered application and also the computational resources. If a quicker and lighter algorithm is needed, the primitive-based version would be better, while if it is necessary to have the solutions with the lowest possible cost, then standard RRT* should be used.

References

[1] Sertac Karaman and Emilio Frazzoli. “Optimal Kinodynamic Motion Planning using Incremental Sampling-based Methods”. In: *49th IEEE Conference on Decision and Control* (2010).

[2] Sertac Karaman and Emilio Frazzoli. “Sampling-based Algorithms for Optimal Motion Planning”. In: (2011). arXiv: 1105.1186.

[3] L.E. Kavraki et al. “Probabilistic roadmaps for path planning in high-dimensional configuration spaces”. In: *IEEE Transactions on Robotics and Automation* 12.4 (1996), pp. 566–580. DOI: 10.1109/70.508439.

[4] Steven M. LaValle. “Rapidly-exploring random trees: a new tool for path planning”. In: *The annual research report* (1998).