J Intell Robot Syst
https://doi.org/10.1007/s10846-017-0693-4

Spline-based RRT* Using Piecewise Continuous Collision-checking Algorithm for Car-like Vehicles

Sangyol Yoon[1] · Dasol Lee[2] · Jiwon Jung[2] · David Hyunchul Shim[2]

Received: 19 September 2016 / Accepted: 1 August 2017

Abstract This paper presents a path planning algorithm that can efficiently check for interference with potential obstacles while piecewise continuously computing the required space of moving car-like vehicles using cubic Bezier curves. Our collision-checking algorithm uses trajectories generated from a vehicle's front outer corner and rear inner axle, as well as partially overlapped rectangles. These outer and inner trajectories are computed from the trajectory generated by the center of the rear axle of the vehicle, which considers the dimensions of the vehicle, and the tangential and normal vectors of the trajectory. To validate the continuity and efficacy of our collision-checking algorithm, the collision-checking algorithm is applied to a spline-based RRT*, where the kinematics (or minimum turning radius) of car-like vehicles is satisfied using cubic Bezier curves. We show the benefits of our method through simulations and experimental results by using an autonomous ground vehicle.

Electronic supplementary material The online version of this article (https://doi.org/10.1007/s10846-017-0693-4) contains supplementary material, which is available to authorized users.

✉ Sangyol Yoon
sangyol.yoon@kaist.ac.kr

Dasol Lee
dasol@kaist.ac.kr

Jiwon Jung
jeongjiwon@kaist.ac.kr

David Hyunchul Shim
hcshim@kaist.ac.kr

[1] LG Electronics Inc., 19, Yangjae-daero 11-gil, Seocho-gu, Seoul 06772, Korea

[2] Department of Aerospace Engineering, KAIST, 291 Daehak-ro, Yuseong-gu, Daejeon 34141, Korea

Keywords Collision checking · Bezier curve · Car-like vehicle · RRT* · Autonomous driving

1 Introduction

Collision-free paths are important in enabling autonomous vehicles to move toward their destination. In particular, car-like vehicles are restricted by their minimum turning radius and shape, which makes the space required when executing a turn greater than that required to move in a straight line. This additional constraint complicates the collision-free path problem for car-like vehicles.

Various path planners have been developed to generate collision-free paths for car-like vehicles, such as multi-resolution lattice [1], Hybrid A* [2], variant of A* using reduced states [3], replanning methods [4, 5], probabilistic roadmaps (PRM) [6], kinodynamic rapidly-exploring random tree (RRT) [7, 8], kinodynamic RRT* [9, 10], spline-based RRT (S-RRT) [11] and optimal spline-based RRT (S-RRT*) [12]. These methods typically employ Dubins curves [13] or splines [14]. In addition, many variants of steer functions use optimal controllers and closed-loop predictions [15–17]. In general, sampling-based algorithms such as PRM and RRT are able to solve the low sampling density problem in narrow passages by employing adaptive sampling [18, 19]. However, some of these approaches do not compute the turning space of the vehicles continuously. Instead, they assume that the shape of a vehicle is a point, and add circles corresponding to the width of the vehicle [12] or its outline using a Minkowski sum to obstacles represented by a point cloud (acquired when using laser scanners) or polygon [20, 21]. These treatments cannot continuously represent the space required for the vehicle to turn, because the outer side of the rectangle vehicle requires more

Published online: 25 September 2017

space than the inner side or angularly discretized rectangles are added to obstacles (configuration space or c-space). Also, the technique that makes the c-space using Minkowski sum can make it more suitable to handle static obstacles than dynamic obstacles because more computation time is needed when the dimension of the vehicle (three dimensions in our application) is increased [22]. So, in general, angularly discretized c-space is precomputed to efficiently use the technique in a static obstacle environment. Dolgov et al. and Minguez et al. computed the turning space required by a vehicle for a path with constant curvature [2, 23], but did not consider variable curvature. To the best of our knowledge, there is no continuous and efficient means of computing the required space of a moving car-like vehicle when planning collision-free paths in a complex environment with obstacles.

In this study, we propose a collision-checking algorithm that computes the required space of moving car-like vehicles using cubic Bezier curves with variable curvature. The proposed method checks for interference with obstacles by computing two trajectories, one from the front outer corner and one from the rear inner axle of the vehicle, instead of the trajectories generated by all corners of the vehicle (see Fig. 1a). Also, our obstacle-checking is employed by partially placing a rectangle representing the vehicle on the generated path at transition points, e.g., between two concatenated Bezier curves and when the vehicle starts to turn, finishes turning, and changes its turning direction. Furthermore, to validate the continuity and efficacy of our collision-checking algorithm, we employ a shape-aware spline-based RRT* (SS-RRT*), where the nonholonomic constraints (or minimum turning radius) of vehicles are satisfied using cubic Bezier curves [24].

2 Related Work

The avoidance of obstacles is a fundamental requirement of a collision-free path. In complex environments with multiple obstacles, path planners must therefore respect the shape and turning space of the car-like vehicle following the given path.

Bicchi et al. and Geraerts et al. represented a vehicle as a point, and added circles surrounding the vehicle to obstacle or corridor maps [25, 26]. However, under these algorithms, vehicles cannot pass through a narrow passage if their computed radius exceeds the width of the passage. Pruski et al. and Varadhan et al. proposed solutions to this narrow-passage problem by taking the Minkowski sum of rectangles [20, 21]. Also, to check for collisions along a continuous path, Maekawa et al. placed a rectangular vehicle shape at each sampled point on the path such that the orientation corresponded to the tangent of the path [27]. We call it rectangle-sampling method. However, these approaches cannot continuously represent the space required by a turning vehicle, i.e., as shown in Fig. 1b, particularly for obstacles outside the vehicle, the vehicle will collide with an obstacle, but the rectangle-sampling method cannot detect this collision unless it overlaps with a sufficient number of rectangles. In other words, the discretization is limited. Gomez et al. also checked for collisions on continuous paths, but their method can only be applied to parallel and diagonal parking because it checks for collisions by using geometrical conditions such as the width and the height of parallel/diagonal parking slot, and the distance between the vehicle and the parking slot [28].

In contrast, our proposed method is a piecewise continuous and efficient collision-checking algorithm for path planners with variable curvature. The algorithm uses piecewise continuous trajectories and partially rectangular vehicle shapes placed on the paths, and can be applied to moving car-like vehicles. In this study, we compare our method with the rectangle-sampling method among collision-checking algorithms using angularly discretized rectangles (Minkowski sum and rectangle-sampling method).

3 Overview of Our Approach

In this section, we provide an overview of the procedure used to compute the space required by a moving car-like vehicle to develop our collision-checking algorithm (Section 4).

In this study, we approximate the shape of a vehicle as an oriented rectangle (Fig. 2a), because this closely approximates the shape of many vehicles. Thus, we focus on developing a collision-checking algorithm that computes

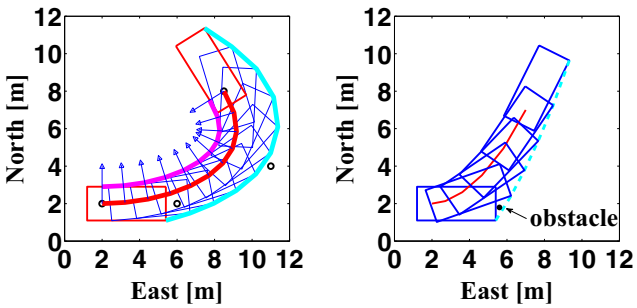


Fig. 1 **a** The trajectories computed from a cubic Bezier curve provide an excellent representation of the space required by a turning vehicle. Rectangles indicate the vehicle, and red, cyan, and magenta curves represent the Bezier curve and the trajectories of the front outer corner and rear inner axle, respectively. Black circles and arrows indicate the control points for the Bezier curve and its normal vector. **b** Rectangles located on the sampled points cannot detect a collision continuously because the discretization is limited. The dashed line represents the approximate trajectory of the right front corner of the vehicle

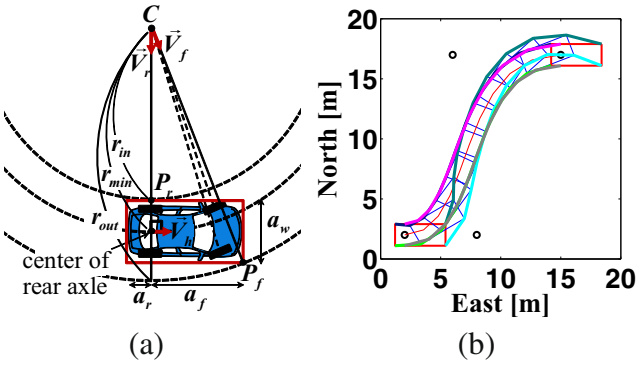


Fig. 2 **a** Illustration of the space required to make a turn for a front-wheel-steering vehicle under kinematic constraints. **b** Selecting all trajectories is less efficient because some of those used to check for interference with obstacles are surrounded by other trajectories. Black circles denote control points for the Bezier curve

the required space of a moving vehicle, and then apply this algorithm to spline-based RRT*.

