ME 599/699 Robot Modeling & Control
Planning

Instructor: Hasan Poonawala

Contents

1 Preliminaries

Definition 1 (Path)**.** A path in $\mathbb{R}^n$ is a continuous function γ from the unit interval $I = [0, 1]$ to $\mathbb{R}^n$.

Definition 2 (Trajectory)**.** A trajectory $q(t)$ in $\mathbb{R}^n$ is a a continuous function q from the an interval of time $[t_0, t_f]$ to $\mathbb{R}^n$.

Definition 3 (Graph)**.** A graph G is an ordered pair $G = (V, E)$ where

- V is a set
- E is a set of (ordered) pairs of elements in V

The interpretation of G is that V is a set of nodes, and E describes directed edges that indicate an 'immediate' operation from one node to another.

2 Path And Trajectory Planning

The trajectory planning problem can be cast as an optimization problem. Suppose we can measure the 'goodness' of a trajectory by a function J. We want to find a solution $q^*(t)$ of the problem:

$$\begin{aligned} \min_{q(t)} \quad & J(q(t)) \\ \text{subject to} \quad & \\ & \text{Robot doesn't destroy itself or things} \\ & \text{Other concerns} \end{aligned}$$

This version of the problem doesn't worry about control. Out pops q^*t and we try and use the trajectory tracking controllers.

Typically solved using heuristics. Why ?

Let's break this down:

Simplest J: all trajectories have value 0 (the same, not useless).

Important thing is to not hit things.

Start with finding a path. Then, attach time to the path to get a trajectory.

- Figure out the Obstacle-free configuration space (difficult, use over-approximations of robot and obstacles)
- Sample points in free space (easy)
- Connect points in free space (depends)

How to connect?

- Potential Field + random walk
- Probabilistic Road Maps
- Rapidly-exploring Random Trees

Attaching time. One approach: use polynomial function of sufficient degree to specify initial point, final point, initial velocity, final velocity, and add accelerations.

Finding coefficients given start and end times and values is a linear program.

3 Potential Field Planning

The aim is to define a continuous real-valued function $U\colon \mathcal{Q} \to \mathbb{R}$ where $\mathcal{Q}$ is the configuration space such that configuration q_f is the unique minimum of the U. A gradient descent algorithm for minimizing U generates a sequence of points in the state space corresponding to a path from an initial point q_0 to the final point q_f. The function U is called a potential function, and its gradient defines a force F as

$$F = -\nabla U(q) \tag{1}$$

The idea for this approach comes from the behavior of particles with inertia moving in potential fields that imply certain forces acting on the particle.

The simplest way to construct U is to express it as a sum of the form

$$U(q) = U_{att}(q) + U_{rep}(q), \tag{2}$$

where U_{att} is an term designed to 'attract' q to q_f. For example,

$$U_{att}(q) = \begin{cases} \frac{1}{2}\|q - q_f\|_2^2 & , \quad \text{if } \|q - q_f\|_2 \leq d \\ d\|q - q_f\|_2 - \frac{1}{2}d^2 & , \quad \text{if } \|q - q_f\|_2 > d \end{cases} \tag{3}$$

The terms U_{rep} will 'repel' the point q from obstacles during the gradient descent. Let $\rho(q)$ be the shortest distance of a point q from any obstacle. One example of such a term is

$$U_{rep}(q) = \begin{cases} \frac{1}{2}\left(\frac{1}{\rho(q)} - \frac{1}{\rho_0}\right)^2 & , \quad \text{if } \rho(q) \leq \rho_0 \\ 0 & , \quad \text{if } \rho(q) > \rho_0 \end{cases} \tag{4}$$

The function $\rho(q)$ can be discontinuous, unless we define ρ_0 to be small enough.

3.1 Gradient Descent

Algorithm 1 implements gradient descent for a function $U(q)$.

3.2 Task Space Potentials

Sometimes defining $\rho(q)$ is hard, but defining the obstacles in the task space is easy. In this situation, one may prefer to define a potential field in the task space, compute that gradient, and map the gradient to the configuration space.

Algorithm 1 Gradient Descent

Require: q_0, q_f, $U(q)$, $\epsilon > 0$. $\{U(q)$ must be positive demi-definite$\}$
Ensure: $q_k \approx \min_q U(q)$
  $q^0 \leftarrow q_s$
  $i \leftarrow 0$ {Loop counter}
  while $U(q) > \epsilon$ **do**
    $q^{i+1} \leftarrow q^i - \frac{\alpha^i}{\|\nabla U(q)\|} \nabla U(q)$
    $i \leftarrow i + 1$
  end while
  return $\{q^j\}_{j \in \{1,\ldots,i\}}$

We represent the robot by the frame origins $o_i^0(q)$ for $i \in 1, \ldots, 6$. Sometimes, we also include additional points to represent parts of the robots not close to these frame origins, to account for the physical space occupied by the robots.

