

Motion Planning

DeepBlue Note

collated by decem17

2 Search-based Path Finding

2.1 Graph Search Basis

配置 configuration: 机器人上所有点的位置描述.

DOF: 最少需要 n 条坐标轴来表示机器人的配置.

配置空间 C-space: 一个包含了机器人所有可能的配置的 n 维空间. 机器人的每个位姿 pose 都可用一个点表示.

在 C-space 中做 planning 之前需要把障碍物表示在内, 称为 C-obstacle.

2.2 Graph and Search Method

图有节点 nodes 和边 edges. 边分为有向和无向, 有权重和无权重.

图搜索可以构建出一棵搜索树.

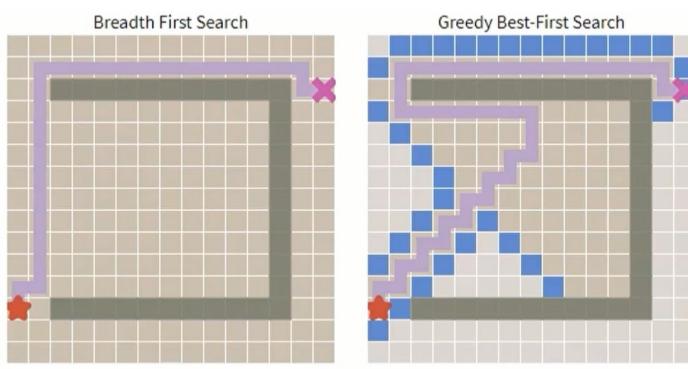
图搜索算法:

1. 构建一个容器 container 以保存所有经过的点 the nodes to be visited.
2. 容器初始化时加入起始点.
3. 循环: 弹出, 扩展, 推入.
4. 当 container 为空时结束循环.
5. 一个节点被弹出后, 就不应再次被推入.

BFS 维护的是 queue, DFS 维护的是 stack.

BFS 可找到最短路径, DFS 则不能.

启发式搜索 heuristic search, 即贪心搜索 greedy best first search. 启发式是指当前点离目标点有多远的估计, 常用 Euclidean distance / Manhattan distance.



But with obstacles ...

2.3 Dijkstra's algorithm

workflow:

- Maintain a **priority queue** to store all the nodes to be expanded
- The priority queue is initialized with the start state X_S
- Assign $g(X_S)=0$, and $g(n)=\infty$ for all other nodes in the graph
- Loop
 - If the queue is empty, return FALSE; break;
 - Remove the node "n" with the lowest $g(n)$ from the priority queue
 - Mark node "n" as expanded
 - If the node "n" is the goal state, return TRUE; break;
 - For all unexpanded neighbors "m" of node "n"
 - If $g(m) = \infty$
 - $g(m) = g(n) + C_{nm}$
 - Push node "m" into the queue
 - If $g(m) > g(n) + C_{nm}$
 - $g(m) = g(n) + C_{nm}$
 - end
- End Loop

优点: 完备且最优.

缺点: 搜索时没有方向性, 朝所有方向均匀扩散.

2.4 A* algorithm

累积 cost: $g(n)$, 启发项: $h(n)$.

workflow:

- Maintain a **priority queue** to store all the nodes to be expanded
 - The heuristic function $h(n)$ for all nodes are pre-defined
 - The priority queue is initialized with the start state X_S
 - Assign $g(X_S)=0$, and $g(n)=\infty$ for all other nodes in the graph
 - Loop
 - If the queue is empty, return FALSE; break;
 - Remove the node "n" with the lowest $f(n)=g(n)+h(n)$ from the priority queue
 - Mark node "n" as expanded
 - If the node "n" is the goal state, return TRUE; break;
 - For all unexpanded neighbors "m" of node "n"
 - If $g(m) = \infty$
 - $g(m) = g(n) + C_{nm}$
 - Push node "m" into the queue
 - If $g(m) > g(n) + C_{nm}$
 - $g(m) = g(n) + C_{nm}$
 - end
 - End Loop
- Only difference comparing to
Dijkstra' s algorithm

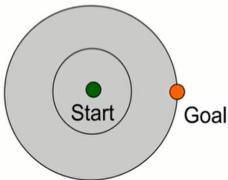
priority queue in C++: std::priority_queue / std::make_heap / std::multimap

open list - priority queue, close list – stores all expanded nodes.

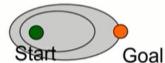
admissible 的启发性:

1. 对于所有的节点, $h(n)$ 都小于其真实的路径 $h^*(n)$.
2. 若 heuristic 是 admissible 时, A*是最优的.
3. 一个启发式函数的 admissible 要根据实际情况决定. 如机器人可任意方向运动时, 采用 Manhattan distance 就不是 admissible.

- Dijkstra' s algorithm expanded in all directions



- A* expands mainly towards the goal, but does not hedge its bets to ensure optimality



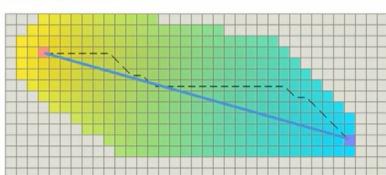
Weighted A* Search: $f = g + \varepsilon h$, $\varepsilon > 1$. bias towards states that are closer to goal.

1. 次优的路径. $cost \leq \varepsilon cost(optimal\ solution)$.
2. 更快. 用最优化换取了速度.
3. Weighted A* -> Anytime A* -> ARA* -> D*

The Best Heuristic

1. 默认为八方向运动. 以上的 $h(n)$ 都不是最好的, 因为不是 tight, 即 $h^*(n) - h(n)$ 很大.
2. Tight 是指 $h(n)$ 测量的是真实的最短距离.

Euclidean Heuristic

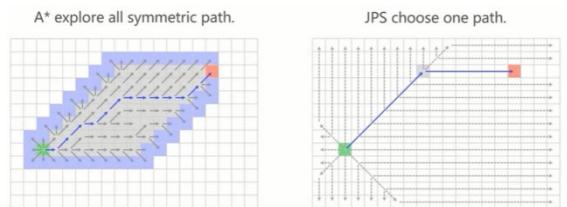


Tie Breaker: 打破路径平衡性.

1. 许多路径有相同的 f 值, 这使得 A*在探索时对这些路径没有倾向性.
2. 解决方法:
 - 2.1 $h = h^* (1.0 + p)$, $p < \min \text{cost of one step} / \text{expected max path cost}$. p 是一个非常小的值.
 - 2.2 当 nodes 有相同的 f 时, 比较它们的 h .
 - 2.3 $h = h + 0.001 * \text{cross}$. cross 是指当前点到起点与终点连线的偏移量.

2.5 Jump Point Search

核心思想: 找到对称性并打破它们.



Look Ahead Rule:

灰: inferior neighbors, 舍弃.

白: natural neighbors, 探索时只考虑白色节点.

红: forced neighbors, 必须被加入进 open list.

1	2	3
4 → x	5	
6	7	8

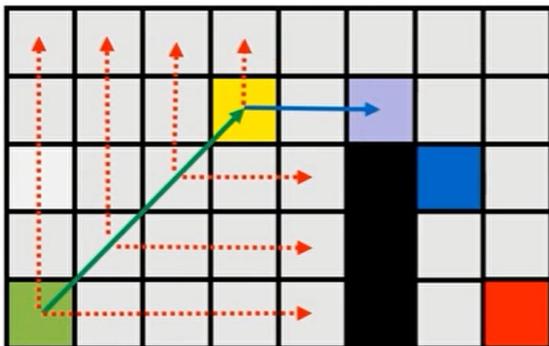
1	2	3
4	x	5
6	7	8

1	2	3
4 → x	5	
6	7	8

1	2	3
4	x	5
6	7	8

Jumping Rules:

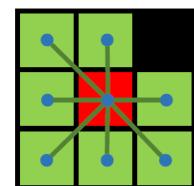
1. 探索时先水平/竖直，再对角。到边界/障碍物处停止。
2. 目标点视为 forced neighbor。
3. 探索到 forced neighbors 时，把 the node with a forced neighbor 的前一个节点加入进 open list。如下图，从绿点开始探索，到黄点时发现蓝点是紫点的一个 forced neighbors，因此把黄点加入进 open list。



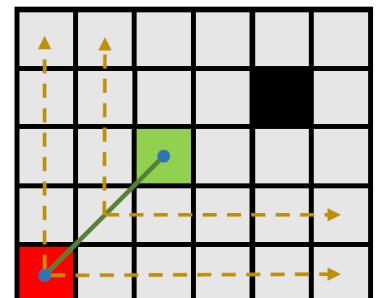
JPS 和 A*的伪代码是一样的，只是推入 open list 中的 neighbors 的选择不同。

- Maintain a priority queue to store all the nodes to be expanded
- The heuristic function $h(n)$ for all nodes are pre-defined
- The priority queue is initialized with the start state X_s
- Assign $g(X_s)=0$, and $g(n)=\infty$ for all other nodes in the graph
- Loop
 - If the queue is empty, return FALSE; break;
 - Remove the node “n” with the lowest $f(n)=g(n)+h(n)$ from the priority queue
 - Mark node “n” as expanded
 - If the node “n” is the goal state, return TRUE; break;
 - For all unexpanded neighbors “m” of node “n”
 - If $g(m) = \infty$
 - $g(m) = g(n) + C_{nm}$
 - Push node “m” into the queue
 - If $g(m) > g(n) + C_{nm}$
 - $g(m) = g(n) + C_{nm}$
 - end
- End Loop

A*: “Geometric” neighbors

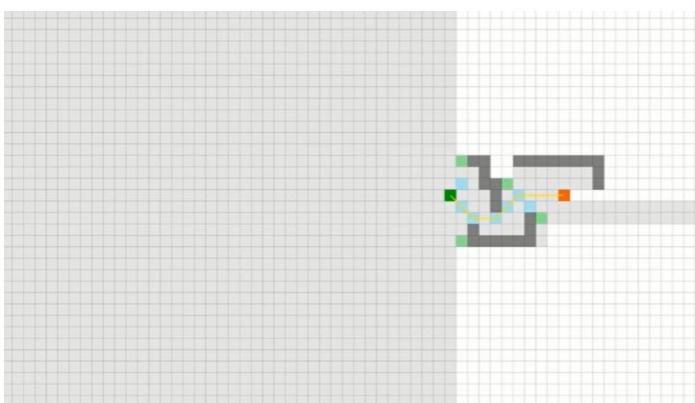


JPS: “Jumping” neighbors



优点: 大多数情况下，特别是在复杂的环境中，JPS 更好。因为 JPS 减少了 open list 中节点数量。

缺点: 但 JPS 增加了 grid 的 status query。如下图，机器人有一个有限的视野 FOV，即无法探索到远处的障碍物，但有一个大的地图。并且 JPS 只使用于 uniform grid map。



3 Sample-based Path Finding

基于搜索: 搜索从起始点到目标点之间所有的空间点.