We compute the space required by a moving vehicle in two scenarios: moving in a straight line, and turning. The process of checking for interference with obstacles along a straight-line path is relatively simple, as we must only consider the trajectories generated from the width of the vehicle. However, when a car-like vehicle turns, it follows an arced trajectory, which makes the collision-checking procedure more complex.

In this study, we consider a path planning method for low-speed car-like vehicles in cluttered obstacle environments. Therefore, we assume that the slip angle at each wheel is zero. Thus, the center of the arc followed by the turning vehicle (C in Fig. 2a) is the point at which the normals to the direction of each wheel intersect [29]. Thus, for a vehicle that is steered by its front wheels, we can assume that the paths generated follow the center of the vehicle's rear axle (red box in Fig. 2a). Also, we assume that the vehicles can follow the generated paths with minimal error using path-following controllers [30]. From the center of curvature C, the inner and outer arcs are computed by the geometric relation shown in Fig. 2a. The turning radii of these inner and outer arcs are denoted by r_{in} and r_{out}, respectively, and are calculated as follows:

$$\begin{aligned} r_{in} &= r_{min} - 0.5a_w \\ r_{out} &= \sqrt{(r_{min} + 0.5a_w)^2 + a_f^2}\,, \end{aligned} \tag{1}$$

where a_w and a_f are the width and front overhang of the vehicle, respectively. After constructing these inner and outer arcs, we can use them to check for interference with obstacles.

In Section 4, we describe our collision-checking algorithm for car-like vehicles. First, we analyze the relation between the vehicle's trajectory and its required space to check for interference with obstacles, especially when making a turn. Then, to determine any interference with obstacles in our path planner (SS-RRT*), we extend the relationships to Bezier curves. Furthermore, we propose a collision-checking algorithm that computes the required space of a moving vehicle using just two trajectories, instead of the trajectories generated by all corners of the vehicle. Finally, we describe the SS-RRT* implementation for our collision-checking algorithm.

4 Collision-checking Algorithm for Car-like Vehicles

In this section, we propose a collision-checking algorithm that piecewise continuously and efficiently checks for interference between obstacles and the trajectories generated by the shape of a car-like vehicle. To do this, we parameterize dominant trajectories among the trajectories generated by all of the corners of the vehicle from the paths (or Bezier curves) planned by our path planner (SS-RRT*).

4.1 Trajectories Generated by the Shape of a Car-like Vehicle

When a vehicle turns, each corner of the vehicle generates a different trajectory according to its kinematics. In this section, we investigate which trajectories are dominant, and check these trajectories for interference with obstacles.

Figure 2b shows six trajectories generated from the front/rear corners and the rear axle of the vehicle. This figure shows that some trajectories are surrounded by other trajectories, i.e., there exist dominant trajectories to check for interference with obstacles. As shown in Fig. 2a, the turning space required by the shape of the vehicle is mainly determined by its front outer corner and rear inner axle. Therefore, to check for interference with obstacles, we need only compute the trajectories of the front outer corner and the rear inner axle. This approach is more effective than computing the trajectories of all corners of the vehicle.

Given the center C of an arc generated by the center of the rear axle, the positions of the front outer corner, P_f, and rear inner axle, P_r, are computed as follows:

$$P_f = r_{out}\overrightarrow{V_f} + C,\;\; P_r = r_{in}\overrightarrow{V_r} + C, \tag{2}$$

where $\overrightarrow{V_f}$ and $\overrightarrow{V_r}$ are the unit vectors from C toward P_f and P_r, respectively. In particular, $\overrightarrow{V_r}$ is the unit vector normal to the arc and $\overrightarrow{V_f}$ can be computed as follows:

$$\overrightarrow{V_f} = \frac{a_f\overrightarrow{V_h} + (r_{min} + 0.5a_w)\overrightarrow{V_r}}{||a_f\overrightarrow{V_h} + (r_{min} + 0.5a_w)\overrightarrow{V_r}||}, \tag{3}$$

where $\overrightarrow{V_h}$ is the unit vector tangential to the arc.

4.2 Trajectories Based on Bezier Curves

When generating paths for a car-like vehicle, we must satisfy the nonholonomic constraint and ensure that the generated paths are continuous. To do this, it is important to consider the vehicle's minimum turning radius. Bezier curves can construct paths in an efficient manner while satisfying these constraints [24]. The method uses two concatenated Bezier curves as a motion primitive for path planning (Fig. 7). In this section, we parameterize the dominant trajectories in Eq. 2.

In this study, we use a cubic Bezier curve, which is formulated as follows [31]:

$$B(t) = (1-t)^3 P_0 + 3(1-t)^2 t P_1 + 3(1-t)t^2 P_2 + t^3 P_3, \quad (4)$$

where P_0–P_3 are control points and $t \in [0, \ 1]$ is a parameter.

To obtain the dominant trajectories generated by the shape of a car-like vehicle, we must compute the tangential and normal unit vectors $\overrightarrow{V_h}$ and $\overrightarrow{V_r}$ and their center C. The tangential and normal unit vectors of a parameterized arc are computed as follows [32]:

$$\overrightarrow{V_h}(t) = \frac{B'(t)}{||B'(t)||},$$
$$\overrightarrow{V_r}(t) = -\frac{B'(t) \times B''(t) \times B'(t)}{||B'(t) \times B''(t) \times B'(t)||}, \quad (5)$$

where primes indicate differentiation with respect to t. The center $C(t)$ for a specific point $B(t)$ on a parameterized arc can be computed as follows:

$$C(t) = B(t) - \frac{1}{\kappa(t)} \overrightarrow{V_r}(t), \quad (6)$$

where $\kappa(t)$ is the curvature of the arc, computed as [32]:

$$\kappa(t) = \frac{||B'(t) \times B''(t)||}{||B'(t)||^3}. \quad (7)$$

The trajectories of the front outer corner and the rear inner axle are obtained by substituting (1) and (5–7) into (2) to give:

$$P_f(t) = \sqrt{\left(\frac{1}{\kappa(t)} + 0.5a_w\right)^2 + a_f^2}\ \overrightarrow{V_f}(t) + C(t),$$
$$P_r(t) = \left(\frac{1}{\kappa(t)} - 0.5a_w^2\right) \overrightarrow{V_r}(t) + C(t). \quad (8)$$

Figure 1a shows that the space required by a turning vehicle can be computed from the trajectories of the front outer corner and the rear inner axle using a cubic Bezier curve.

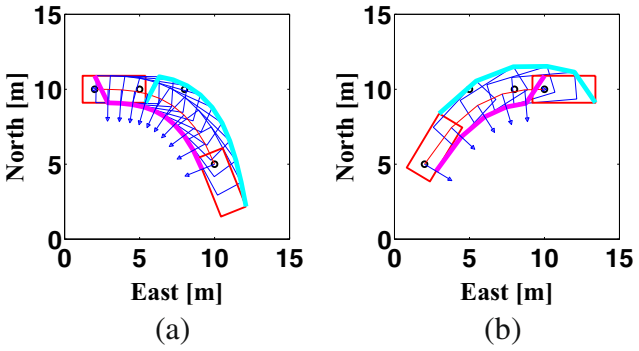


Fig. 3 If a vehicle starts to turn from a straight line (**a**) or finishes turning (**b**), trajectories cannot cover all required space

4.3 Description of Our Collision-checking Algorithm

The trajectories of the front outer corner and rear inner axle can represent the space required by a turning vehicle. We now examine whether these trajectories can represent the space required during transitions, because they do not include the shape of the vehicle at the start and finish of the motion primitive (see Fig. 1a (red boxes)). These transitions can be divided into three categories: (i) Points at which the vehicle starts or finishes turning; (ii) The intersection of two concatenated motion primitives; (iii) Points at which the vehicle changes its turning angle within a motion primitive.