We define potential functions for each point $o_i^0(q)$ and compute the gradient. These gradients represent 'forces' F_i in task space that should act on the frame origins. To obtain the corresponding force in coordinate space, we use the transformation $\tau_i = J_v^T F_i$.

4 Probabilistic Road Maps

The probabilistic roadmap planner is a motion planning algorithm in robotics, which solves the problem of determining a path between a starting configuration of the robot and a goal configuration while avoiding collisions.

Motion planning using a PRM involves a construction phase and a query phase.

4.1 Construction

The construction phase involves building a graph, which is a set of nodes and a set of edges.

1. Take random samples from the configuration space of the robot
2. Test for membership in free space
3. If in free space, add node to PRM graph
4. select subset of existing nodes based on proximity
5. Connect new node to selected nodes by straight line paths, check for collisions by sampling along these paths
6. If there are no collisions for a path, add corresponding edge to PRM

4.2 Query

Once a sufficiently dense PRM is available, one can query the algorithm to connect any initial and goal points. Connect initial and goal points to nearest nodes creating a graph corresponding to that query. Use Djikstra's shortest path algorithm to connect the initial and final nodes in the graph. The paths along the edges of the shortest path yield the planned path.

4.3 Analysis

Given certain relatively weak conditions on the shape of the free space, PRM is provably probabilistically complete, meaning that as the number of sampled points increases without bound, the probability that the algorithm will not find a path if one exists approaches zero. The rate of convergence depends on certain visibility properties of the free space, where visibility is determined by the local planner. Roughly, if each point can "see" a large fraction of the space, and also if a large fraction of each subset of the space can "see" a large fraction of its complement, then the planner will find a path quickly.

5 RRT

First, a quick summary of RRT(*)

1. A fundamental assumption to these planning methods is that we can represent the task as a sequence of nodes.
2. RRT is about computing (growing) an under-approximating abstraction of a complex state space with a tree topology.
3. Two important questions:
   (a) How to find a new leaf
   (b) Where to connect new leaf to the tree?
4. Loosely, RRT answers 3a by random sampling, and 3b by nearest-node in underlying space (more accurate details below).
5. RRT* improves RRT by using weights for the edges, and ensuring that all paths to leaves are minimum-weight.
6. Informed RRT* answers 3a for a subclass of edge weights, namely, length. For shortest-path problems between two given points, to provably reduce length of current path by sampling, it is necessary to sample a particular ellipsoid defined by points and the current optimal path.

RRT(*) works off of a transition system $TS = (S, Act, \rightarrow, I, AP, L)$ that abstracts the underlying dynamical system whose motion we wish to plan for. The states S are a finite set of points in the state space of the dynamical system. For RRT(*), an action is to attempt to reach another state. Therefore, $Act = S$. The available transitions are encoded in $\rightarrow$. The key point is that TS models a tree, therefore only one action actually results in a transition in each state, to a unique successor node not identical to the current state. I is the root node/state of the tree. For RRT(*), an obvious simple definition of AP is $AP = \{goal, waypoint\}$.

The goal of RRT*is to find a minimum cost path from the initial state I to the goal state in the dynamical system. The goal states are $s \in S$ such that $L(s) = goal$. Note the identification of states in TS and the dynamical system's state space. TS is not fixed, but grown incrementally, one state at a time. These states come from the underlying dynamical system. To justify that TS simulates the dynamical system, RRT*also needs the following functions.

- *sample*
- *nearest neighbor*
- (*local*) *steer*

- *collision check*
- *nearest vertex*
- *cost* or *distance*

The *cost* is required when finding an optimal path, though it can trivially be taken as 0 to make any found path optimal.

We start the growth of TS by obtaining a sample z from X using *sample*. The point of z is to become a temporary goal for the nodes in the tree, to help it extend into unexplored regions. We find the nearest neighbor s_v to candidate z in S, using *nearest neighbor*. The steer function in RRT returns a point y that minimizes distance from y to z while being only so far away from v.

The reason for using y and not z is that for a new point to accepted, there must be a transition from S to it. In RRT without dynamics, the assumption is that the free space is connected, so there must be such a transition. However, if z is far from v, the connection is likely to pass through an obstacle. Therefore, we first find y, and attempt to connect it to the tree after it passes a collision check. Since y is not too far from v, it is more likely that the path from y to v is collision-free.