基于采样: 不需要遍历所有的点. 通过在空间中随机撒点构建出 roadmap.

完备性: 是指如果在起始点和目标点间有路径解存在, 那么一定可以得到解, 如果得不到解那么一定说明没有解存在.

概率完备性: 是指如果在起始点和目标点间有路径解存在, 只要规划或搜索的时间足够长, 就一定能确保找到一条路径解.

最优性: 是指规划得到的路径在某个评价指标上是最优的(评价指标一般为路径的长度).

渐进最优性: 是指经过有限次规划迭代后得到的路径是接近最优的次优路径, 且每次迭代后都与最优路径更加接近, 是一个逐渐收敛的过程.

3.1 Probabilistic Road Map (PRM)

规划分为 learning phase 和 query phase.

Learning phase: 1. 在 C-space 内按一定规则采样(撒点), 再将位于 C-obstacle 内的点删去.

2. 把点用线段连接, 包括起始点和目标点. 各线段有最大长度限制. 再把 C-obstacle 内的线段删去.

Query phase: 用 Dijkstra 或 A*搜索出起始点和目标点的最短路径. (简化的 grid map)

PRM 优点: 概率完备.

缺点: 需要解决两点间的边界值问题 boundary value problem, 因为是用直线连接, 不适合机器人行走.
在 C-space 中构建图花费了大量时间, 没有专注于生成路径. 不够高效.

如何提高效率? Lazy collision-checking: 不进行碰撞检测, 因为碰撞检测很耗时 time-consuming.
在规划出路径后对路径进行检测, 删去碰撞的线段.

3.2 Rapidly-exploring Random Tree (RRT)

Algorithm 1: RRT Algorithm

Input: $\mathcal{M}, x_{init}, x_{goal}$

Result: A path Γ from x_{init} to x_{goal}

$\mathcal{T}.init();$

for $i = 1$ to n **do**

$x_{rand} \leftarrow Sample(\mathcal{M});$

$x_{near} \leftarrow Near(x_{rand}, \mathcal{T});$

$x_{new} \leftarrow Steer(x_{rand}, x_{near}, StepSize);$

$E_i \leftarrow Edge(x_{new}, x_{near});$

if $CollisionFree(\mathcal{M}, E_i)$ **then**

$\mathcal{T}.addNode(x_{new});$

$\mathcal{T}.addEdge(E_i);$

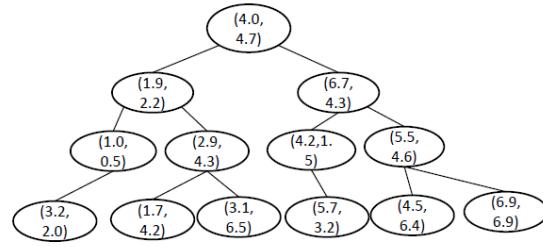
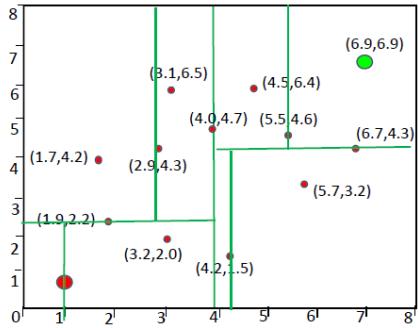
if $x_{new} = x_{goal}$ **then**

 Success();

RRT 优点: 专注于搜索起始点和目标点的路径. 比 PRM 更面向目标 target-oriented.

缺点: 不是最优的. 不够高效. 在整个空间中采样.

改进: Kd-tree: 采样出新的 x_{rand} 时, 帮助更快速地找到 x_{near} . 横向找所有节点的中位数, 然后纵向找 2 个部分各自的中位数...



改进: Bidirectional RRT / RRT Connect: 从起始点和目标点分别构建 2 棵树. (narrow path)

3.3 RRT*

Algorithm 2: RRT Algorithm

Input: $\mathcal{M}, x_{init}, x_{goal}$

Result: A path Γ from x_{init} to x_{goal}

$\mathcal{T}.init();$

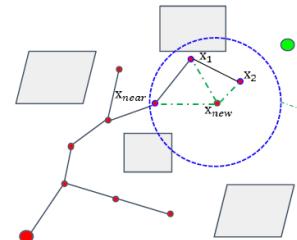
for $i = 1$ to n **do**

```

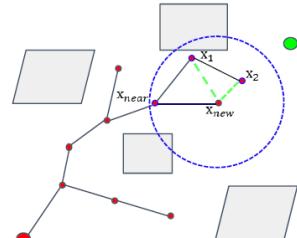
 $x_{rand} \leftarrow Sample(\mathcal{M});$ 
 $x_{near} \leftarrow Near(x_{rand}, \mathcal{T});$ 
 $x_{new} \leftarrow Steer(x_{rand}, x_{near}, StepSize);$ 
if CollisionFree( $x_{new}$ ) then
     $X_{near} \leftarrow NearC(\mathcal{T}, x_{new});$ 
     $x_{min} \leftarrow ChooseParent(X_{near}, x_{near}, x_{new});$ 
     $\mathcal{T}.addNodEdge(x_{min}, x_{new});$ 
     $\mathcal{T}.rewire();$ 

```

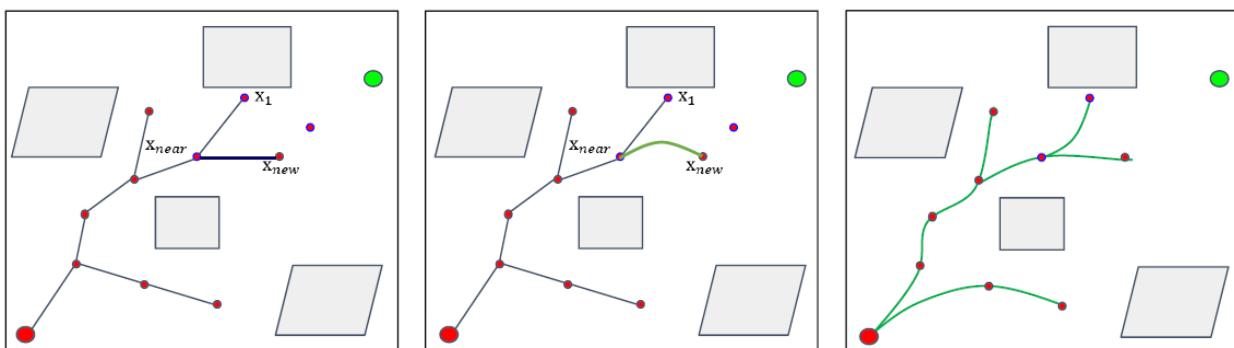
chooseParent:



rewire:



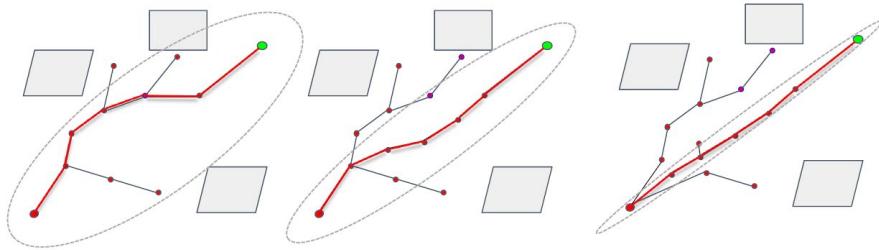
Kinodynamic-RRT*: 改进 steer() 函数, 考虑机器人的运动学约束.



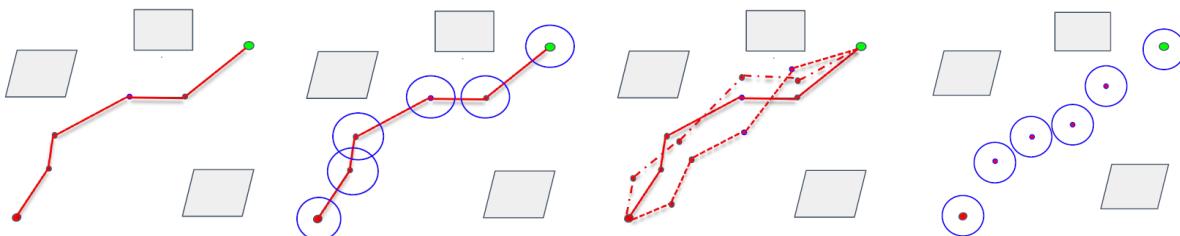
Anytime-RRT*: 机器人一边运动一边实时运行 RRT*.

3.4 Advanced Sampling-based Methods

Informed RRT*: 生成路径之后，把采样的过程限制在一个椭圆内，然后逐渐减小该椭圆.



Cross-entropy motion planning: 生成路径之后，在节点附近采样，得到多条路径. 然后以均值为圆心开始下一轮采样.



4 Kinodynamic Path Finding

考虑机器人运动学模型的运动规划. 需考虑障碍物约束 / 微分约束 / 加速度约束.

不再考虑质点模型，我们希望构建出可行的运动连接的图.

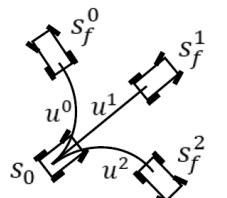
构建图分为 2 种方法

正向: 在控制空间 control space 中离散化(采样)

逆向: 在状态空间 state space 中离散化(采样)

机器人模型的状态方程: $\dot{s} = f(s, u)$, u 为系统的输入. 初始状态 s_0 .