In the first case, when the vehicle is located on a straight line, we define one of the front corners as the front outer corner, and define the inner rear axle accordingly. (If the vehicle starts turning toward the selected corner, the side describing the front outer and inner corners is switched, as shown in Fig. 3a and b.) However, these trajectories do not cover all of the space required during the change in turning direction. To solve this problem, we define the front outer corner (and rear inner axle) for a straight line to be the same as for a curved line, depending on the turning direction. In other words, if the vehicle is located on a straight-line path in the current step (line 2 in Algorithm 1) and starts to make a right turn in the next step (line 3 in Algorithm 1), we define the final front outer corner for the straight line to be the left front corner (line 4 in Algorithm 1). We also refine the final rear inner axle for the straight line in a similar manner (line 5 in Algorithm 1), and perform the same process if the vehicle starts to make a left turn from the straight path (lines 6 to 9 in Algorithm 1). When the vehicle finishes turning (line 11 in Algorithm 1), we check whether it made a right or left turn (lines 12 and 15 in Algorithm 1), and define its final front outer corner and rear inner axle accordingly (lines 13 and 14 in Algorithm 1 for a right turn, and lines 16 and 17 in Algorithm 1 for a left turn). In this algorithm, Veh_info denotes information about the vehicle dimensions, such as a_f and a_w.

Algorithm 1 Logic when refining the trajectories

Procedure $(P_f,\ P_r) = refTraj(B,\ \overrightarrow{V_h},\ \overrightarrow{V_r},\ Veh\_info)$

1: **for** $t = 0 \rightarrow 1$ **step** $tstep$ **do**
2: **if** $\overrightarrow{V_h}(t) \times \overrightarrow{V_r}(t) = 0$ AND $t + tstep \leq 1$ **then**
3: **if** $\left(\overrightarrow{V_h}(t + tstep) \times \overrightarrow{V_r}(t + tstep)\right) \cdot \overrightarrow{N} > 0$ **then** //from a straight line to a right turn
4: $P_f(t) \leftarrow B(t) + P_{fL}(\overrightarrow{V_h}(t),\ Veh\_info)$
5: $P_r(t) \leftarrow B(t) + P_{rR}(\overrightarrow{V_h}(t),\ Veh\_info)$
6: **else if** $\left(\overrightarrow{V_h}(t + tstep) \times \overrightarrow{V_r}(t + tstep)\right) \cdot \overrightarrow{N} < 0$ **then** //from a straight line to a left turn
7: $P_f(t) \leftarrow B(t) + P_{fR}(\overrightarrow{V_h}(t),\ Veh\_info)$
8: $P_r(t) \leftarrow B(t) + P_{rL}(\overrightarrow{V_h}(t),\ Veh\_info)$
9: **end if**
10: **end if**
11: **if** $\overrightarrow{V_h}(t) \times \overrightarrow{V_r}(t) = 0$ AND $t - tstep \geq 0$ **then**
12: **if** $\left(\overrightarrow{V_h}(t - tstep) \times \overrightarrow{V_r}(t - tstep)\right) \cdot \overrightarrow{N} > 0$ **then** //from a right turn to a straight line
13: $P_f(t) \leftarrow B(t) + P_{fL}(\overrightarrow{V_h}(t),\ Veh\_info)$
14: $P_r(t) \leftarrow B(t) + P_{rR}(\overrightarrow{V_h}(t),\ Veh\_info)$
15: **else if** $\left(\overrightarrow{V_h}(t - tstep) \times \overrightarrow{V_r}(t - tstep)\right) \cdot \overrightarrow{N} < 0$ **then** //from a left turn to a straight line
16: $P_f(t) \leftarrow B(t) + P_{fR}(\overrightarrow{V_h}(t),\ Veh\_info)$
17: $P_r(t) \leftarrow B(t) + P_{rL}(\overrightarrow{V_h}(t),\ Veh\_info)$
18: **end if**
19: **end if**
20: **end for**

In Algorithm 1, $\overrightarrow{N}$ is the unit vector normal to a plane, and $P_{fR}(\cdot)$, $P_{fL}(\cdot)$, $P_{rR}(\cdot)$, and $P_{rL}(\cdot)$ are functions to compute the positions of the right/left front corners and rear axle, respectively. These positions are calculated as follows:

$$
\begin{aligned}
P_{fR}(\cdot) &= 0.5a_w \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \overrightarrow{V_h} + a_f \overrightarrow{V_h}, \\
P_{fL}(\cdot) &= 0.5a_w \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \overrightarrow{V_h} + a_f \overrightarrow{V_h}, \\
P_{rR}(\cdot) &= 0.5a_w \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \overrightarrow{V_h}, \\
P_{rL}(\cdot) &= 0.5a_w \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \overrightarrow{V_h}.
\end{aligned} \tag{9}
$$

Figure 4 shows that the trajectories can cover the required space when a vehicle starts to turn from a straight line or finishes turning after applying Algorithm 1 to the scenario in Fig. 3a and b.

In the second case, a collision-checking algorithm can be implemented as follows. For a path generated along each Bezier curve, we check for interference with obstacles using the rectangle of the vehicle at the start and end positions (lines 5 to 10 in Algorithm 2).

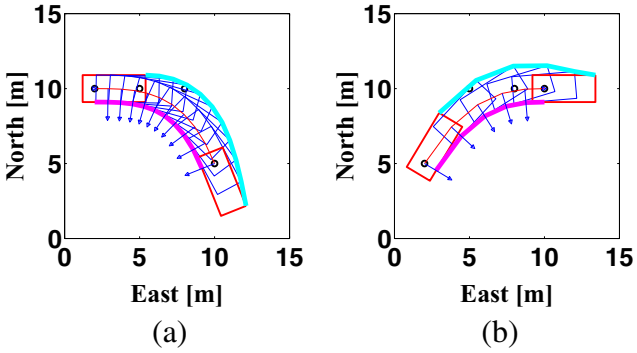


Fig. 4 After applying Algorithm 1 to the scenario in Fig. 3a and b, trajectories can cover the required space when a vehicle starts to turn from a straight line (**a**) or finishes turning (**b**)

In the third case, when a vehicle changes its turning direction (lines 14 and 20 in Algorithm 2), the side describing the front outer corner and rear inner axle is switched. In this case, as shown in Fig. 5a, the trajectories cannot cover the required space, even if Algorithm 1 is applied. This is mainly because the outer and inner trajectories of the vehicle are occupied by its front corner and rear axle, respectively. To solve this problem, we check for interference with obstacles using the rectangle of the vehicle during the transition (lines 15, 16, 21, and 22 in Algorithm 2).

A collision-checking algorithm that can handle these transitions is given in Algorithm 2, where $chkCollVeh(\cdot)$ checks for interference with obstacles using the rectangle of the vehicle. In particular, Algorithm 2 is applied when $tstep$ is determined for the rectangle of the vehicle as overlapping with another $tstep$, i.e, the length of a Bezier curve discretized by $tstep$ is less than that of the vehicle $(a_f + a_r)$.

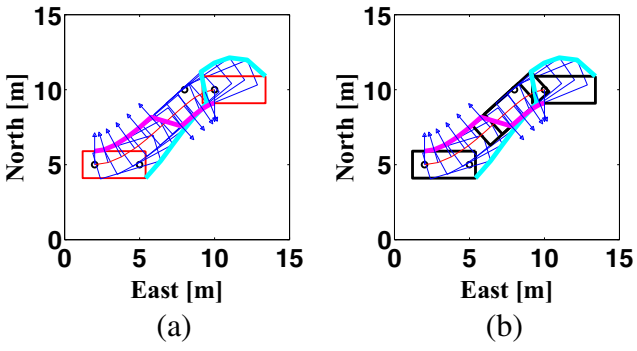


Fig. 5 **a** Trajectories cannot cover the space required when a vehicle changes the direction it is turning. **b** Trajectories can cover the space required using rectangles representing a vehicle to check for interference with obstacles when the vehicle changes the direction it is turning

Algorithm 2 Collision-checking algorithm that considers transition conditions between trajectories