In RRT or RRT*with dynamics, the *steer* function finds an (optimal) trajectory between y and v, where again y is closest possible to z, but only so far from v. The trajectory returned by the *steer* function confirms that we can connect y to v. Again, there may be an obstacle in the way, and *collision check* determines if the entire trajectory is safe from collision. Once the trajectory is shown to be collision-free, we can add s_y with a transition from its nearest neighbor. Again, to prevent wasted collision checks, we aim to add y and not z, since the latter could be far from v.

In RRT*we then rewire the connections between the outputs of *nearest vertex* to s_y using cost-information. This rewiring makes sure that the transitions in TS encodes optimal paths.

5.1 RRT* Simulation

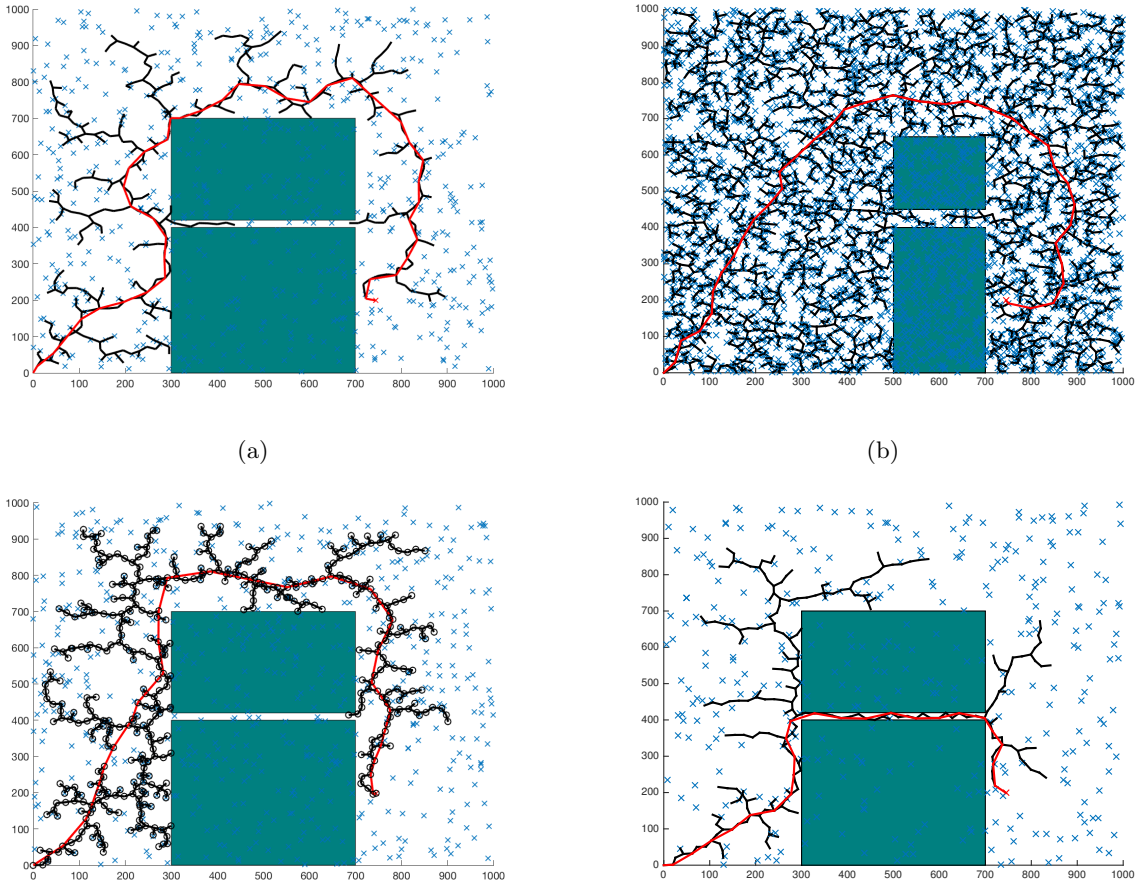
This section focuses on the example provided, and the concern that RRT* is not likely to find the shorter path through the gap. The folder RRT_Star_2D contains code that implements the RRT* algorithm for motion planning in two dimensions. There are no dynamic constraints. This code was downloaded from MATLAB's forums, and modified. There might be some errors in the re-wiring step, but it runs well enough to see how the RRT tree grows as samples are added.

Figure 1 shows the tree without rewiring (in black), with the optimal path in red. The initial state is the origin in the lower left corner, the goal state is the red check mark. Figure 2 shows a branch of the tree that successfully passed through the narrower gap.