给定一个输入量 u , T 时间后, 正向仿真该系统:



- Forward simulation
- Fixed u, T
- No mission guidance,
- Easy to implement
- low planning efficiency

给定一个状态 s_f , 则 s_0 到 s_f 的轨迹为: (启发性)



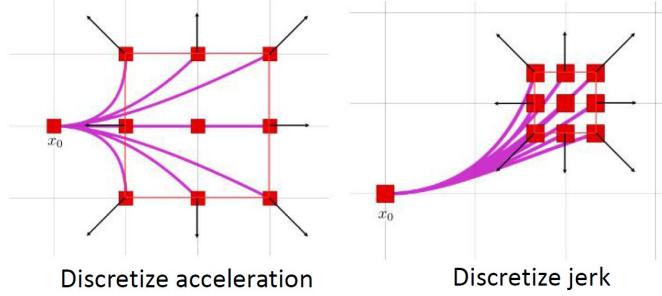
- Backward calculation
- Need calculate u, T
- Good mission guidance
- Hard to implement
- High planning efficiency

4.1 Sample in Control Space

状态 $s = (x \ y \ z \ \dot{x} \ \dot{y} \ \dot{z})^T$ 输入 $u = (\ddot{x} \ \ddot{y} \ \ddot{z})^T$ 则系统方程为 $\dot{s} = A \cdot s + B \cdot u$, A 为幂零矩阵 nilpotent. 幂零矩阵的性质为在 i 次方后, $A^i = 0, A^{i+1} = 0, A^{i+2} = 0, \dots$

$$A = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad B = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

1. 给定 $v_0 = [1 \ 0 \ 0]^T$ 和 T , 离散化加速度(即把 u 平均分为 9 份)
2. 给定 $v_0 = [1 \ 0 \ 0]^T, a_0 = [0 \ 1 \ 0]^T$ 和 T , 离散化 jerk(此处的 u 为 jerk), 得到:



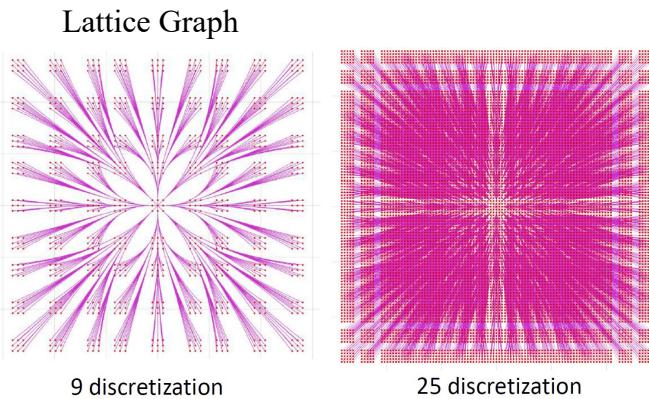
如何还原 2 个状态间的轨迹呢?

$$s(t) = \underbrace{e^{At} s_0}_{F(t)} + \underbrace{\left[\int_0^t e^{A(t-\sigma)} B d\sigma \right] u_m}_{G(t)}$$

u_m 为给定的控制量,
 e^{At} 为状态转移矩阵,
 $F(t)$ 为零输入相应,
 $G(t)$ 为零状态相应,

其中有 $e^{At} = I + \frac{At}{1!} + \frac{(At)^2}{2!} + \frac{(At)^3}{3!} + \dots + \frac{(At)^k}{k!} + \dots$

4.2 状态栅格搜索算法 State Lattice Planning

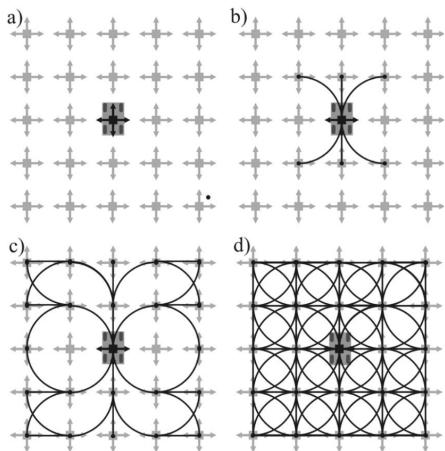


Note

- During searching, the graph can be built when necessary.
- Create nodes (state) and edges (motion primitive) when nodes are newly discovered.
- Save computational time/space.

在启发式函数的帮助下, 可以只构建一个方向的晶格图.

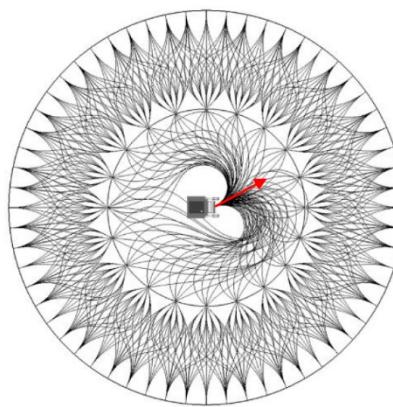
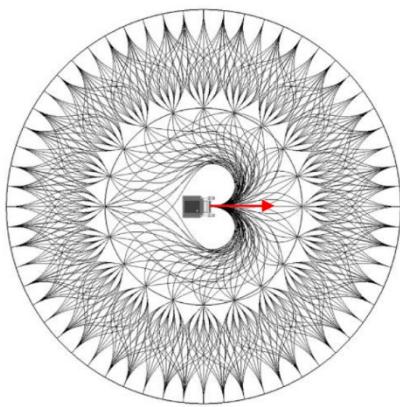
4.3 Sample in State Space



Build a lattice graph:

- Given an origin.
- for 8 neighbor nodes around the origin, feasible paths are found.
- extend outward to 24 neighbors.
- complete lattice.

Reeds-Shepp Car Model

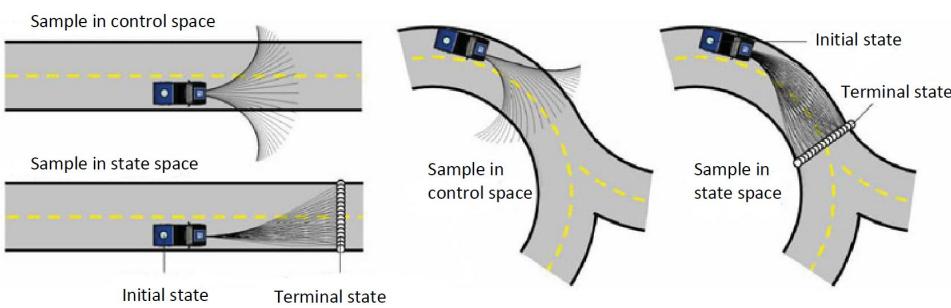


2 层 lattice graph

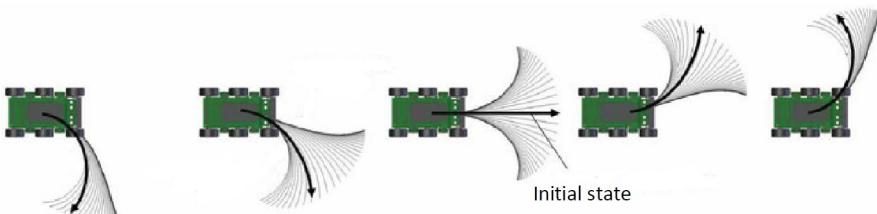
只有第 1 层是不一样的，因为初始状态不一样，而第 2 层是根据采样出来的状态彼此连接的

4.4 控制空间和状态空间采样的比较

在控制空间采样是非常没有目的性的，而状态空间采样能保证尽可能多的末状态可用。



只离散化控制空间，自由度是很小的，同质化很强。



- Trajectories are denser in the direction of the initial angular velocity.
- Very similar outputs for several distinct inputs.

那么给定初状态和末状态，如何求解中间状态？这是一个边界值问题 Boundary Value Problem.

4.5 边界值问题 Boundary Value Problem

BVP 是状态采样晶格规划的基础. 没有一般性的解决方法, case by case. 通常用复杂的数值最优化来解决.

一维系统: 轨迹为 $x(t)$. $x(0) = a$, $x(T) = b$. 参数化 $x(t) = c_5t^5 + c_4t^4 + c_3t^3 + c_2t^2 + c_1t + c_0$. 边界值条件 $x(0) = a$, $x(T) = b$, $v(0) = 0$, $v(T) = 0$, $a(0) = 0$, $a(T) = 0$. 因此求解:

$$\begin{bmatrix} a \\ b \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ T^5 & T^4 & T^3 & T^2 & T & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 5T^4 & 4T^3 & 3T^2 & 2T & 1 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \\ 20T^3 & 12T^2 & 6T & 2 & 0 & 0 \end{bmatrix} \begin{bmatrix} c_5 \\ c_4 \\ c_3 \\ c_2 \\ c_1 \\ c_0 \end{bmatrix}$$

会得到很多解, 怎么判断哪个解是最优的呢?

4.6 Optimal Boundary Value Problem

最小化 jerk. 其中 $J_k = \frac{1}{T} \int_0^T j_k(t)^2 dt$, k 表示某个轴. 状态 $s_k = (p_k, v_k, a_k)$. 输入 $u_k = j_k$.

系统模型为 $\dot{s}_k = f_s(s_k, u_k) = (v_k, a_k, j_k)$.

通过 Pontryain's minimum principle 求解. 定义协态 costate: $\lambda = (\lambda_1, \lambda_2, \lambda_3)$. 协态个数等于系统模型中的变量个数. 定义 Hamiltonian function: $H(s, u, \lambda) = g(s, u) + \lambda^T f_s(s, u) = \frac{1}{T} j^2 + \lambda_1 v + \lambda_2 a + \lambda_3 j$.

一般解法: $J = h(s(T)) + \int_0^T g(s(t), u(t)) dt$. $h(s(T))$ 为末状态的惩罚项(若没有这一项说明一定要达到该末状态). $\int_0^T g(s(t), u(t)) dt$ 为转移代价 transition cost. 要求解最优状态 s^* 和最优输入 u^* .

1. 写出 Hamiltonian function $H(s, u, \lambda) = g(s, u) + \lambda^T f_s(s, u)$ 和 costate $\lambda = (\lambda_1, \lambda_2, \lambda_3)$.
2. 设 $s^*(t) = f(s^*(t), u^*(t))$, 给定 $s^*(0) = s(0)$.
3. 根据 Pontryain's minimum principle, $\lambda(t)$ 是方程 $\dot{\lambda}(t) = -\nabla_s H(s^*(t), u^*(t), \lambda(t))$ 的解, 其中有边界条件 $\lambda(T) = -\nabla_s h(s^*(T))$
4. 把 $\lambda(t)$ 代入, 求出最优的输入量 $u^*(t) = \arg \min_{u(t)} H(s^*(t), u(t), \lambda(t))$.

举例: 最小化 jerk. $J_\Sigma = \sum_{k=1}^3 J_k$, 其中 $J_k = \frac{1}{T} \int_0^T j_k(t)^2 dt$. 状态 $s_k = (p_k, v_k, a_k)$, 输入 $u = j_k$. 系统方程为 $\dot{s} = f_s(s, u) = (p_k, v_k, a_k) = (v, a, j)$.