Procedure $chkColl = chkCollTrans(\overrightarrow{V_h},\ \overrightarrow{V_r},\ Veh\_info)$
$chkColl \leftarrow$ Collision-free
$chkColl1 \leftarrow$ Collision-free
$chkColl2 \leftarrow$ Collision-free
for $t = 0 \rightarrow 1$ **step** $tstep$ **do**
  if $t = 0$ OR $t = 1$ **then** //at start and end positions
    $chkColl \leftarrow chkCollVeh(\overrightarrow{V_h}(t),\ Veh\_info)$
    if $chkColl =$ Collision **then**
      exit
    end if
  end if
  if $t - tstep \geq 0$ **then**
    $\overrightarrow{V_1} \leftarrow \overrightarrow{V_h}(t) \times \overrightarrow{V_r}(t)$
    $\overrightarrow{V_2} \leftarrow \overrightarrow{V_h}(t - tstep) \times \overrightarrow{V_r}(t - tstep)$
    if $\overrightarrow{V_1} \cdot \overrightarrow{V_2} < 0$ **then** //change to different turning direction
      $chkColl1 \leftarrow chkCollVeh(\overrightarrow{V_h}(t),\ Veh\_info)$
      $chkColl2 \leftarrow chkCollVeh(\overrightarrow{V_h}(t - tstep),\ Veh\_info)$
    end if
    if $t + tstep \leq 1$ **then**
      $\overrightarrow{V_3} \leftarrow \overrightarrow{V_h}(t + tstep) \times \overrightarrow{V_r}(t + tstep)$
      if $\overrightarrow{V_h}(t) = 0$ AND $\overrightarrow{V_2} \cdot \overrightarrow{V_3} < 0$ **then**
        $chkColl1 \leftarrow chkCollVeh(\overrightarrow{V_h}(t),\ Veh\_info)$
        $chkColl2 \leftarrow chkCollVeh(\overrightarrow{V_h}(t + tstep),\ Veh\_info)$
      end if
    end if
    if $chkColl1 =$ Collision OR $chkColl2 =$ Collision **then**
      $chkColl \leftarrow$ Collision
      exit
    end if
  end if
end for

Figure 5b shows that the trajectories can cover the space required using rectangles representing a vehicle to check for interference with obstacles when the vehicle changes the direction it is turning, after applying Algorithm 2 to the scenario in Fig. 5a. The black boxes in Fig. 5b are included when checking for interference with obstacles. The final collision-checking algorithm is given in Algorithm 3, where $getVect(\cdot)$ computes the tangential and normal unit vectors of the Bezier curve using Eq. 5 and $chkCollTraj(\cdot)$

Algorithm 3 Final collision-checking algorithm

Procedure $chkColl = ObstacleFree(B,\ Veh\_info)$
$(\overrightarrow{V_h},\ \overrightarrow{V_r}) \leftarrow getVect(B,\ Veh\_info)$ using Eq. 5
$(P_f,\ P_r) \leftarrow refTraj(B,\ \overrightarrow{V_h},\ \overrightarrow{V_r},\ Veh\_info)$
$chkColl \leftarrow chkCollTraj(P_f,\ P_r)$
if $chkColl =$ Collision-free **then**
  $chkColl \leftarrow chkCollTrans(\overrightarrow{V_h},\ \overrightarrow{V_r},\ Veh\_info)$
else
  $chkColl \leftarrow$ Collision
end if

checks for interference with obstacles using the trajectories of the vehicle computed in Algorithm 1. Therefore, as shown in Algorithm 1, the time complexity is linear when computing the dominant trajectories, i.e., $O(n)$, where n is the step of the parameter $t \in [0,\ 1]$ for the Bezier curve. Thus, the time complexity of our collision-checking algorithm is $O(n \times m \times k)$, where m and k are the numbers of trajectories generated by a moving vehicle and obstacles, respectively. This final collision-checking algorithm is employed in Section 5 to plan paths using the spline-based RRT*.

The results shown in Fig. 6 include transitions: (i) from a straight line to a right turn, (ii) from a right turn to a straight line, (iii) from a straight line to a right turn, (iv) from a right turn to a left turn, and (v) from a left turn to a right turn. As shown in Fig. 6, the trajectories of the front outer corner and rear inner axle of the vehicle, and the rectangles of the vehicle during transitions, can cover the space required by the vehicle when moving along straight and curved lines of variable curvature.

5 Shape-aware Spline-based RRT* for Car-like Vehicles

In this section, we provide overviews of RRT [7] and its optimal version, i.e., RRT* [33]. In succession, we extend the

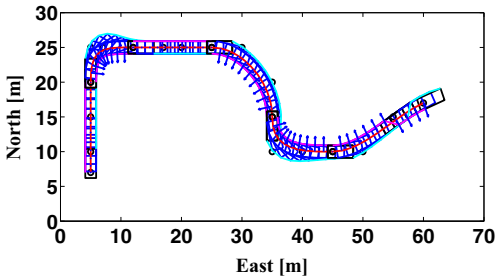


Fig. 6 Trajectories can cover the space required using rectangles representing the vehicle to check for interference with obstacles when the vehicle changes the direction it is turning

Algorithm 4 Main body of RRT or SS-RRT*

1: $V \leftarrow x_{init}$; $E \leftarrow$ NULL; $i \leftarrow 0$
2: **while** $i < N$ **do**
3: $G \leftarrow (V, E)$
4: $x_{rand} \leftarrow Sample(i)$; $i \leftarrow i + 1$
5: $(V, E) \leftarrow ExtendRRT(G, x_{rand})$ or $ExtendSSRRT^*(G, x_{rand})$
6: **end while**

Algorithm 5 $ExtendRRT(G, x)$

1: $V' \leftarrow V$; $E' \leftarrow E$
2: $x_{nearest} \leftarrow Nearest(G, x)$
3: $(x_{new}) \leftarrow Steer(x_{nearest}, x)$
4: **if** $ObstacleFree(x_{nearest}, x_{new})$ **then**
5: $V' \leftarrow V' \cup \{x_{new}\}$
6: $E' \leftarrow E' \cup \{x_{nearest}, x_{new}\}$
7: **end if**
8: **return** $G' = (V', E')$

RRT* to our SS-RRT* using the S-RRT* [12] exploiting the aforementioned collision checking algorithm (Section 4).

5.1 Overview of RRT

For a given number of iterations (line 5 in Algorithm 4), the RRT repeats the following steps until a tree reaches a goal state from an initial state: (i) The RRT randomly generates a sample (x_{rand}) in a c-space (line 4 in Algorithm 4), and extends existing tree ($G = (V, E)$, where V is a set of nodes and E is a set of edges) using this sample. (ii) During the extension process, RRT finds the nearest node ($x_{nearest}$) to x_{rand} ($Nearest(\cdot)$, line 2 in Algorithm 5), and connects a new node (x_{new}) to the trees if the edge generated by $x_{nearest}$ and x_{new} is collision-free (lines 4 to 7 in Algorithm 5). (iii) x_{new} minimizes $||x_{new} - x_{rand}||$ while maintaining $||x_{new} - x_{nearest}|| \leq \alpha$ for some predefined $\alpha > 0$ ($Steer(\cdot)$, line 3 in Algorithm 5). In this simulation, α is 5 m and the diameter of goal region is 4 m.

5.2 Overview of RRT*

In contrast to RRT, the paths found by RRT* are asymptotically optimal because of the following two steps: (i) It connects x_{new} to a node (x_{min}) with the minimum path cost (in this study, we consider that the path cost is the path length) among the neighboring nodes (X_{near}) (*new wiring*, lines 4 to 16 in Algorithm 6). (ii) It reconnects X_{near} to x_{new} and disconnect X_{near} from parent nodes if the path cost is greater than that of the path from X_{near} to x_{new} (*rewiring*, lines 17 to 26 in Algorithm 6).

Algorithm 6 $ExtendSSRRT^*(G, x)$

1: $V' \leftarrow V$; $E' \leftarrow E$; $c \leftarrow \infty$
2: $x_{nearest} \leftarrow Nearest(G, x)$
3: $x_{new} \leftarrow Steer(x_{nearest}, x)$
4: $X_{near} \leftarrow Near(G, x_{new}, |V|)$
5: **for** *all* $x_{near} \in X_{near}$ **do**
6: $B \leftarrow GetBezierCurve(x_{near}, x_{new})$
7: **if** $ObstacleFree(B, Veh\_info)$ **then**
8: $c' \leftarrow Cost(x_{near}) + CurveLength(B)$
9: **if** $c' < c$ **then**
10: $x_{min} \leftarrow x_{near}$; $c \leftarrow c'$
11: **end if**
12: **end if**
13: **end for**
14: $V' \leftarrow V' \cup \{x_{new}\}$; $E' \leftarrow E' \cup \{(x_{min}, x_{new})\}$
15: **for** *all* $x_{near} \in X_{near} \setminus \{x_{min}\}$ **do**
16: $B \leftarrow GetBezierCurve(x_{new}, x_{near})$
17: **if** $ObstacleFree(B, Veh\_info)$ AND $Cost(x_{near}) > Cost(x_{new}) + CurveLength(B)$ **then**
18: $x_{parent} \leftarrow Parent(x_{near})$
19: $E' \leftarrow E' \setminus \{(x_{parent}, x_{near})\}$
20: $E' \leftarrow E' \cup \{(x_{new}, x_{near})\}$
21: **end if**
22: **end for**
23: **return** $G' = (V', E')$

