★ ~~Sensing Uncertainty~~

Anytime Motion Planning using the RRT*

(by Sertac Karaman et al.)

Abstract

→ This paper describes an anytime algorithm based on RRT* which finds an initial feasible solution quickly, but almost surely converges to an optimal solution.

★ Introduction

⇒ The motion planning problem is to find a dynamically feasible trajectory that takes the robot from an initial state to a goal state while avoiding collision with obstacles.

⇒ An anytime motion planning algorithm should two properties:

- → A form completeness
- → Asymptotic optimality.

⇒ A system based on anytime planning overlaps two functions in time

- → execution of its current plan

↳ Computation to replan the current plan with an improved plan.

* The RRT* Algorithm

⇒ Consider a system with dynamics of the following form,

$$\boxed{\dot{x}(t) = f(x(t), u(t))}$$

where, $x(t) \in X$ {State space}
$u(t) \in U$ {Input space}

⇒ Let X_{obs} denote the obstacle region, and $X_{free} = X \setminus X_{obs}$ define the obstacle-free space.

⇒ Finally, let $X_{goal} \subset X$ denote the goal region.

⇒ <u>Motion problem:</u>

"Find a control input $u : [0,T] \to U$ that yields a feasible path $x(t) \in X_{free}$ $\forall t \in [0,T]$ from an initial state $x(0) = x_{init}$ to the goal region $x(T) \in X_{goal}$ that obeys the system dynamics."

⇒ The optimal motion planning problem impose the additional requirement that the resulting feasible path minimizes a given cost function $C(x)$.

↳ $C(x)$ maps every admissible trajectory $x : [0,T] \to X$ to a positive real number.

⇒ In solving the optimal motion planning problem, the RRT* algorithm builds and maintain a tree $\Upsilon = (V, E)$ comprised of a vertex set V of states from X_{free} connected by directed edges $E \subseteq V \times V$.

⇒ The manner in which the RRT* generates this tree closely resembles that of the standard RRT, with the addition of a few key steps that achieve optimality.

Sampling: The Sample function randomly samples a state $z_{rand} \in X_{free}$ from the obstacle-free region of the state space.

Dist:

Distance: $X \times X \to R_{\geq 0}$ returns the cost of the optimal trajectory between two states, assuming no obstical

Nearest Neighbor: Given a state $z \in X$ and the tree $T = (V, E)$, the $v =$ Nearest (T, z) function returns the nearest node in the tree in terms of the distance function.

Near-by Vertices: Given a state $z \in X$, tree $T = (V, E)$ and a number n, the $Z_{nearby} = Near(T, z, n)$ function returns the vertices in V that are near z.

↳ $Reach(z, l) = \{z' \in X \mid Dist(z, z') \leq l\}$.

↳ Chose $l(n)$ such that $Reach(z, l(n))$ contains a ball of volume $\gamma((\log n)/n)^d$, where γ is a fixed number.

Collision Check: The $ObstacleFree(x)$ function checks whether a path $x: [0,T] \to X$ lies within the obstacle-free region of state space.

Steering: The $(x, u, T) = Steer(z_1, z_2)$ function solves for the control input $u: [0,T]$ that drives the system from $x(0) = z_1$ to $x(T) = z_2$ along the path $x: [0,T] \to X$.

Node Insertion: Given the current tree $T = (V, E)$, an existing state $Z_{current} \in V$, and a new state Z_{new} ~~to V and creates an edge to Zcurrent~~ the $InsertNode(Z_{current}, Z_{new}, T)$ procedure adds Z_{new} to V and creates an edge to $Z_{current}$ as its parent, which it adds to E.

↳ It assigns a $Cost(Z_{new})$ to Z_{new} equal to that of its parent, plus the cost $c(x)$ of the trajectory associated with the new edge.

Algorithm 1: $T=(V,E) \leftarrow RRT^*(z_{init})$

$T \leftarrow$ InitializeTree();
$T \leftarrow$ InsertNode(ϕ, z_{init}, T);
for $i=1$ to $i=N$ do
    $z_{rand} \leftarrow$ Sample(i);
    $z_{nearest} \leftarrow$ Nearest(T, z_{rand});
    $(x_{new}, u_{new}, T_{new}) \leftarrow$ Steer$(z_{nearest}, z_{rand})$;
    if ObstacleFree(x_{new}) then
        $Z_{near} \leftarrow$ Near$(T, z_{new}, |V|)$;
        $z_{min} \leftarrow$ ChooseParent$(Z_{near}, z_{nearest}, z_{new}, x_{new})$;
        $T \leftarrow$ InsertNode(z_{min}, z_{new}, T);
        $T \leftarrow$ ReWire$(T, Z_{near}, z_{min}, z_{new})$;
return T;

Algorithm 2: $z_{min} \leftarrow$ ChooseParent$(Z_{near}, z_{nearest}, x_{new})$

$z_{min} \leftarrow z_{nearest}$;
$c_{min} \leftarrow$ Cost$(z_{nearest}) + c(x_{new})$;
for $z_{near} \in Z_{near}$ do
    $(x', u', T') \leftarrow$ Steer(z_{near}, z_{new});
    if ObstacleFree(x') & $x'(T') = z_{new}$ then
        $c' =$ Cost$(z_{near}) + c(x')$;

```
        if c' < Cost(z_new) & c' < c_min then
            z_min ← z_near;
            c_min ← c';

return z_min;
```