解: 引入协态 $\lambda = (\lambda_1, \lambda_2, \lambda_3)$ 和 Hamiltonian 函数 $H(s, u, \lambda) = g(s, u) + \lambda^T f_s(s, u) = \frac{1}{T} j^2 + \lambda_1 v + \lambda_2 a + \lambda_3 j$. 解出 $\dot{\lambda}(t) = -\nabla_s H(s^*(t), u^*(t), \lambda(t)) = (0, -\lambda_1, -\lambda_2)$ (对 H 求导 s , 即对 $p / v / a$ 求偏导).

不考虑边界条件, 因为本题中 $J = h(s(T)) + \int_0^T g(s(t), u(t)) dt$ 不考虑末状态的惩罚项 $h(s(T))$.

是一个硬约束. 也可表示为 $h(s(T)) = 0$ 当 $s(T) = \text{期望状态}$ or ∞ 当 $s(T) \neq \text{期望状态}$.

由 $\lambda = (\lambda_1, \lambda_2, \lambda_3)$ 和 $\dot{\lambda}(t) = (0, -\lambda_1, -\lambda_2)$ 解得 $\lambda(t) = \frac{1}{T} \begin{bmatrix} -2\alpha \\ 2\alpha t + 2\beta \\ -\alpha t^2 - 2\beta t - 2\gamma \end{bmatrix}$. (其中 $1/T$ 是为了后面推导方便), α, β, γ 本来应用边界条件 $\lambda(T) = -\nabla_s h(s^*(T))$ 求解的, 这里通过 $s(T) = \text{期望状态}$ 来求解.

因此最优输入 $u^*(t) = \arg \min_{u(t)} H(s^*(t), u(t), \lambda(t)) = \arg \min_{u(t)} \frac{1}{T} j^2 + \lambda_1 v + \lambda_2 a + \lambda_3 j = -\frac{2}{T} \lambda_3 =$

$$\frac{1}{2} \alpha t^2 + \beta t + \gamma \quad \text{最} \quad \text{优} \quad \text{状} \quad \text{态} \quad s^*(t) = \begin{bmatrix} p^* \\ v^* \\ a^* \end{bmatrix} = \begin{bmatrix} \int \int \int u^*(t) dt dt dt \\ \int \int u^*(t) dt dt \\ \int u^*(t) dt \end{bmatrix} =$$

$$\begin{bmatrix} \frac{\alpha}{120} t^5 + \frac{\beta}{24} t^4 + \frac{\gamma}{6} t^3 + \frac{\alpha_0}{2} t^2 + v_0 t + p_0 \\ \frac{\alpha}{24} t^4 + \frac{\beta}{6} t^3 + \frac{\gamma}{2} t^2 + a_0 t + v_0 \\ \frac{\alpha}{6} t^3 + \frac{\beta}{2} t^2 + \gamma t + a_0 \end{bmatrix}, \text{ 其中 } s(0) = (p_0, v_0, a_0) \text{ 为初始状态.}$$

如何求解 α, β, γ ? 把末状态 $s^*(T) = \text{期望状态} s_f = (p_f, v_f, a_f)$ 代入上式, 得到 $\begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} =$

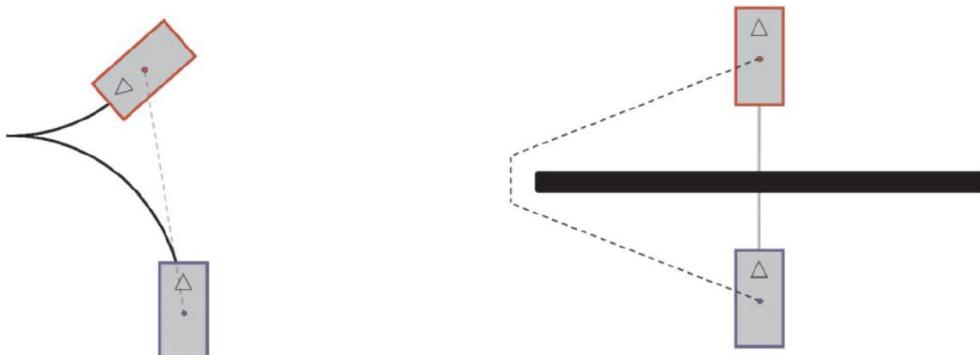
$$\begin{bmatrix} 720 & -360T & 60T^2 \\ -360T & 168T^2 & -24T^3 \\ 60T^2 & -24T^3 & 3T^4 \end{bmatrix} \begin{bmatrix} \Delta p \\ \Delta v \\ \Delta a \end{bmatrix}, \text{ 其中 } \begin{bmatrix} \Delta p \\ \Delta v \\ \Delta a \end{bmatrix} = \begin{bmatrix} p_f - p_0 - v_0 T - \frac{1}{2} a_0 T^2 \\ v_f - v_0 - a_0 T \\ a_f - a_0 \end{bmatrix}.$$

解得最小代价 $J = \frac{1}{T} \int_0^T j(t)^2 dt = \frac{1}{T} \int_0^T u^*(t)^2 dt = \gamma^2 + \beta \gamma T + \frac{1}{3} \beta^2 T^2 + \frac{1}{3} \alpha \gamma T^2 + \frac{1}{4} \alpha \beta T^3 + \frac{1}{20} \alpha^2 T^4$.

无论末状态 $s(T) = (p_T, v_T, a_T)$ 中的 3 个变量是全部给定还是部分给定(用边界条件), 都能求出相应的最优解.

4.7 启发式函数的设计

假设不存在障碍物 / 假设不存在动力学约束



4.8 通过 A*算法完成图搜索问题

对于每一个节点(状态), 将 OBVP 作为启发式函数 $h(n)$ 求解到规划目标状态, $g(n)$ 指累计代价 accumulate cost.

- Maintain a **priority queue** to store all the nodes to be expanded
- The heuristic function $h(n)$ for all nodes are pre-defined
- The priority queue is initialized with the start state X_s
- Assign $g(X_s)=0$, and $g(n)=\infty$ for all other nodes in the graph
- **Loop**
 - If the queue is empty, return FALSE; break;
 - Remove the node “ n ” with the lowest $f(n)=g(n)+h(n)$ from the priority queue
 - Mark node “ n ” as **expanded**
 - If the node “ n ” is the goal state, return TRUE; break;
 - For all **unexpanded** neighbors “ m ” of node “ n ”
 - If $g(m) = \infty$
 - $g(m) = g(n) + C_{nm}$
 - Push node “ m ” into the queue
 - If $g(m) > g(n) + C_{nm}$
 - $g(m) = g(n) + C_{nm}$
 - end
- End Loop

4.9 Frenet 坐标系

对于自动驾驶问题, 可分解为横向 lateral 和纵向 longitudinal 的规划问题.

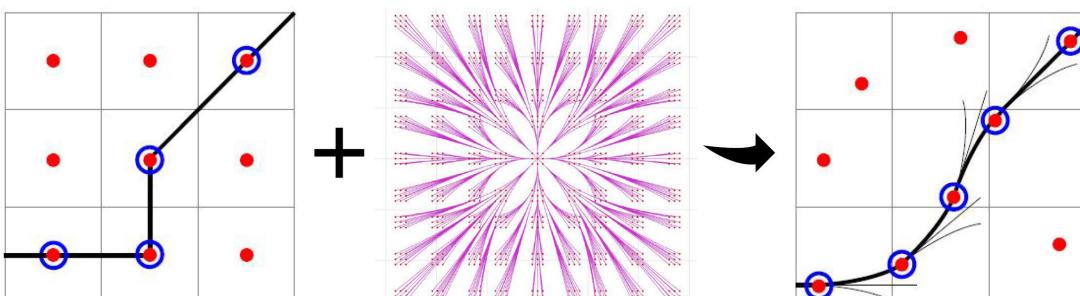
1. 运动/控制参数化: 五次多项式. 横向 $d(t) = a_{d0} + a_{d1}t + a_{d2}t^2 + a_{d3}t^3 + a_{d4}t^4 + a_{d5}t^5$ 和 纵向 $s(t) = a_{s0} + a_{s1}t + a_{s2}t^2 + a_{s3}t^3 + a_{s4}t^4 + a_{s5}t^5$.

以横向轨迹规划为例. 初始条件 $D(0) = (d_0, \dot{d}_0, \ddot{d}_0)$, 终止条件 $D(T) = (d_f, \dot{d}_f, \ddot{d}_f)$. 一般而言是希望横向的速度和加速度都为 0, 因此 $D(T) = (d_f, 0, 0)$.

4.10 Hybrid A*

A*: 在栅格地图中找一条 path(格子中心点的连线). + Lattice Planning: 构建非完整约束的搜索树.

对一个机器人模型, 把控制空间离散化, 如把控制量 $u \in [-u_{max}, u_{max}]$ 十等分, 然后分别在任务空间内前向积分, 得到一个稠密的 Lattice Graph. 在线生成的话需要耗费大量时间, 因此需要剪枝 prune 一些节点. 这种思想就是 Hybrid A*. (永远保持一个网格内只有一个节点)



workflow:

- Maintain a **priority queue** to store all the nodes to be expanded
- The heuristic function $h(n)$ for all nodes are pre-defined
- The priority queue is initialized with the start state X_s
- Assign $g(X_s)=0$, and $g(n)=\infty$ for all other nodes in the graph
- Loop
 - If the queue is empty, return FALSE; break;
 - Remove** the node "n" with the lowest $f(n)=g(n)+h(n)$ from the priority queue
 - Mark node "n" as **expanded**
 - If the node "n" is the goal state, return TRUE; break;
 - For all **unexpanded** neighbors "m" of node "n"
 - If $g(m) = \infty$
 - $g(m) = g(n) + C_{nm}$
 - Push node "m" into the queue
 - If $g(m) > g(n) + C_{nm}$
 - $g(m) = g(n) + C_{nm}$
 - end
- End Loop

如何设计合理的启发式函数?