5.3 Overview of S-RRT*

In the SS-RRT*, we apply our collision-checking algorithm ($ObstacleFree(\cdot)$ in Algorithm 3) to $ExtendSSRRT^*(\cdot)$ (Algorithm 6) exploiting the S-RRT* [12], where it uses two concatenated cubic Bezier curves for a motion primitive. For the S-RRT*, the functions used in Algorithm 6 are defined as follows. $Nearest(\cdot)$ and $Steer(\cdot)$ have the same function as in RRT. $Near(\cdot)$ returns a set of nodes (X_{near}) that lie within a certain range of a given node (x_{new}) within the tree, and $GetBezierCurve(\cdot)$ obtains two concatenated cubic Bezier curves from two given nodes. $CurveLength(\cdot)$ computes the length of a given motion primitive, and $Cost(\cdot)$ computes the path length from the start node to a given node. $Parent(\cdot)$ returns the parent node of a given node. To generate a path from x_{near} to x_{new} (new wiring) or from x_{new} to x_{near} (rewiring), as shown in Fig. 7, the S-RRT* assigns inner control points (P_0–P_3 in Eq. 4) using a relation among two concatenated cubic Bezier curves to satisfy the topological property of nonholonomic systems (see [11] for details). For new wiring processes, x_{int} is selected to make $||x_{near} - x_{int}||$ the same as $||x_{new} - x_{int}||$, whereas for rewiring processes, x_{int} is selected along the heading of x_{new}, as determined in the last extension. In particular, collisions along the paths $\overline{x_{near} P_{0,1}}$ and $\overline{x_{new} P_{3,2}}$ are checked using rectangles in straight lines.

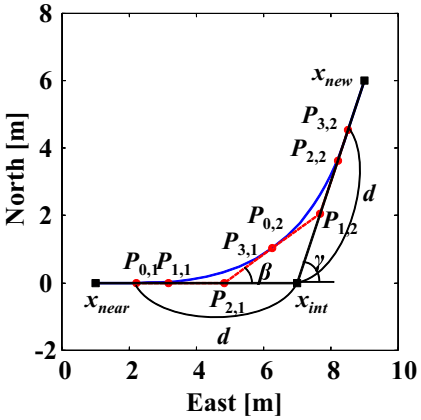


Fig. 7 A piecewise continuous path can be generated using two concatenated cubic Bezier curves when the maximum curvature and γ are specified [11]

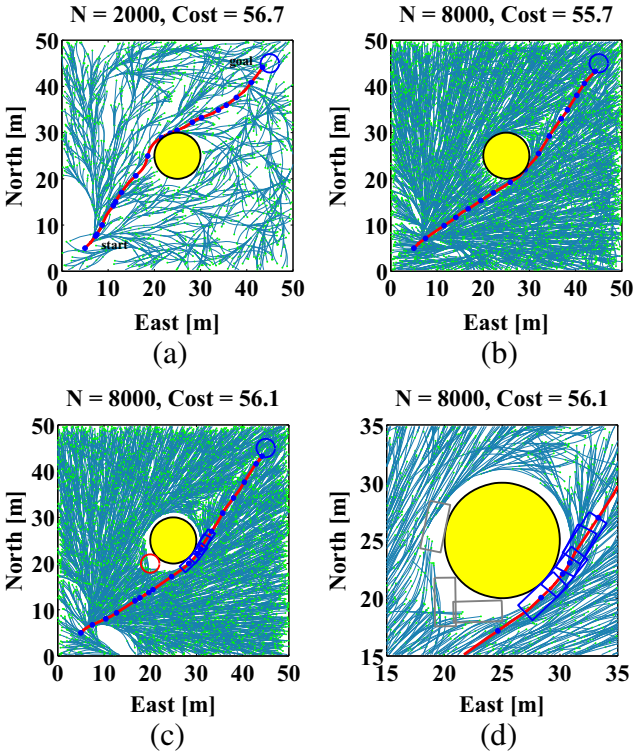


Fig. 8 Comparison between spline-based RRT* (S-RRT*, (**a**) to (**b**)) and our shape-aware spline-based RRT* (SS-RRT*, (**c**) to (**d**)) at various iterations (N is the number of expanded nodes). These figures show the paths found (red lines) and the trees expanded (green lines) using two methods. Unlike S-RRT*, the proposed SS-RRT* generates the tree outside the boundary of an obstacle (yellow filled circles), but not on its boundary. In particular, the space not expanded by the tree in the region indicated by a red circle (**c**) is larger than other spaces because of the overhang at the front of the vehicle. The blue circles and rectangles represent the goal region and the vehicle shapes, respectively. **d** Enlargement of the region indicated in the red circle. Our SS-RRT* can generate paths that respect the shape and turning space of a vehicle (a_f = 3.4 m, a_r = 0.8 m, a_w = 1.8 m, r_{min} = 4.8 m)

Figure 8 shows the results for the paths and trees expanded using S-RRT* and our SS-RRT*. Both can be employed to generate ϵ-reachable paths using $Steer(\cdot)$ and $GetBezierCurve(\cdot)$, and to obtain ϵ-collision-free paths with each collision-checking method (Fig. 8), i.e., our path planners are asymptotically optimal [10, 34]. Unlike S-RRT*, SS-RRT* generates trees outside the boundary of an obstacle (yellow filled circle), but not on its boundary. In particular, the space not expanded by trees in the region indicated by the red circle is larger than other spaces because of the front overhang of the vehicle. Figure 8d shows an enlargement of the region indicated by the red circle in Fig. 8c. Thus, our SS-RRT* can generate paths that are checked for interference against obstacles using the shape and turning space of a vehicle.

6 Results

Thus far, we have explained our collision checking algorithm and its application to the spline-based RRT*, and demonstrated the effectiveness of SS-RRT* using our collision-checking algorithm through the simple example. In this section, we show the efficiency of our SS-RRT* in complex environments, followed by experimental results obtained by using an autonomous vehicle. The simulations and experiments are conducted using MATLAB, and our autonomous vehicle is equipped with controllers and a path planner, which are integrated using LabVIEW on an i7 computer with a 3.4 GHz CPU and 3 GB DRAM.

6.1 Simulation Results

This section compares the continuity and efficiency of our collision-checking algorithm with other methods, and provides additional simulation examples for complex environments to show that SS-RRT* computes the required space of the moving vehicle more continuously and efficiently than S-RRT* and RS-RRT* (using the collision-checking algorithm with rectangle-sampling method).

To show that our method more continuously and efficiently checks for interference with obstacles than S-RRT* and RS-RRT* for a moving vehicle, respectively, we compare SS-RRT*, S-RRT* and RS-RRT* in a complex environment with a narrow passage (width = 3.7 m). This minimum width is selected so that new samples can be generated in narrow passages. Although the sampling density can be increased in such spaces by employing an adaptive strategy, this is beyond the scope of the current paper. The results are shown in Fig. 9. We assume that the obstacles are points (acquired when using laser scanners), and add circles corresponding to the width of the vehicle and the

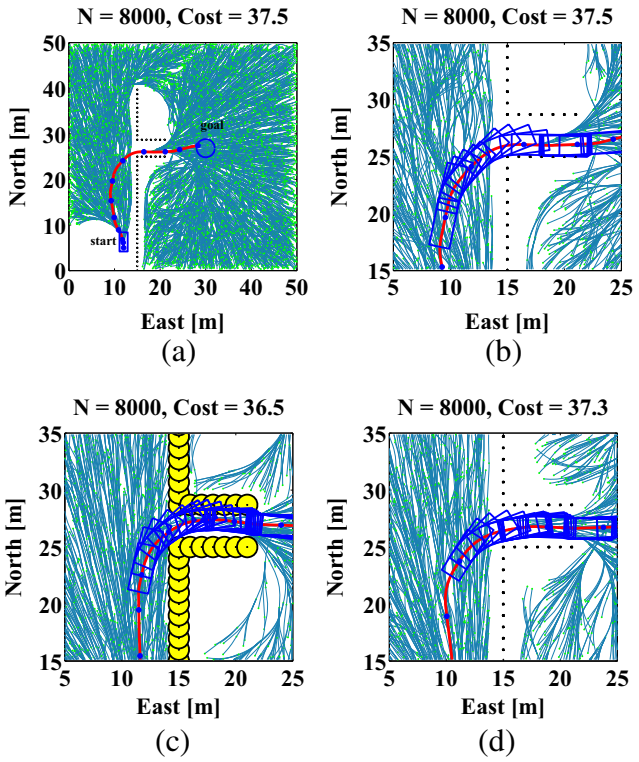


Fig. 9 Comparison between SS-RRT* (our method) (**a**), (**b**), S-RRT* (**c**), and RS-RRT* (**d**). These figures show the paths found (red lines) and the trees expanded (green lines) using the three methods. Our method and RS-RRT* found a collision-free path with a detour near the entrance to the narrow passage, whereas S-RRT* did not find a collision-free path, i.e. the vehicle collides with the yellow circle (in (**c**), the yellow circles mean obstacles dilated by considering the width of vehicle). **b** Enlargement of (**a**) (a_f = 3.4 m, a_r = 0.8 m, a_w = 1.8 m, r_{min} = 4.8 m, width of narrow passage = 3.7 m)