Observations. The outputs of the `nearest-neighbor`, the `steer`, and the `collision_check` functions interact with the existing nodes to promote or inhibit growth of the tree through the gap. For a path to be found through the gap, the following two situations seem to be necessary

- Nodes near the entrance should be 'in front' of the gap.
- The tree in the longer path must not be well developed.

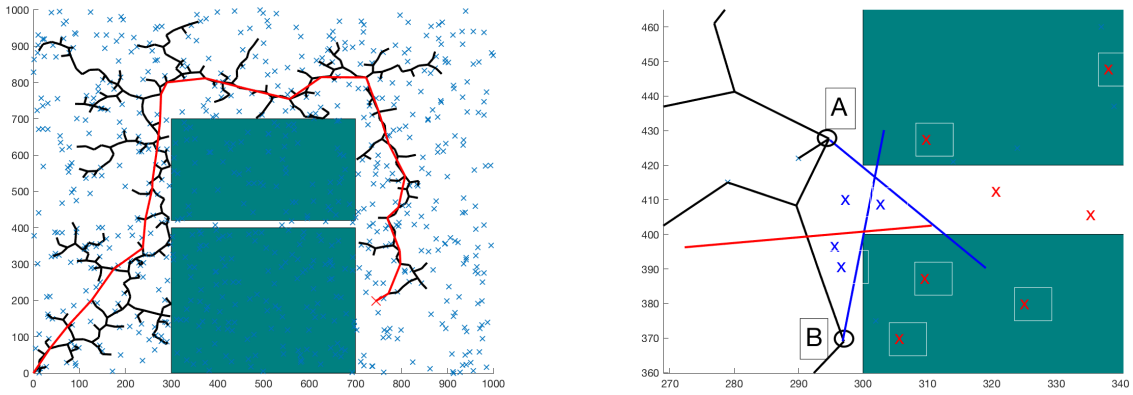
Figure 2 shows two two nodes at points A and B that inhibit growth of the tree through the gap since they create a very small region that a sample must lie in to successfully create a new node of the tree lying within the gap. For any sample to create a branch through the gap, it will have to select A or B as the nearest



(a) (b)

(c) (d)

Figure 1: A 2D path planning scenario with two obstacles, with a red check mark indicating the goal at $(750, 200)$. The origin $(0, 0)$ is the root node. The tree obtained using RRT* is in black. (a) The shortest path in the tree found by RRT* does not pass through the narrow gap. (b) RRT* may ignore shorter and wider gaps. (c) The gap is not even explored, unlike in a) (d) The optimal path goes through the gap.



(a) (b)

Figure 2: Another run of RRT* on the 2D path planning example where the gap is never explored. (a) The entire tree and optimal path. (b) A blow-up of nodes close to the left entrance. Node A and B interact with the obstacles to create a very small region within which samples can successfully grow the tree through the gap. Examples of successful samples are the blue cross marks. The samples marked in red (even the sample in free space) would be rejected, since any sufficiently long path from points A or B passes through an obstacle.

node. Unfortunately, the obstacle intersects paths between most of such samples and the nodes at A and B, causing a rejection of those samples. This inhibition allows the other branch to grow longer while no progress is made on the branch through the gap most of the time (unless the small region is sampled).

Figure 3 depicts the location of a node near the left end of the gap that would enable growth of the tree through the gap. The growth is possible because a vey large set of samples can lead to successfully placing new nodes within the gap. However, if the alternate branch has grown significantly, then again the path through the gap is not discovered. The new samples that could have promoted growth from C instead select D as their closest node, and grow the branch emanating from D.

Importance Sampling This mechanism hopefully convinces the reader that the issue is not that samples will rarely fall within the gap. The issue is that the components of the RRT* algorithm interact in a way that makes the usefulness of a sample non-deterministic.

One simple way to overcome the second necessary condition above is to use new samples to grow all 'branches'. The technical problem is characterizing how many 'branches' there are, and which set of nodes belong to which branch. There are probably algorithms for versions of this problem already, if not the exact problem. Solving this problem is a way of incorporating some notion of topology into the RRT* algorithm.

The first necessary condition is perhaps a place for important sampling, where one identifies these bottleneck node placement situations and *avoids* placing nodes there, or promotes placement of nodes in 'extendable' positions.

6 Trajectories From Paths

The algorithms we've mentioned so far provide a sequence of points that belong to a path joining q_0 and q_f. We can connect any two points by a straight line, thereby defining a path between the two points.