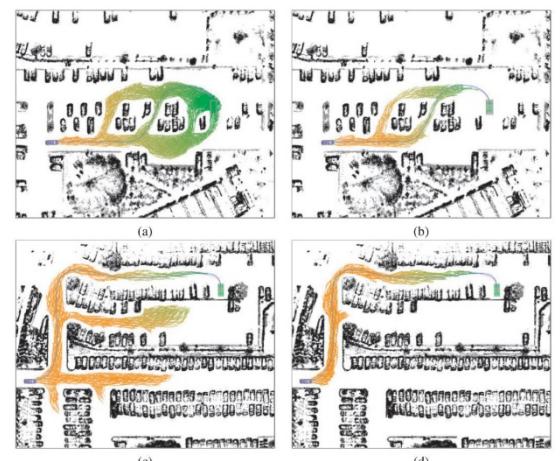
1. 2D 欧氏距离(很多搜索)
2. 不考虑障碍物但考虑动力学约束
3. 不考虑障碍物但考虑动力学约束, 走进了死胡同
4. 不考虑障碍物但考虑动力学约束 + 考虑障碍物但不考虑动力学约束(2D 最短路径)

选择合适的启发式函数

通过在节点中前向积分该状态来发现 neighbors

根据 m 节点中的状态

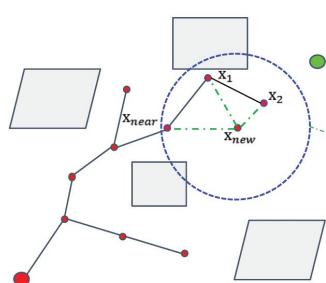
更新 m 节点中的状态



其他技巧: analytic expansions (one-shot): 搜索树运行过程中, 有一定概率当一个状态被生成后直接解到终点状态的理论上的最优控制. 如果这个路径满足动力学约束和障碍物, 就可结束搜索. 工程上中有一个概率, 比如说 $N=20$, 每搜索 20 次就运行一次 one-shot. 或 N 随 frontier 离终点的距离成正比.

4.11 Kinodynamic RRT*

和 RRT*区别在于选择父节点, Kinodynamic RRT*需要解一个 OBVP 问题.



Algorithm 2: RRT Algorithm

```

Input:  $\mathcal{M}, x_{init}, x_{goal}$ 
Result: A path  $\Gamma$  from  $x_{init}$  to  $x_{goal}$ 
 $\mathcal{T}.init();$ 
for  $i = 1$  to  $n$  do
   $x_{rand} \leftarrow Sample(\mathcal{M});$ 
   $x_{near} \leftarrow Near(x_{rand}, \mathcal{T});$ 
   $x_{new} \leftarrow Steer(x_{rand}, x_{near}, StepSize);$ 
  if CollisionFree( $x_{new}$ ) then
     $X_{near} \leftarrow NearC(\mathcal{T}, x_{new});$ 
     $x_{min} \leftarrow ChooseParent(X_{near}, x_{near}, x_{new});$ 
     $\mathcal{T}.addNodeEdge(x_{min}, x_{new});$ 
     $\mathcal{T}.rewire();$ 

```

Kinodynamic RRT*

Input: E, x_{init}, x_{goal}

Output: A trajectory T from x_{init} to x_{goal}

```

T.init();
for  $i = 1$  to  $n$  do
   $x_{rand} \leftarrow Sample(E);$ 
   $X_{near} \leftarrow Near(T, x_{rand});$ 
   $x_{min} \leftarrow ChooseParent(X_{near}, x_{near}, x_{rand});$ 
   $T.addNode(x_{rand});$ 
   $T.rewire();$ 

```

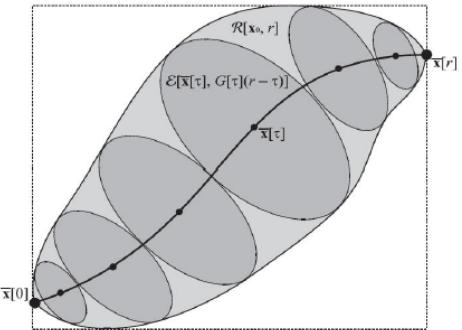
如何采样? Kinodynamic RRT*不光要采样位置, 还要采样速度和加速度.

如何定义"邻近"? 若不考虑运动学约束, 可用欧拉距离或曼哈顿距离. 若考虑, 则是一个最优控制问题. 可定义一个代价函数. 通常采用二次型的时间-能量最优解 $c[\pi] = \int_0^\tau (1 + u(t)^T R u(t)) dt$. $u(t)$ 相当于上文的 $j(t)$, R 是个权重项, 因为 cost function 中加了时间项 ($\int_0^\tau dt$). τ 是到达时间. 该式的含义是最小化能量消耗和时间的加权. 若从一个状态转移到另一个状态的代价较小, 则说明"邻近" (注意, 如果反向转移, 代价可能会不同).

如何选择父节点? 以一个状态为圆心, 一定代价 $c[\pi]$ 为半径找搜索树中已有的节点. 这些状态的集合称该状态的前向可达集 forward reachable set. 反之, 对特定的末状态, 所有可到达该状态的初始状态称该状态的后向可达集 backward reachable set.

如何有效率地找到邻近节点? 下图为高维空间中的前向可达集 (很多椭球组成). 椭球可用 AABB 法构成一个正方体的 bounding box, 再在高维 kd-tree 中进行近邻查询, 再找出最好的父节点.

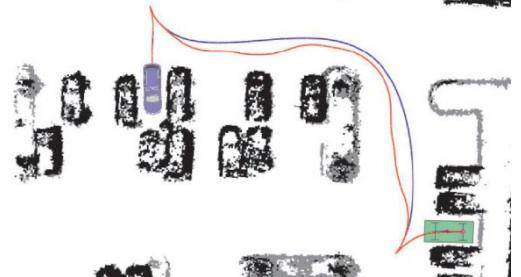
如何重连接? 计算 x_rand 的前向可达集, 再解 OBVP 问题.



5 Minimum Snap Trajectory Generation

Pipeline: 先生成一个路径(可以不光滑), 再把它从所在的空间转化到机器人的状态空间(光滑, 动力学可行).

为什么要光滑? 适合机器人自主运动. 速度/更高阶的动力学不能突变. 机器人不应该在转弯时停下来. 节约能量.



已经有了前端 front-end (path finding), 为什么需要后端 back-end (trajectory generation/optimization)? 因为可以提升 path 质量.

前端已经适用于动力学约束了, 为什么需要后端? 因为可以节约时间提高效率.

5.1 生成光滑的轨迹

边界条件: 起始点和目标点的位置(和方向).

中间条件: waypoint 位置(和方向). waypoints 可以通过路径规划找到(A*/RRT*/etc.).

光滑度标准: 一般转化为最小化输入量的变化率.

5.2 Differential Flatness

机器人(如四旋翼)的状态和输入量可被表示为数个(四个)精心选择的 flat 输出量以及它们的导数所构成的代数函数. 可被应用在轨迹的自动生成中.

在 flat 输出空间(有合适的边界导数)中的任一光滑轨迹可被

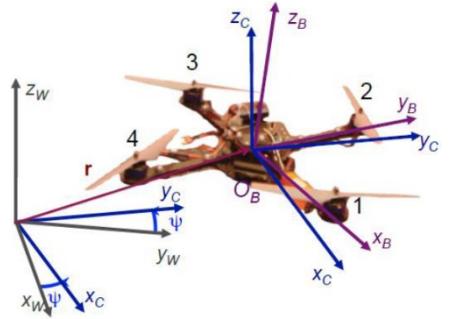
一个可能的选择为 $\sigma = [x, y, z, \psi]^T$, ψ 表示偏航角 yaw.

轨迹在 flat 输出空间中: $\sigma = [T_0, T_M] \rightarrow \mathbb{R}^3 \times SO(2)$. $T_0 \sim T_M$ 内, 轨迹是定义在 \mathbb{R}^3 欧式空间和 $SO(2)$ 旋转空间内的.

5.3 无人机状态

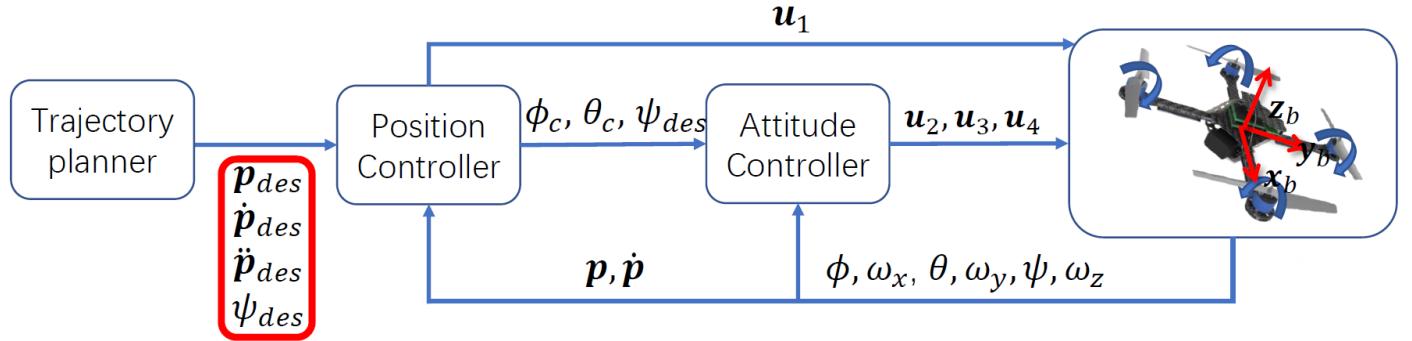
状态: 位置, 方向, 线速度, 角速度, 注意角速度的轴是 $x/y/z$ 轴.

$$\mathbf{X} = [x, y, z, \phi, \theta, \psi, \dot{x}, \dot{y}, \dot{z}, \omega_x, \omega_y, \omega_z]^T.$$



因为四旋翼是欠驱动的, 其俯仰和横滚角与加速度方向有关, 所以 flat outputs 为: $\sigma = [x, y, z, \psi]^T$. 位置, 速度和加速度可表示为 flat outputs 的导数.

规划-控制环路: (规划的是 $[p_{des}, \dot{p}_{des}, \ddot{p}_{des}, \psi_{des}]^T$, 用 x, y, z, ψ 表示)



令规划出的轨迹为一个多项式. 因为多项式可明确地表示出 flat outputs 空间内的轨迹. 优点有: 用多项式阶数简单地定义光滑度准则. 可计算导数的闭式解. 在三个维度上可解耦地生成轨迹.

5.4 Minimum Snap

使多段轨迹光滑: 使线段的折角光滑. 最好保持恒定的速度运动. 最好加速度为 0. 需要对较短的线段进行特殊处理.

用优化的方法计算出机器人到达每个折点时最佳的速度和加速度.

下表中每行的物理量都是一一对应的: snap 为 jerk 的导数, thrust 为推力.