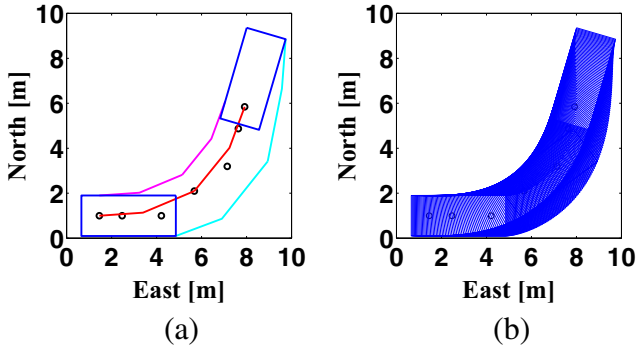


Fig. 10 Trajectories and overlapped rectangles for our collision-checking algorithm (**a**) and the rectangle-sampling method [27] for the comparison shown in Fig. 9b. The distance between front outer corners of two adjacent rectangles for the rectangle-sampling method is about 0.1 m. This value was selected to check for interference against obstacles with 0.1 m diameter for avoiding the limitation of discretization as shown in Fig. 1b and the low sampling density in the narrow passage shown in Fig. 9. Black circles are the control points for two concatenated Bezier curves (a_f = 3.4 m, a_r = 0.8 m, a_w = 1.8 m, r_{min} = 4.8 m)

distance between front outer corners of two adjacent rectangles to simulate S-RRT* and RS-RRT*, respectively. However, we do not preprocess the obstacles when using SS-RRT*, i.e., we assume that the obstacles are points to show the continuity of our collision-checking algorithm. As shown in Fig. 9, the paths found by the proposed and rectangle-sampling methods include a detour near the entrance to the narrow passage, because SS-RRT* and RS-RRT* compute the turning space required by the overhang at the front of the vehicle. In contrast, the path found by S-RRT* caused this overhang to collide with obstacles dilated by considering the width of vehicle (Fig. 9d) when the vehicle entered the narrow passage. Thus, our method computes the required space of the moving vehicle while checking for interference with obstacles as continuously as RS-RRT* using densely sampled rectangles (Fig. 10b). In the below, we explain the efficiency of our method while comparing RS-RRT*.

Figure 10 shows the trajectories and overlapped rectangles given by our collision-checking method and the rectangle-sampling method [27] for the comparison shown in Fig. 9. In this comparison, our method uses two concatenated Bezier curves consisting of four lines as a motion primitive, and the rectangle-sampling method uses rectangles sampled every specified distance. In addition, obstacles are dilated for guaranteeing a clearance between the vehicle and obstacles, i.e., we select their diameter to be 0.1 m. Therefore, the rectangles are sampled with a distance between adjacent outside front corners of about 0.1 m for avoiding the limitation of discretization as shown in Fig. 1b. Also, 0.1 m diameter is adaptable for avoiding the low sampling density in the narrow passage shown in Fig. 9.

Figure 11a illustrates the computation time of our collision-checking algorithm depending on the number of obstacles and lines consisting of Bezier curves. To do this, we apply it to the scenario in Fig. 10. This figure also shows the performance of the other baselines: (i) adding a circle corresponding to the width of the vehicle to obstacles in a point cloud, (ii) using all (six) trajectories of the corners and rear axle of the vehicle without applying Algorithm 1 and 2, (iii) and rectangle-sampling [27]. We then execute each method over the number of obstacles. The results show that our collision-checking algorithm can reduce the computation time by about 28% over the methods adding a circle and about 55% and 78% over the methods using all trajectories and rectangle-sampling, respectively. In addition, Fig. 11b shows the computational time required by Algorithm 1 and 2 without $chkCollVeh(\cdot)$ relative to the number of lines for two concatenated Bezier curves. As shown in Fig. 11b, the computational time required to obtain dominant trajectories increases linearly relative to the number of lines for two concatenated Bezier curves. In particular, for Bezier

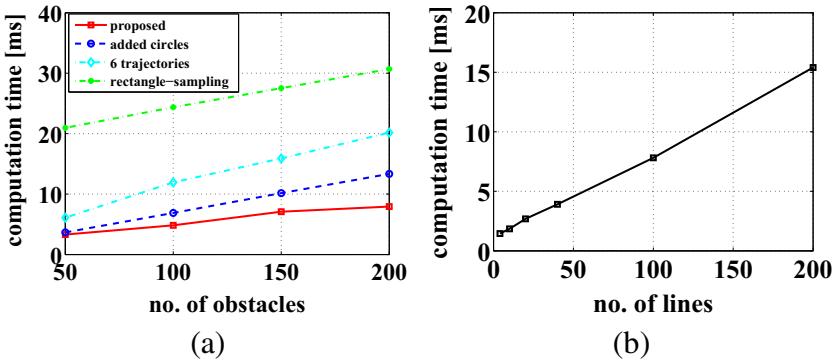


Fig. 11 Computation times (average over 10 trials) of (**a**) our collision-checking algorithm and the methods of adding a circle corresponding to the width of the vehicle to obstacles, using all (six) trajectories of the corners and rear axle of the vehicle without applying Algorithm 1 and 2, and rectangle-sampling [27] over the number of obstacles, and (**b**) Algorithm 1 and 2 without $chkCollVeh(\cdot)$ relative to the number of lines for two concatenated Bezier curves when the number of obstacles is 200. For this comparison, we apply the collision-checking algorithms to the scenario in Fig. 10

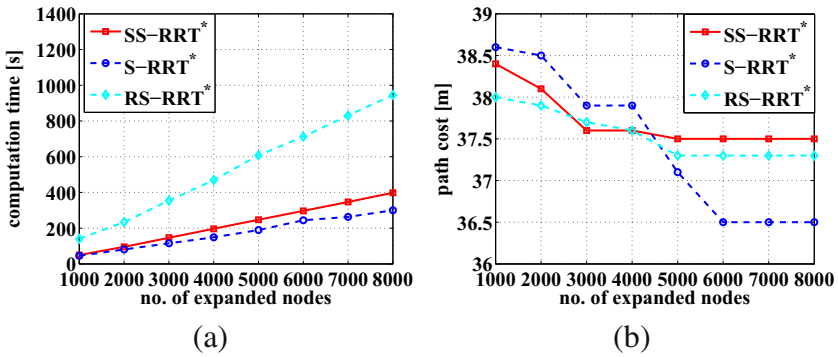


Fig. 12 **a** Computational time and **b** path costs for SS-RRT*, S-RRT* and RS-RRT* with different numbers of expanded nodes for the results shown in Fig. 9