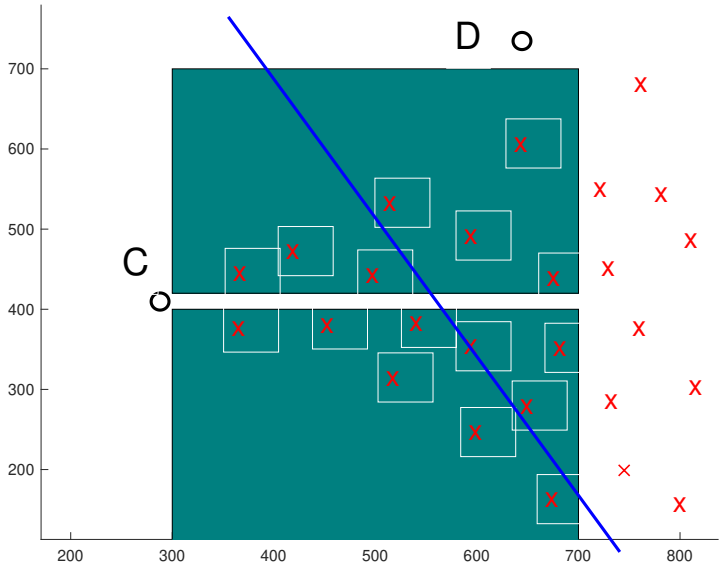


Figure 3: Point C, in the absence of point D, will be extended through the gap by samples in a large region, potentially represented by the red check marks. Note that if a one of the blue check marks closer to point A in Figure 2b were sampled, they would become points similar to point C here. If the alternate branch grows, and a node exists at point D, then the samples above the blue line would be closer to node D, inhibiting growth of the branch through the gap.

What we want is a trajectory, which means that we associate each point on the path with a time in a continuous manner. We may also want the path to be sufficiently smooth, so that the derivatives are bounded. Since each point corresponds to a unique time, the first and second derivatives of a trajectory correspond to velocity and acceleration respectively.

The full problem is known as trajectory optimization.

Typically, we are converting line segments defined by end points into trajectories. The start and end points are fixed, and we assume that the task define the velocities and accelerations only at the end points. We will search for trajectories that satisfy these requirements on the end points from the set of polynomial trajectories.

6.1 Polynomials

We consider each joint coordinate separately, since they are independent scalars. Suppose we know that

$$q(t_0) = q_0, \qquad q(t_f) = q_f \tag{5}$$
$$\dot{q}(t_0) = v_0, \qquad \dot{q}(t_f) = v_f \tag{6}$$

We have four constraints, and so we use a cubic polynomial to generate $q(t)$ satisfying these constraints:

$$q(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3. \tag{7}$$

We rewrite this into the linear equation

$$q_0 = a_0 + a_1 t_0 + a_2 t_0^2 + a_3 t_0^3 \tag{8}$$
$$v_0 = a_1 + 2a_2 t_0 + 3a_3 t_0^2 \tag{9}$$
$$q_f = a_0 + a_1 t_f + a_2 t_f^2 + a_3 t_f^3 \tag{10}$$
$$v_f = a_1 + 2a_2 t_f + 3a_3 t_f^2 \tag{11}$$

$$\begin{bmatrix} 1 & t_0 & t_0^2 & t_0^3 \\ 0 & 1 & 2t_0 & 3t_0^2 \\ 1 & t_f & t_f^2 & t_f^3 \\ 0 & 1 & 2t_f & 3t_f^2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} q_0 \\ v_0 \\ q_f \\ v_f \end{bmatrix} \tag{12}$$

The determinant of the matrix is $(t_f - t_0)^4$, so that a solution exists on all non-trivial time intervals.

If we want to specify accelerations, we need quintic polynomials, and obtain similar equations. Why specify accelerations? So that there are no discontinuities when combining multiple segments, which would require infinite jerk.

6.2 Parabolic Blends

Divide the time interval $[t_0, t_f]$ into three segments where the velocity on the middle segment is specified, and the first and last segment represent smooth transitions between zero velocity and the specified middle segment velocity.

Minimum Time Trajectories Final time is not fixed, just need to when to switch from positive maximum acceleration to negative maximum acceleration.

7 Optimal Control

If we keep seeing that the control isn't able to follow the trajectory returned by the previous methods, perhaps we need to constrain trajectories using control. Leads to an optimal control problem.

$$\begin{aligned} \min_{q(t)} \quad & J(q(t)) \\ \text{subject to} \quad & \\ & q(t) \text{ satisfies dynamics and input constraints} \\ & \text{Robot doesn't destroy itself} \\ & \text{Other concerns} \end{aligned}$$