Derivative	Translation	Rotation	Thrust
0	Position		
1	Velocity		
2	Acceleration	Rotation	
3	Jerk	Angular Velocity	Thrust
4	Snap	Angular Acceleration	Differential Thrust

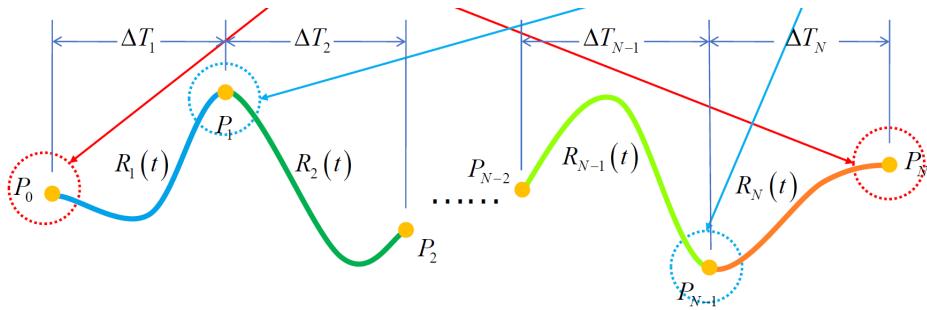
上表说明如果要最小化 jerk, 相当于最小化了角速度, 利于视觉追踪(视觉导航 VIO). 如果要最小化 snap, 相当于最小化了推力的导数, 使推力的变化尽可能平缓, 利于节约能量.

分段轨迹参数化:

- 每段都是一个多项式.
- 不需要保持阶数一致, 但是一致的阶数能简化问题.
- 每段的时间长度 T_0, T_1, T_2, \dots 必须是已知的. 这里的时间不能达到最优.

$$f(t) = \begin{cases} f_1(t) \doteq \sum_{i=0}^N p_{1,i} t^i & T_0 \leq t \leq T_1 \\ f_2(t) \doteq \sum_{i=0}^N p_{2,i} t^i & T_1 \leq t \leq T_2 \\ \vdots & \vdots \\ f_M(t) \doteq \sum_{i=0}^N p_{M,i} t^i & T_{M-1} \leq t \leq T_M \end{cases}$$

边界条件: 微分约束 $\begin{cases} f_0^{(k)}(T_0) = x_{0,0}^{(k)}, \\ f_N^{(k)}(T_N) = x_{T,N}^{(k)}, \end{cases}$, 连续性约束 $f_j^{(k)}(T_j) = f_{j+1}^{(k)}(T_j)$.



k 阶光滑意味着 $k+1$ 阶连续.

最小化 jerk 最少需要 5 阶多项式才能保证一段轨迹的光滑性. 因为 5 阶多项式有 6 个未知数(首末点的 $p/v/a$).

最小化 snap 最少需要 7 阶多项式才能保证一段轨迹的光滑性. 因为 7 阶多项式有 8 个未知数(首末点的 $p/v/a/j$).

对于 k 段轨迹, 最小化 jerk, 其约束数为 $3 + 3 + (k - 1) = k + 5$. 每段轨迹提供的未知数为 $(N + 1) * k$ 个. 令 $k + 5 = (N + 1) * k$, 得每段轨迹最少需要 $N = 5 / k$ 阶多项式. 段数越多, 需要的阶数就越少. 工程上一般按 $k = 1$ 来实现, 保证可靠性.

推荐自己写代码时, 把每段时间设为 $0 \sim T_0, 0 \sim T_1, 0 \sim T_2, \dots$, 而不是课件中的 $0 \sim T_0, 0 \sim T_1 - T_0, 0 \sim T_2 - T_1, \dots$

5.5 Minimum Snap Trajectory Generation

对于一个多项式段的代价函数:

$$\begin{aligned}
 f(t) &= \sum_i p_i t^i \\
 \Rightarrow f^{(4)}(t) &= \sum_{i \geq 4} i(i-1)(i-2)(i-3)t^{i-4} p_i \\
 \Rightarrow (f^{(4)}(t))^2 &= \sum_{i \geq 4, l \geq 4} i(i-1)(i-2)(i-3)l(l-1)(l-2)(l-3)t^{i+l-8} p_i p_l \\
 \Rightarrow J(T) &= \int_{T_{j-1}}^{T_j} (f^4(t))^2 dt = \sum_{i \geq 4, l \geq 4} \frac{i(i-1)(i-2)(i-3)j(l-1)(l-2)(l-3)}{i+l-7} (T_j^{i+l-7} - T_{j-1}^{i+l-7}) p_i p_l \\
 \Rightarrow J(T) &= \int_{T_{j-1}}^{T_j} (f^4(t))^2 dt \\
 &= \begin{bmatrix} \vdots \\ p_i \\ \vdots \end{bmatrix}^T \begin{bmatrix} \vdots & & & \\ \cdots & \frac{i(i-1)(i-2)(i-3)l(l-1)(l-2)(l-3)}{i+l-7} T_j^{i+l-7} & \cdots & \\ \vdots & & & \end{bmatrix} \begin{bmatrix} \vdots \\ p_l \\ \vdots \end{bmatrix} \\
 \Rightarrow J_j(T) &= \mathbf{p}_j^T \mathbf{Q}_j \mathbf{p}_j \quad \text{Minimize this!}
 \end{aligned}$$

\mathbf{Q}_j 称为 Hessian 矩阵(权重矩阵).

对于一个多项式段的导数约束: (也建模了 waypoint 约束, 即 0 阶导数)

$$\begin{aligned}
 f_j^{(k)}(T_j) &= x_j^{(k)} & x(t) &= p_5 t^5 + p_4 t^4 + p_3 t^3 + p_2 t^2 + p_1 t + p_0 \\
 \Rightarrow \sum_{i \geq k} \frac{i!}{(i-k)!} T_j^{i-k} p_{j,i} &= x_{T,j}^{(k)} & x(0) &= \dots, x(T) = \dots \\
 \Rightarrow \left[\dots \frac{i!}{(i-k)!} T_j^{i-k} \dots \right] \begin{bmatrix} \vdots \\ p_{j,i} \\ \vdots \end{bmatrix} &= x_{T,j}^{(k)} & \dot{x}(0) &= \dots, \dot{x}(T) = \dots \\
 \Rightarrow \left[\dots \frac{i!}{(i-k)!} T_{j-1}^{i-k} \dots \right] \begin{bmatrix} \vdots \\ p_{j,i} \\ \vdots \end{bmatrix} &= \begin{bmatrix} x_{0,j}^{(k)} \\ x_{T,j}^{(k)} \end{bmatrix} & \ddots & \\
 \Rightarrow \left[\dots \frac{i!}{(i-k)!} T_j^{i-k} \dots \right] \begin{bmatrix} \vdots \\ p_{j,i} \\ \vdots \end{bmatrix} &= \begin{bmatrix} x_{0,j}^{(k)} \\ x_{T,j}^{(k)} \end{bmatrix} & p_0 &= \dots, \\
 \Rightarrow \mathbf{A}_j \mathbf{p}_j &= \mathbf{d}_j & p_5 T^5 + p_4 T^4 + p_3 T^3 + p_2 T^2 + p_1 T + p_0 &= \dots \\
 && [T^5, T^4, T^3, T^2, T, 1] \begin{bmatrix} p_5 \\ p_4 \\ p_3 \\ p_2 \\ p_1 \\ p_0 \end{bmatrix} &= \dots
 \end{aligned}$$

对于两段之间的连续性约束: 左右极限一致(速度, 加速度是一样的).

$$\begin{aligned}
 f_j^{(k)}(T_j) &= f_{j+1}^{(k)}(T_j) \\
 \Rightarrow \sum_{i \geq k} \frac{i!}{(i-k)!} T_j^{i-k} p_{j,i} - \sum_{l \geq k} \frac{l!}{(l-k)!} T_j^{l-k} p_{j+1,l} &= 0 \\
 \Rightarrow \left[\dots \frac{i!}{(i-k)!} T_j^{i-k} \dots - \frac{l!}{(l-k)!} T_j^{l-k} \dots \right] \begin{bmatrix} \vdots \\ p_{j,i} \\ \vdots \\ p_{j+1,l} \\ \vdots \end{bmatrix} &= 0 \\
 \Rightarrow [\mathbf{A}_j \quad -\mathbf{A}_{j+1}] \begin{bmatrix} \mathbf{p}_j \\ \mathbf{p}_{j+1} \end{bmatrix} &= 0
 \end{aligned}$$

把所有约束整合到一起, 即称作有约束的二次优化问题(quadratic programming, QP 问题). 这是一种标准的凸优化 convex optimization 问题(即一定可以求解的问题).

$$\begin{aligned}
 \min \quad & \left[\begin{bmatrix} \mathbf{p}_1 \\ \vdots \\ \mathbf{p}_M \end{bmatrix} \right]^T \begin{bmatrix} \mathbf{Q}_1 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \ddots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{Q}_M \end{bmatrix} \begin{bmatrix} \mathbf{p}_1 \\ \vdots \\ \mathbf{p}_M \end{bmatrix} \\
 \text{s. t. } \quad & \mathbf{A}_{eq} \begin{bmatrix} \mathbf{p}_1 \\ \vdots \\ \mathbf{p}_M \end{bmatrix} = \mathbf{d}_{eq}
 \end{aligned}$$

凸优化问题的标准形式: 其中 f_0, f_1, \dots, f_m 是凸的, 等式约束 equality constraints 一定是 affine 的.

$$\begin{aligned}
 \underset{x}{\text{minimize}} \quad & f_0(x) \\
 \text{subject to} \quad & f_i(x) \leq 0 \quad i = 1, \dots, m \\
 & Ax = b
 \end{aligned}$$

凸优化问题的任何一个局部最优点都是全局最优的.

大多数问题最初用公式表达时都不是凸的, 但有些可以通过数学变化变成凸的.

5.6 Minimum Snap 的闭式解

上节中 $J = \left[\begin{bmatrix} \mathbf{p}_1 \\ \vdots \\ \mathbf{p}_M \end{bmatrix} \right]^T \begin{bmatrix} \mathbf{Q}_1 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \ddots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{Q}_M \end{bmatrix} \begin{bmatrix} \mathbf{p}_1 \\ \vdots \\ \mathbf{p}_M \end{bmatrix}$ 中的 $\mathbf{p}_1 \sim \mathbf{p}_M$ 没有实际物理含义, 对多项式轨迹进行直接优化

会带来数值上的不稳定性. 因此相比于优化分段端点导数, 改变变量的值是更好的, 即把优化 $\mathbf{p}_1 \sim \mathbf{p}_M$

转化为优化 waypoints 的速度和加速度(位置是给定的), 因为有明确的物理含义.