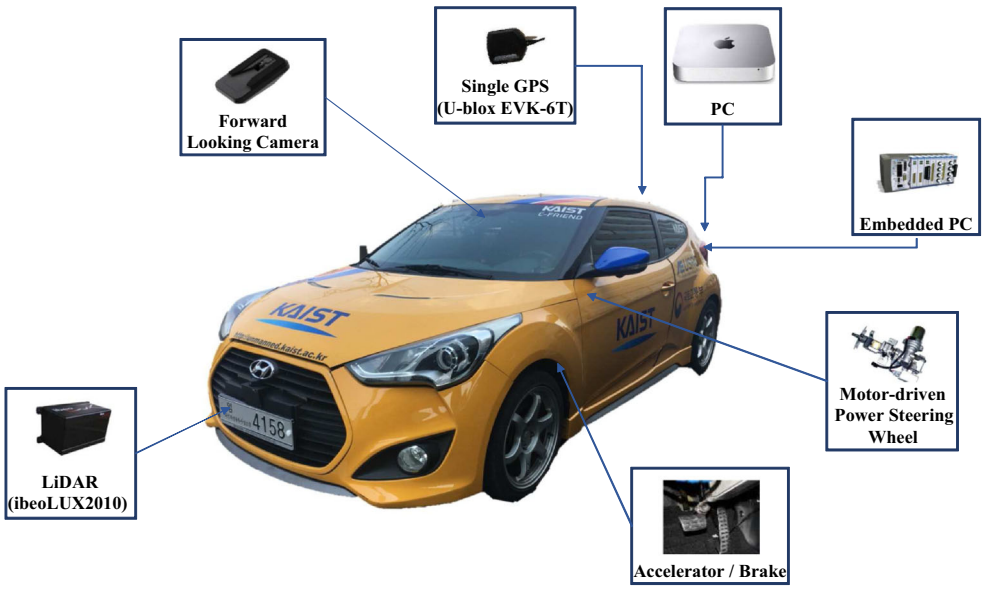


Fig. 13 KAIST autonomous ground vehicle

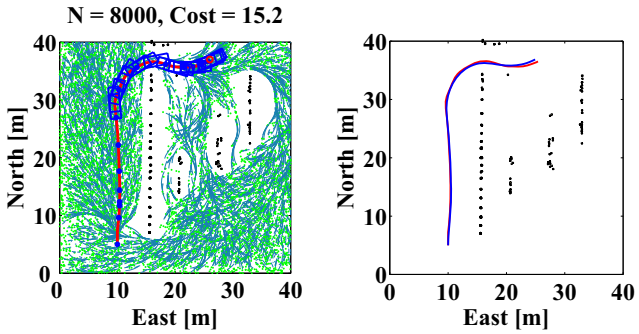


Fig. 14 **a** This figure shows the result for the path and trees expanded using our SS-RRT* after building an obstacle map from obstacles obtained by the LiDAR (ibeo LUX2010) equipped in our autonomous vehicle. The blue rectangles and the red circle represent the vehicle shapes and the goal region (radius = 0.5 m), respectively. **b** Comparison between the path found by our SS-RRT* (red lines) and the position of the vehicle measured by a GPS with an extended Kalman filter (blue lines). (a_f = 3.5 m, a_r = 0.75 m, a_w = 1.805 m, r_{min} = 4.2 m, width of narrow passage = 5 m)

curves that comprise four lines, the proportion of the computational time required to obtain the dominant trajectories (ca 1.5 ms) is approximately 19% of the total computational time when the number of obstacles is 200. This means that our collision-checking algorithm can increase the continuity and efficacy but with only a small increase in the computational time for obtaining the dominant trajectories .

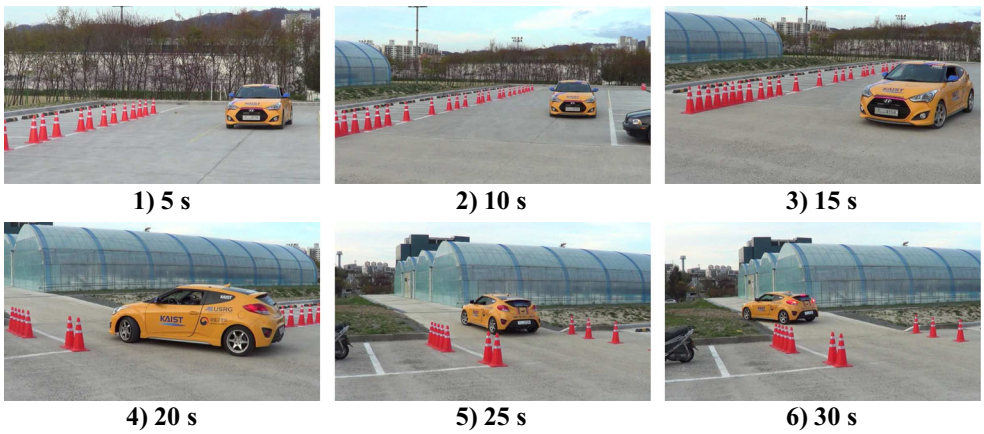
We compared the computational time of our path planning method based on the results shown in Fig. 9 (Fig. 12). The computational time required by SS-RRT* is greater than that required by S-RRT* as the number of expanded nodes increases. This is mainly because a space to generate new nodes in the narrow passage is smaller in SS-RRT* than S-RRT* due to the overhang and the width of vehicle as the number of expanded nodes increases. However, it is similar when the number of expanded nodes is 2000, where the paths almost converge on the asymptotic optimal paths (Fig. 12b) whereas S-RRT* collides with dilated obstacles (Fig. 9d). As shown in Fig. 12b, the quality (path cost) of path found by SS-RRT* is similar to that found by RS-RRT* and the paths found by SS-RRT* and RS-RRT* converge on the optimal paths when the number of expanded nodes is 5000. In other words, the paths probabilistically converge on the optimal as the number of expanded nodes increases because RRT* has the rewiring process to connect a new node to its neighboring node if the path cost of the new node is smaller than that of current tree. As shown in Fig. 12a, however, our collision-checking method is more efficient (shorter computation time) than RS-RRT* while using collision-checking algorithm taking into account the overhang and the width of vehicle like RS-RRT*. In other words, our collision-checking algorithm can more continuously check for collisions while using longer sampling distance than rectangle-sampling method, in order to avoid the limitation of discretization. In addition, unlike our collision-checking method, the number of sampled rectangles for rectangle-sampling method (used for RS-RRT*) more depends on the size of circle added to obstacles, which is determined for avoiding the limitation of discretization and the low sampling density in the narrow passage as illustrated in Fig. 10b.

In summary, the proposed SS-RRT* finds collision-free paths by checking for interference against obstacles while accurately computing the required space of moving car-like vehicles than S-RRT*. Furthermore, our method is more continuous and efficient than the rectangle-sampling method.

6.2 Experimental Results

We performed experiments on a ground vehicle (Fig. 13). The ground vehicle was equipped with a gasoline engine of 1591 cm^3, a six-speed automatic transmission, a front-wheel drive, and an anti-lock braking system (ABS). Its wheelbase and total length were 2.65 m and 4.25 m long,

Fig. 15 An image sequence of passing through the narrow passage shown in Fig. 9



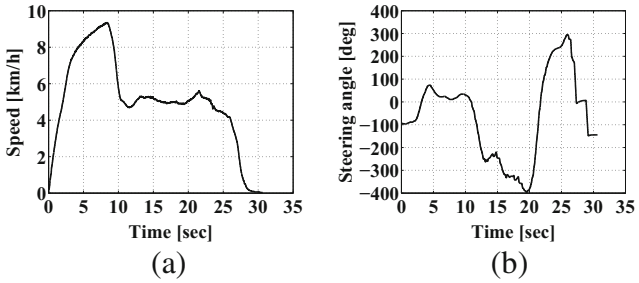


Fig. 16 **a** Speed and **b** steering angle of the vehicle over time, which were measured by in-vehicle sensors when Fig. 14 was tested

respectively. In these experiments, we acquired the position, steering angle, and speed of the vehicle using a single global positioning system (GPS) with an extended Kalman filter and sensors installed by the car manufacturer (Hyundai Motors Corp.). In addition, the vehicle was installed with LiDAR (ibeo LUX2010) in its front bumper in order to obtain the information of obstacles. Our autonomous vehicle has already been operated in the 2014 Hyundai Motor Company's Autonomous Car Competition [35].

We tested our method in a real environment with a narrow passage in a correspondent manner shown in Fig. 9. Figures 14 and 15 show the result for the paths and trees expanded using our SS-RRT* after building an obstacle map from obstacles obtained by the LiDAR equipped in our autonomous vehicle and an image sequence of passing through the narrow passage (width = 5 m), respectively. In order to protect our autonomous vehicle from colliding with obstacles, we added the clearance (1 m in each side) to the width of our autonomous vehicle when operating our SS-RRT*.

It shows that our SS-RRT* can generate the collision-free path while piecewise continuously computing the required space of moving car-like vehicles. Further, it shows that our SS-RRT* satisfies the topological property of nonholonomic systems by using two concatenated cubic Bezier curves as stated in Section 5.3. It can be demonstrated from the result that the vehicle can follow the generated path without any pause of the vehicle and jerk of its steering angle as shown in Fig. 16.

7 Conclusion

We have developed a collision-checking method that determines a collision-free path for car-like vehicles. Our approach computes the dominant trajectories generated from the front outer corner and rear inner axle of the vehicle, and uses partially overlapped rectangles. Because the dominant trajectories used to check for interference with obstacles are generated from the front outer corner and rear inner axle, they cannot cover the overall regions occupied by the vehicle when changing its turning direction, or at the start and end positions. To address this problem, we detect collisions by overlapping the rectangles during transitions. Therefore, our method can check for interference with obstacles on paths with variable curvatures in a piecewise continuous and efficient manner.

To demonstrate the effectiveness of our collision-checking algorithm for car-like vehicles, we applied the algorithm to spline-based RRT* (SS-RRT*). The simulation and experiment results showed that SS-RRT* can plan a collision-free path while satisfying the topological property of nonholonomic systems and computing the required space of a moving vehicle.