Algorithm 3: $T \leftarrow \text{ReWire}(T, Z_{near}, z_{min}, z_{new})$

```
for z_near ∈ Z_near \ {z_min} do
    (x', u', T') ← Steer(z_new, z_near);
    if ObstacleFree(x') & x'(T') = z_near and
       Cost(z_new) + c(x') < Cost(z_near) then
        T ← ReConnect(z_new, z_near, T);
return T;
```

★ Extensions for Anytime Motion Planning

A. Committed Trajectory

⇒ Upon receiving the goal region, the online planning algorithm starts an initial planning phase, in which RRT* runs until the robot must start moving toward ~~*~~ its goal.

⇒ Once the initial planning phase is completed, the online algorithm goes into an iterative planning phase, in which the robot starts to execute the initial portion of the best trajectory in the tree maintained by the RRT* algorithm.

⇒ Meanwhile, the RRT* algorithm focuses on improving the remaining part of the trajectory.

⇒ Once the robot reaches the end of the portion that it is executing, the iterative phase is restarted by picking the current best path in the tree and executing its initial portion.

⇒ Given a motion plan $x : [0,T] \rightarrow X_{free}$ generated by the RRT* algorithm, the robot starts to execute an initial portion of $x : [0, t_{com}]$ until a given commit time t_{com}.

↳ We refer this initial path as the Committed trajectory.

⇒ Once the robot starts executing the committed trajectory, the RRT* algorithm deletes each of its branches and declares the end of the committed trajectory $x(t_{com})$ to be the new tree root.

⇒ As the robot proceeds along the committed trajectory, the RRT* algorithm continues to improve the motion plan within the new (uncommitted) tree of trajectory.

⇒ Once the robot reaches the end of the committed trajectory, the procedure restarts.

⇒ The iterative phase repeats until the robot reaches the goal region.

B. Branch-and-Bound

⇒ In addition to considering a committed trajectory, we also employ a <u>branch & bound</u> technique to more efficiently build the tree.

↳ Used in many domain of optimization and artificial intelligence.

1. Cost-to-go function

⇒ For an arbitrary state $z \in X_{free}$, let c_z^* be the cost of the optimal path that starts at z and reaches the goal region X_{goal}.

⇒ A cost-to-go function CostToGo(z) associates each $z \in X_{free}$ with a real number between 0 & c_z^*.

⇒ The cost-to-go function described here is equivalent to the admissible heuristic employed by A* planning algorithms.

⇒ There are many ways to define a cost-to-go function, the most trivial being $CostToGo(z) = 0 \ \forall \ z \in X_{free}$.

⇒ In this paper, we use the Euclidean distance between z and X_{goal} (neglecting obstacles) divided by the maximum speed of the vehicle as a cost-to-go function.

② Branch and bound algorithm

⇒ Let $T = (V, E)$ be a tree and $z \in V$ be a vertex in T.

⇒ Let Z_{min} be the node that lies in the goal region and has the lowest-cost trajectory that reaches X_{goal} along the edges of T.

⇒ Let V' denote the set of nodes z for which the cost to get to z, plus the lower-bound on the optimal cost-to-go, is more than the upper bound

$$V' = \{ z \in V \mid Cost(z) + CostToGo(z) \geq Cost(Z_{min}) \}$$

⇒ The branch & bound algorithm keeps track of all such nodes and periodically deletes them from the tree.

★ System Dynamics & the Control Procedure

A. Dubins Curve Steering function

⇒ Dubins vehicle dynamics have the general form:

$$\boxed{\begin{aligned} \dot{x}_D &= V_D \cos(\theta_D) \\ \dot{y}_D &= V_D \sin(\theta_D) \\ \dot{\theta}_D &= u_D \qquad |u_D| \leq \frac{V_D}{\rho} \end{aligned}}$$

- where (x_D, y_D) and θ_D specify the position and orientation.
- u_D ⇒ Steering input
- V_D ⇒ Velocity
- ρ ⇒ minimum turning radius

B. Trajectory Tracking

⇒ The steering function returns a trajectory parameterized by a sequence of reference states (x_R, y_R, θ_R) and a reference velocity V_R.

⇒ Let Z_n be the robot's current state and Z_{n+1} be the next reference point.

⇒ Define the cross-track error e_{ct}: be the distance between Z_n & Z_{n+1}: along a line $\perp$ to the desired orientation θ_{n+1}.

⇒ We steer the vehicle along the trajectory by controlling the steering angle δ via

$$\boxed{\delta = K_{str} \arctan(K_{ct} e_{ct}) + K_{str} e_{\theta}}$$

where K_{str} and K_{ct} are gains.

⇒ Meanwhile, we employ a PI controller to track the ~~staff~~ reference speed V_R.

$$\boxed{u = K_p (V_R - V) - K_i \int_0^t (V_R - V(\tau))\, d\tau}$$