构建一个映射 $\mathbf{M}_j \mathbf{p}_j = \mathbf{d}_j$, 其中 \mathbf{M}_j 是多项式系数 \mathbf{p}_j 到 waypoints 的各个导数 \mathbf{d}_j 的映射矩阵.

$$J = \begin{bmatrix} \mathbf{p}_1 \\ \vdots \\ \mathbf{p}_M \end{bmatrix}^T \begin{bmatrix} \mathbf{Q}_1 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \ddots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{Q}_M \end{bmatrix} \begin{bmatrix} \mathbf{p}_1 \\ \vdots \\ \mathbf{p}_M \end{bmatrix} \quad + \quad \mathbf{M}_j \mathbf{p}_j = \mathbf{d}_j \quad \Rightarrow \quad J = \begin{bmatrix} \mathbf{d}_1 \\ \vdots \\ \mathbf{d}_M \end{bmatrix}^T \begin{bmatrix} \mathbf{M}_1 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \ddots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{M}_M \end{bmatrix}^{-T} \begin{bmatrix} \mathbf{Q}_1 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \ddots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{Q}_M \end{bmatrix} \begin{bmatrix} \mathbf{M}_1 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \ddots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{M}_M \end{bmatrix}^{-1} \begin{bmatrix} \mathbf{d}_1 \\ \vdots \\ \mathbf{d}_M \end{bmatrix}$$

例如多项式为 5 阶时,

$$\begin{aligned} x(t) &= p_5 t^5 + p_4 t^4 + p_3 t^3 + p_2 t^2 + p_1 t + p_0 \\ x'(t) &= 5p_5 t^4 + 4p_4 t^3 + 3p_3 t^2 + 2p_2 t + p_1 \\ x''(t) &= 20p_5 t^3 + 12p_4 t^2 + 6p_3 t + 2p_2 \end{aligned} \quad M = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 \\ T^5 & T^4 & T^3 & T^2 & T & 1 \\ 5T^4 & 4T^3 & 3T^2 & 2T & 1 & 0 \\ 20T^3 & 12T^2 & 6T & 2 & 0 & 0 \end{bmatrix}$$

5.7 分解 fixed 变量和 free 变量

用一个选择矩阵 \mathbf{C} 来分解 free 变量 \mathbf{d}_P (要优化的, waypoints 的高阶导数)和 constrained/fixed 变量 \mathbf{d}_F . Free 变量是指未指定导数, 仅受连续性约束的变量.

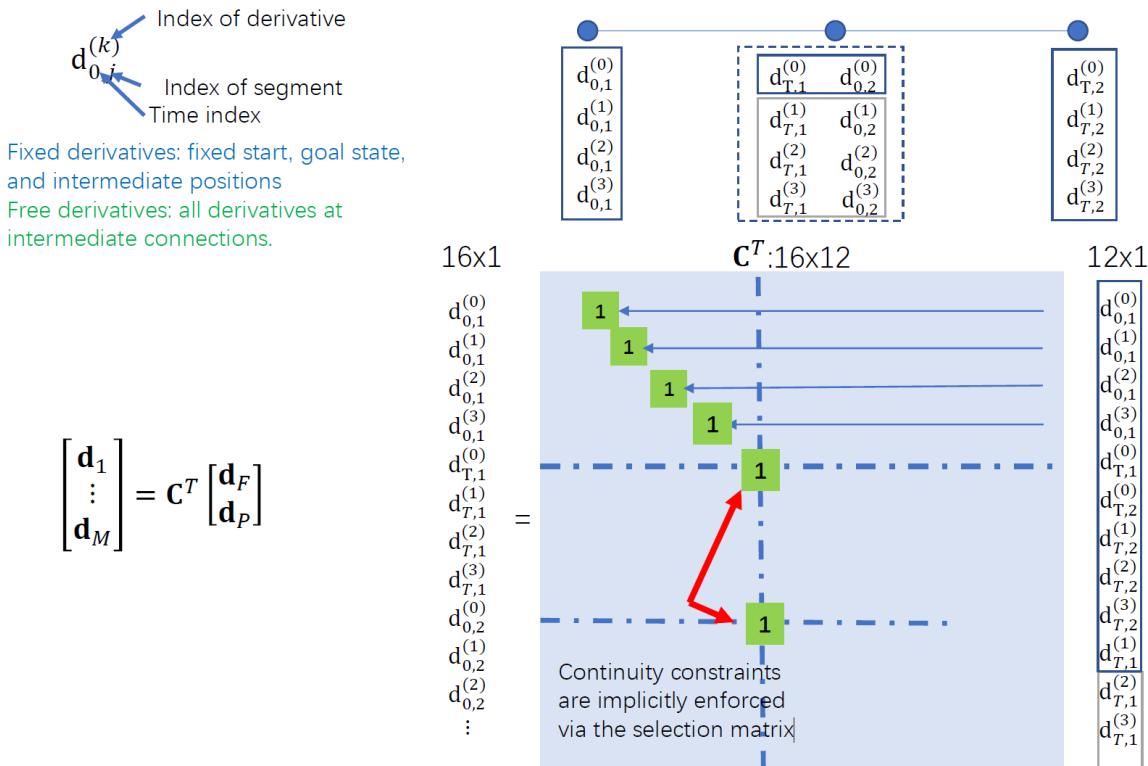
$$\mathbf{C}^T \begin{bmatrix} \mathbf{d}_F \\ \mathbf{d}_P \\ \vdots \\ \mathbf{d}_M \end{bmatrix} = \begin{bmatrix} \mathbf{d}_1 \\ \vdots \\ \mathbf{d}_M \end{bmatrix} \quad \Rightarrow \quad J = \begin{bmatrix} \mathbf{d}_F \\ \mathbf{d}_P \\ \vdots \\ \mathbf{d}_M \end{bmatrix}^T \underbrace{\mathbf{C} \mathbf{M}^{-T} \mathbf{Q} \mathbf{M}^{-1} \mathbf{C}^T}_{\mathbf{R}} \begin{bmatrix} \mathbf{d}_F \\ \mathbf{d}_P \\ \vdots \\ \mathbf{d}_M \end{bmatrix} = \begin{bmatrix} \mathbf{d}_F \\ \mathbf{d}_P \\ \vdots \\ \mathbf{d}_M \end{bmatrix}^T \begin{bmatrix} \mathbf{R}_{FF} & \mathbf{R}_{FP} \\ \mathbf{R}_{PF} & \mathbf{R}_{PP} \end{bmatrix} \begin{bmatrix} \mathbf{d}_F \\ \mathbf{d}_P \\ \vdots \\ \mathbf{d}_M \end{bmatrix}$$

通过上述方法转换成了无约束的 QP 问题, $J = \mathbf{d}_F^T \mathbf{R}_{FF} \mathbf{d}_F + \mathbf{d}_F^T \mathbf{R}_{FP} \mathbf{d}_P + \mathbf{d}_P^T \mathbf{R}_{PF} \mathbf{d}_F + \mathbf{d}_P^T \mathbf{R}_{PP} \mathbf{d}_P$. 对 \mathbf{d}_P 求偏导, 求出闭式解: $\mathbf{d}_P^* = -\mathbf{R}_{PP}^{-1} \mathbf{R}_{FP}^T \mathbf{d}_F$ (使 J 取得最优值时的 \mathbf{d}_P^*).

5.8 构建选择矩阵 \mathbf{C}

Fixed 导数: 固定起始, 目标状态和 waypoints 位置. Free 导数: 所有 waypoints 处的导数.

连续性约束通过选择矩阵隐含地强制执行.

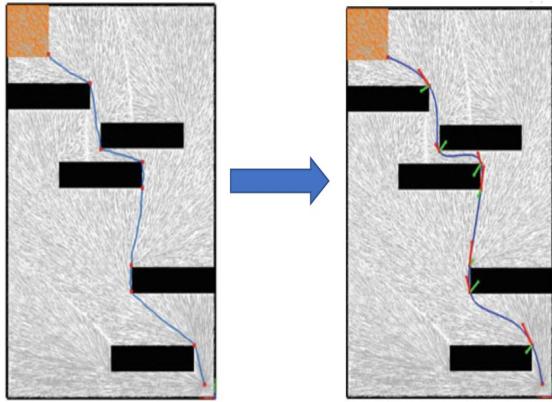


以上 2 种通过 \mathbf{p} 和通过 \mathbf{d} 求得最优的 J 的方法, 最终得到的优化轨迹是完全一样的.

5.9 Hierarchical approach

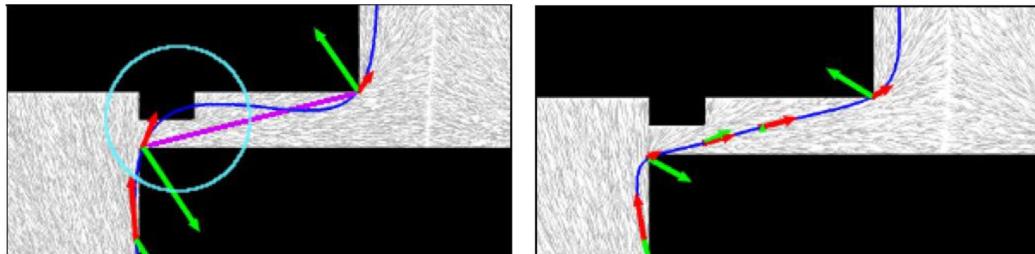
即 path planning + trajectory generation. 这种层级式的解法复杂度较低, 因为 path planning 是在较低维度的状态空间中完成的, 效率更高. 那么如何得到 waypoints 呢?

举例: 用 RRT*先生成出路径, 再把路标点记录下来, 作为 waypoints 用 minimum snap 生成轨迹.



5.10 安全性问题 + 迭代方法

trajectory generation 没有考虑避障的问题. 最初生成的 path 是无碰撞的, 若优化的 trajectory 碰撞了, 就在 path 中加一个 waypoint, 重新生成 trajectory. 若还是碰撞, 就在发生碰撞的 2 个 waypoints 之间再加一个 waypoint, 重新生成 trajectory...迭代式的生成无碰撞的 trajectory.



这种方法可能会迭代很多次, 也无法确保得到无碰撞的轨迹, 也会导致局部轨迹质量较差. 有没有更好的解决方法呢? 硬约束/软约束. 下节课详细说明.

5.11 工程应用的细节

图优化 solver: 很多现成的 solver 可以解决这些问题.

CVX: <http://cvxr.com/cvx/>. MatLab wrapper. Let you write down the convex program like mathematical equations, then call other solvers to solve the problem.

Mosek: <https://www.mosek.com/>. Very robust convex solvers, can solve almost all kinds of convex programs. Can apply free academic license. Only library available (x86).

OOQP: <http://pages.cs.wisc.edu/~swright/ooqp/>. Very fast, robust QP solver. Open sourced.

GLPK: <https://www.gnu.org/software/glpk/>. Very fast, robust LP solver. Open sourced.

数值稳定性.

时间归一化, 空间归一化. 把每段路径的时间和距离都放缩到相同尺度. 这 2 种方法可以相当高地提高数值稳定性.