References

1. Likhachev, M., Ferguson, D.: Planning long dynamically feasible maneuvers for autonomous vehicles. Int. J. Robot. Res. **28**(8), 933–945 (2009)
2. Dolgov, D., Thrun, S., Montemerlo, M., Diebel, J.: Path planning for autonomous vehicles in unknown semi-structured environments. Int. J. Robot. Res. **29**(5), 485–501 (2010)
3. Yoon, S., Yoon, S.E., Lee, U., Shim, D.H.: Recursive path planning using reduced states for car-like vehicles on grid maps. IEEE Trans. Intell. Transp. Syst. **16**(5), 2797–2813 (2015)
4. Koenig, S., Likhachev, M., Furcy, D.: Lifelong planning A*. Artif. Intell. **155**, 93–146 (2004)
5. Yoon, S., Shim, D.H.: SLPA,*: Shape-aware lifelong planning A* for differential wheeled vehicles. IEEE Trans. Intell. Transp. Syst. **16**(2), 730–740 (2015)
6. Svestka, P., Overmars, M.H.: Coordinated motion planning for multiple car-like robots using probabilistic roadmaps. In: ICRA, pp 1631–1636 (1995)
7. LaValle, S.M., James, J., Kuffner, J.: Randomized kinodynamic planning. Int. J. Robot. Res. **20**(5), 378–400 (2001)
8. Kuwata, Y., Teo, J., Fiore, G., Karaman, S., Frazzoli, E., How, J.P.: Real-time motion planning with applications to autonomous urban driving. IEEE Trans. Control Syst. Technol. **17**(5), 1105–1118 (2009)
9. Goretkin, G., Perez, A., Platt, R., Konidaris, G.: Optimal sampling-based planning for linear-quadratic kinodynamic systems. In: ICRA, pp. 2429–2436 (2013)
10. Karaman, S., Frazzoli, E.: Optimal kinodynamic motion planning using incremental sampling-based methods. In: IEEE Conference on Decision and Control, pp. 7681–7687 (2010)
11. Yang, K., Moon, S., Yoo, S., Kang, J., Doh, N.L., Kim, H.B., Joo, S.: Spline-based RRT path planner for non-holonomic robots. J. Intell Robot Syst. 1–20 (2013)
12. Yang, K., Gan, S.K., Huh, J., Joo, S.: Optimal spline-based RRT path planning using probabilistic map. In: International Conference on Control, Automation and Systems (ICCAS), pp. 643–646 (2014)
13. Dubins, L.E.: On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents. Am. J. Math. **79**(3), 497–516 (1957)
14. Connors, J., Elkaim, G.: Analysis of a spline based, obstacle avoiding path planning algorithm. In: IEEE Vehicular Technology Conference, pp. 2565–2569 (2007)
15. Perez, A., Platt, R., Konidaris, G., Kaelbling, L., Lozano-Perez, T.: LQR-RRT*: Optimal sampling-based motion planning with automatically derived extension heuristics. In: ICRA, pp. 2537–2542 (2012)

16. Webb, D.J., van den Berg, J.: Kinodynamic RRT*: Asymptotically optimal motion planning for robots with linear dynamics. In: ICRA, pp. 5054–5061 (2013)
17. Palmieri, L., Arras, K.O.: A novel RRT extend function for efficient and smooth mobile robot motion planning. In: IROS, pp. 205–211 (2014)
18. Yershova, A., Jaillet, L., Siméon, T., LaValle, S.M.: Dynamic-domain RRTs: Efficient exploration by controlling the sampling domain. In: ICRA, pp. 3856–3861 (2005)
19. Lee, J., Kwon, O., Zhang, L., Yoon, S.E.: A selective retraction-based RRT planner for various environments. IEEE Trans. Robot. **30**(4), 1002–1011 (2014)
20. Pruski, A., Rohmer, S.: Robust path planning for non-holonomic robots. J. Intell. Robot. Syst. **18**(4), 329–350 (1997)
21. Varadhan, G., Sriram, S.K.T.V.N., Manocha, D.: A simple algorithm for complete motion planning of translating polyhedral robots. Int. J. Robot. Res. **24**(11), 983–995 (2005)
22. Wise, K.D., Bowyer, A.: A survey of global configuration-space mapping techniques for a single robot in a static environment. Int. J. Robot. Res. **19**(8), 762–779 (2000)
23. Minguez, J., Montano, L.: Extending collision avoidance methods to consider the vehicle shape, kinematics, and dynamics of a mobile robot. IEEE Trans. Robot. **25**(2), 367–381 (2009)
24. Yang, K., Sukkarieh, S.: An analytical continuous-curvature path-smoothing algorithm. IEEE Trans. Robot. **26**(3), 561–568 (2010)
25. Bicchi, A., Casalino, G., Santilli, C.: Planning shortest bounded-curvature paths for a class of nonholonomic vehicles among obstacles. J. Intell. Robot. Syst. **16**, 387–405 (1996)
26. Geraerts, R., Overmars, M.H.: The corridor map method: a general framework for real-time high-quality path planning. Comput. Anim. Virt. Worlds **18**, 107–119 (2007)
27. Maekawa, T., Noda, T., Tamura, S., Ozaki, T., Ichiro Machida, K.: Curvature continuous path generation for autonomous vehicle using B-spline curves. Comput. Aided Des. **42**, 350–359 (2010)
28. Gómez-Bravo, F., Cuesta, F., Ollero, A.: Parallel and diagonal parking in nonholonomic autonomous vehicles. Eng. Appl. Artif. Intel. **14**(4), 419–434 (2001)
29. Jazar, R.N.: Vehicle Dynamics: Theory and Applications. Springer (2008)
30. Luca, A.D., Oriolo, G., Samson, C., Laumond, J.P.: Feedback control of a nonholonomic car-like robot. In:Robot motion planning and control, pp. 171–253. Springer (1998)
31. Anand, V.B.: Computer Graphics and Geometric Modeling for Engineers, 1st edn. Wiley (1993)
32. Zill, D.G., Cullen, M.R.: Advanced Engineering Mathematics PWS. Publishing Company (1992)
33. Karaman, S., Frazzoli, E.: Sampling-based algorithms for optimal motion planning. Int. J. Robot. Res. **30**(7), 846–894 (2011)
34. Karaman, S., Frazzoli, E.: Sampling-based optimal motion planning for non-holonomic dynamical systems. In: ICRA, pp. 5041–5047 (2013)
35. Lee, U., Jung, J., Shin, S., Jeong, Y., Park, K., Shim, D.H., So Kwon, I.: EureCar turbo: A self-driving car that can handle adverse weather conditions. In: The International Conference of Intelligent Robots and Systems (IROS), pp. 2301–2306 (2016)

Sangyol Yoon received the B.S. in mechanical engineering from Hongik University, Seoul, Korea, in 1999, his M.S. in mechatronics from Gwangju Institute of Science and Technology, Gwangju, Korea, in 2001, and his Ph.D. from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea, in 2014. From 2003 to 2007, he was with Samsung Electronics Company Ltd., Suwon, Korea, where he was engaged in the development of three-axis optical pickup actuators for optical disk drives. From 2007 to 2009 he was with Hyundai Mobis, Yongin, Korea, where he was involved in pre-crash and advanced external airbag systems development. Since 2014, he has been with LG Electronics Inc., Seoul, Korea. His research interests include vehicle motion planning and control, and autonomous driving cars.

Dasol Lee received the B.S. in aerospace engineering from Korea Aerospace University, Goyang, Korea in 2013 and the M.S. in aerospace engineering from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea in 2015, respectively. He is now a graduate researcher, Ph. D. candidate in Unmanned System Research Group, KAIST. His interests include path planning, guidance, and control for unmanned aerial vehicles (UAVs).

Jiwon Jung received the B.S. degree in aerospace engineering from Sejong University, Seoul, Korea, in 2012 and the M.S. degree in aerospace engineering from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea, in 2014. He is currently working toward the Ph.D. degree in aerospace engineering with KAIST. His research interests include path planning and navigation for autonomous systems.

David Hyunchul Shim (Member, IEEE) received the B.S. and M.S. degrees in mechanical design and production engineering from Seoul National University, Seoul, Korea, in 1991 and 1993, respectively, and the Ph.D. degree in mechanical engineering from the University of California Berkeley, Berkeley, in 2000. From 1993 to 1994, he was with Hyundai Motor Company, Korea, as Transmission Design Engineer. From 2001 to 2005, he was with Maxtor Corporation, Milpitas, CA as Staff Engineer for advanced servo control in hard disk drives. From 2005 to 2007, he was with the University of California Berkeley as Principal Engineer, in charge of Berkeley Aerobot Team. In 2007, he joined the Department of Aerospace Engineering, Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Korea, as an Assistant Professor. He is now tenured Associate Professor and Director of Center of Field Robotics in KAIST. His research interests include control theory, unmanned aerial vehicles, self-driving cars, and field robotics.