

## Chapter 9

# Nonholonomic Planning

This chapter presents three alternative path planning methods. For each method, it is assumed that a state transition equation or incremental simulator has been defined over a state space. The state could represent configuration or both configuration and velocity.

### 9.1 RRT-Based Methods

The RRT planning method can be easily adapted to the case of nonholonomic planning. All references to configurations are replaced by references to states; this is merely a change of names. The only important difference between holonomic planning and nonholonomic planning with an RRT occurs in the EXTEND procedure. For holonomic planning, the function NEW\_CONFIG generated a configuration that lies on the line segment that connects  $q$  to  $q_{near}$ . For nonholonomic planning, motions must be generated by applying inputs. The NEW\_CONFIG function is replaced by NEW\_STATE, which attempts to apply all of the inputs in  $U$ , and selects the input that generates an  $x_{new}$  that is closest to  $x_{near}$  with respect to the metric  $\rho$ . If  $U$  is infinite, then it can be approximated with a finite set of inputs.

### 9.2 Forward Dynamic Programming

The forward dynamic programming (FDP) method is similar to an RRT in that it grows a tree from  $x_{init}$ . The key difference is that FDP uses dynamic programming to decide how to incrementally expand the tree, as opposed to nearest-neighbors of random samples. FDP performs a systematic exploration over fine-resolution grid that is placed over the state space. This limits its applicability to low-dimensional state spaces (up to 3 or 4 dimensions).

The configuration space,  $X$ , is divided into a rectangular grid (typically there are a hundred grid points per axis). Each element of the grid is called a *cell*, which designates a rectangular subset of  $X$ . One of three different labels can be applied to each cell:

- OBST: The cell contains points in  $X_{obs}$ .
- FREE: The cell has not yet been visited by the algorithm, and it lies entirely in  $X_{free}$ .
- VISITED: The cell has been visited, and it lies entirely in  $X_{free}$ .

Initially, all cells are labeled either FREE or OBST by using an collision detection algorithm.

Let  $Q$  represent a priority queue in which the elements are configurations, sorted in increasing order according to  $L$ , which represents the cost accumulated along the path constructed so far from  $x_{init}$  to  $x$ . This cost can be assigned in many different ways. It could simply represent the time (number of  $\Delta t$  steps), or could count the number of times a car changes directions.

The algorithm proceeds as follows:

---

```

FORWARD_DYNAMIC_PROGRAMMING( $x_{init}, x_{goal}$ )
1   $Q.insert(x_{init}, L)$ ;
2   $G.init(x_{init})$ ;
3  while  $Q \neq \emptyset$  and  $FREE(x_{goal})$ 
4       $x_{cur} \rightarrow Q.pop()$ ;
5      for each  $x \in NBHD(x_{cur})$ 
6          if  $FREE(x)$ 
7               $Q.insert(x, L)$ ;
8               $G.add\_vertex(x)$ ;
9               $G.add\_edge(x_{cur}, x)$ ;
10         Label cell that contains  $x$  as VISITED;
11  Return  $G$ ;

```

---

The algorithm iteratively grows a tree,  $G$ , which is rooted at  $x_{init}$ . The NBHD function tries the possible inputs, and returns a set of configurations that can be reached in time  $\Delta t$ . For each of these configurations, if the cell that contains it is FREE, then  $G$  is extended. At any given time, there is at most one vertex per cell. The algorithm terminates when the cell that contains the goal has been reached.

### 9.3 Steering-Based Methods

A common theme for many planning approaches is to divide the problem into two phases. In the first phase, a holonomic planning method is used by producing a collision-free path that ignores the nonholonomic constraints. In the second phase, an iterative method attempts to replace portions of the holonomic path with portions that satisfy the nonholonomic constraints, yet still avoid obstacles. In general, this will lead to an incomplete algorithm because there is no guarantee that the original path provides a correct starting point for obtaining a nonholonomic solution. However, it typically leads to a fast planning algorithm.

In this section, we describe this approach for the case of a car-like robot. Assume that a fast holonomic planning method has been selected for the first phase. Suppose that a path,  $\tau : [0, 1] \rightarrow \mathcal{C}_{free}$  has been computed. The path can be iteratively improved as follows. Randomly select two real numbers  $\alpha_1 \in [0, 1]$  and  $\alpha_2 \in [0, 1]$ . Assuming  $\alpha_2 > \alpha_1$  (if not, then swap them), attempt to replace the portion of  $\tau$  from  $\tau(\alpha_1)$  to  $\tau(\alpha_2)$  with a path segment that satisfies the nonholonomic constraints. This implies that  $\tau$  is broken into three segments,  $\tau_1 : [0, \alpha_1] \rightarrow \mathcal{C}_{free}$ ,  $\tau_2 : [\alpha_1, \alpha_2] \rightarrow \mathcal{C}_{free}$ , and  $\tau_3 : [\alpha_2, 1] \rightarrow \mathcal{C}_{free}$ . Note that  $\tau_1(\alpha_1) = \tau_2(\alpha_1)$  and  $\tau_2(\alpha_2) = \tau_3(\alpha_2)$ . The portions  $\tau_1$  and  $\tau_3$  remain fixed, but  $\tau_2$  is replaced with a new path,  $\tau' : [\alpha_1, \alpha_2] \rightarrow \mathcal{C}_{free}$ , that satisfies the nonholonomic constraints. Note that  $\tau'$  must also avoid collisions,  $\tau'(\alpha_1) = \tau_1(\alpha_1)$ , and  $\tau'(\alpha_2) = \tau_3(\alpha_2)$ . This procedure can be iterated multiple times until eventually, the original path is completely transformed into a nonholonomic path. Note that  $\alpha_1 = 0$  and  $\alpha_2 = 1$  must each have nonzero probability of being chosen in each iteration. In many iterations, the path substitution will fail; in this case, the previous path is retained.

To make this and related approaches succeed, a fast technique is needed that constructs a nonholonomic path between any two configurations. Although this might appear as difficult as the original nonholonomic planning problem, it is assumed that the obstacles are ignored. In general, this is referred to as the *steering problem*, which has received a considerable amount of attention in recent years, particularly for car-like robots that pull trailers. For the case of a simple car-like robot with a limited steering angle, there are some analytical solutions to the problem of finding the shortest path between two configurations. In 1957, Dubins showed that for a car that can only go forward, the optimal path will take one of the six following forms:

$$\{LRL, LSL, LSR, RLR, RSR, RSL\}.$$

Each sequence of labels indicates the type of path. For example, “LRL” indicates a path that consists of a sharp left turn, immediately followed by a sharp right turn, immediately followed by a sharp left turn.

Above, “S” denotes a straight segment. For a given pair of configurations, one can simply attempt to connect them using all six path types. The one with the shortest path length among the six choices is known to be the minimum-length path out of all possible paths. This path provides a nice substitution for  $\tau_2$ , as described above.

For the case of a car-like robot that can move forward or backwards, Reeds and Shepp showed in 1990 that the optimal path between two configurations will take one of 48 different forms. Although this situation is more complicated, the same general strategy can be applied as for the case of a forward-only car.