闭式解方案是不是总是更好的？当矩阵很大时，求逆运算会耗费很多时间，因此数值凸优化 solver 更加鲁棒。

多项式能应用于所有场景吗？大多数如 $J = \int_0^T \rho_1 \cdot jerk^2(t) + \rho_2 \cdot snap^2(t) dt$, 即代价函数包含 2 个优化项，如 jerk 和 snap，会有细微差异，但也可应用。

时间分配。分段轨迹依赖于分段时间分配。时间分配显著影响最终轨迹。如何获得适当时间分配？

$$J_T = \left[\begin{matrix} \mathbf{p}_1 \\ \vdots \\ \mathbf{p}_M \end{matrix} \right]^T \begin{bmatrix} Q_1(T_1) & & \\ & \ddots & \\ & & Q_M(T_M) \end{bmatrix} \left[\begin{matrix} \mathbf{p}_1 \\ \vdots \\ \mathbf{p}_M \end{matrix} \right] + k_T \sum_{i=1}^M T_i$$

Penalize time duration in the overall cost

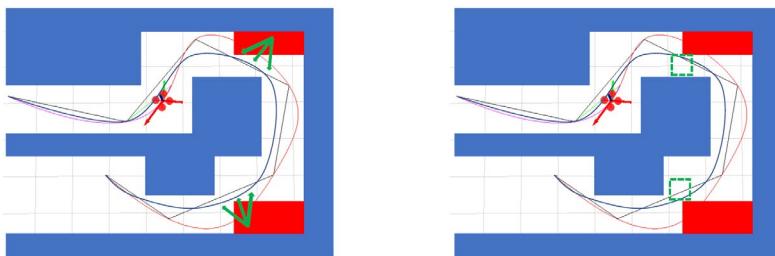
- Minimize this objective function
- Get the gradient to T numerically

6 软约束/硬约束限制下的轨迹优化

基础的 minimum snap 框架能生成光滑的曲线，但没有考虑避障。轨迹的"overshoot"不可避免。

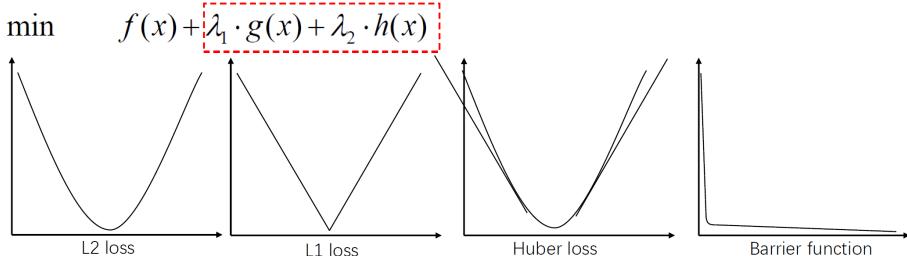
adding forces(软约束)

/ adding bounds(硬约束)



硬约束：边界条件必须严格满足。

软约束：把约束添加到代价函数中成为一个惩罚项 / loss function，并不要求严格满足。



6.1 硬约束优化 hard-constrained optimization

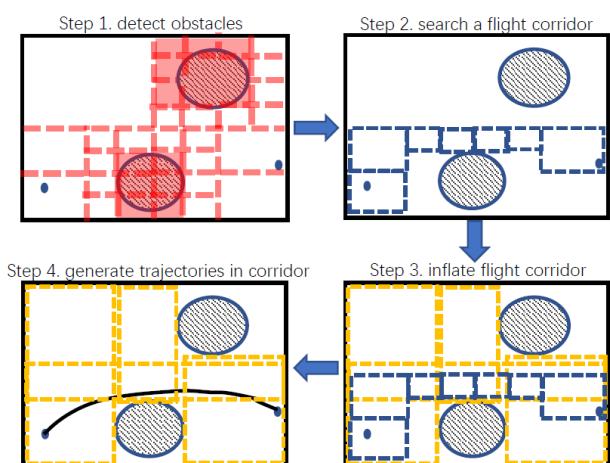
6.1.1 Corridor-based Trajectory Optimization

Step 1：把障碍物信息输入八叉树地图。

Step 2：用图搜索方法在八叉树地图中规划出一条无碰撞路径(不考虑动力学约束)。但是仍有很大的剩余空间。

Step 3：把走廊在地图中进行膨胀。

Step 4：剔除掉重复部分。在走廊中生成优化后的多项式轨迹。(和上节不用之处在于，这里没有 waypoints 了，只满足轨迹在走廊中即可 safety constraints)



- Differential flatness property
- $\{x, y, z, \dot{x}, \dot{y}, \dot{z}, \emptyset, \theta, \varphi, p, q, r\} \rightarrow \{x, y, z, \varphi\}$

- Piecewise polynomial trajectory

$$f_\mu(t) = \begin{cases} \sum_{j=0}^N p_{1j}(t - T_0)^j & T_0 \leq t \leq T_1 \\ \sum_{j=0}^N p_{2j}(t - T_1)^j & T_0 \leq t \leq T_1 \\ \vdots \\ \sum_{j=0}^N p_{Mj}(t - T_{M-1})^j & T_0 \leq t \leq T_1 \end{cases}$$

- Cost function (minimum jerk)

$$J = \sum_{\mu \in \{x, y, z\}} \int_0^T \left(\frac{d^k f_\mu(t)}{dt^k} \right)^2 dt$$



$$\begin{aligned} \min \quad & \mathbf{p}^T \mathbf{H} \mathbf{p} \\ \text{s.t.} \quad & \mathbf{A}_{eq} \mathbf{p} = \mathbf{b}_{eq} \\ & \mathbf{A}_{lq} \mathbf{p} \leq \mathbf{b}_{lq} \end{aligned}$$

Quadratic Program

- Boundary constrains
- Continuity constrains
- Safety constrains

为什么走廊里每一个 cube 都是方块? x/y/z 轴都是相互垂直的, 用方块可以在膨胀走廊时更加便捷. 并且凸的形状可以转化为凸优化问题.

包含的约束:

Instant linear constraints:

起点和终点的状态约束. $\mathbf{Ap} = \mathbf{b}$

Transition point 约束(段与段的连接点 waypoints 约束在 2 个膨胀后的 cube 相交的区域, 这是为了一定程度上(不是 100%)保证轨迹上的所有点都在 cube 内). $\mathbf{Ap} = \mathbf{b}, \mathbf{Ap} \leq \mathbf{b}$

连续性约束. $\mathbf{Ap}_i = \mathbf{Ap}_{i+1}$

Interval linear constraints: (每个点都检测的话太消耗计算资源了, 采用后验方法)

边界约束(轨迹不能越出 cube). $\mathbf{A}(t)\mathbf{p} \leq \mathbf{b}, \forall t \in [t_l, t_r]$

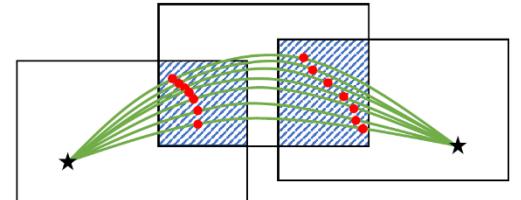
动力学约束(速度/加速度约束). $\mathbf{A}(t)\mathbf{p} \leq \mathbf{b}, \forall t \in [t_l, t_r]$

优点:

有效率: 在 reduced graph 中搜索路径, 在走廊上做凸优化.

高质量: 走廊提供了较大的优化自由度.

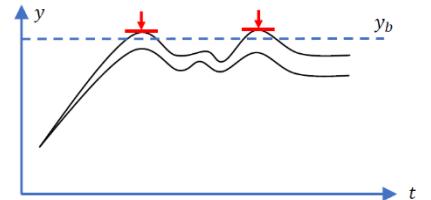
QP 问题: 可求出全局最优解.



缺点:

边界约束和动力学约束不好计算. 策略:

检查给定约束下轨迹的极值. 若极值超过了一个边界 y_b , 就把约束收紧一点.



Corridor-based Trajectory Optimization 的问题在于施加全局的/安全的/动力学可行的约束不是很方便. 一般是把这些约束施加在轨迹给定的若干点上, 然后求出轨迹后, 求出极值, 检查极值是否违反约束. 若违反了, 再对该极值点施加额外的约束. 如此迭代的求解. 缺点是可能需要把轨迹求解很多次, 并且如果问题本身无解也需要多次迭代才被发现.

6.1.2 Bezier 曲线优化

用 Bernstein 多项式替换常规多项式的基底.

$$P_j(t) = p_j^0 + p_j^1 t + p_j^2 t^2 + \dots + p_j^n t^n$$

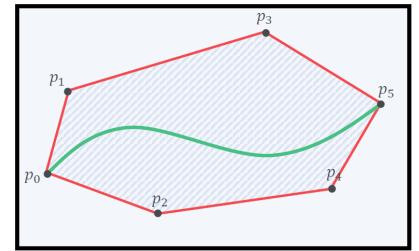
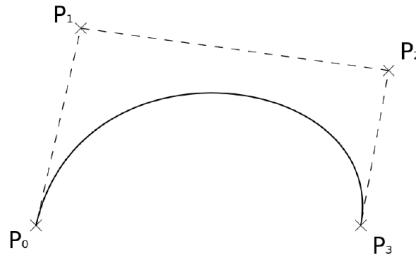


$$\begin{aligned} B_j(t) &= c_j^0 b_n^0(t) + c_j^1 b_n^1(t) + \dots + c_j^n b_n^n(t) = \sum_{i=0}^n c_j^i b_n^i(t) \\ b_n^i(t) &= \binom{n}{i} \cdot t^i \cdot (1-t)^{n-i} \end{aligned}$$

其中多项式系数 c_j^i 称为控制点. Bernstein 多项式与常规多项式通过 $p = \mathbf{M} \cdot c$ 建立一一映射关系. 如六

阶多项式的变换矩阵为

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -6 & 6 & 0 & 0 & 0 & 0 & 0 \\ 15 & -30 & 15 & 0 & 0 & 0 & 0 \\ -20 & 60 & -60 & 20 & 0 & 0 & 0 \\ 15 & -60 & 90 & -60 & 15 & 0 & 0 \\ -6 & 30 & -60 & 60 & -30 & 6 & 0 \\ 1 & -6 & 15 & -20 & 15 & -6 & 1 \end{bmatrix}$$



性质:

终点插值 endpoint interpolation. Bezier 曲线总是起始于第一个控制点, 终止与最后一个控制点, 而不经过任何一个中间控制点.

凸包 convex hull. 一段 Bezier 曲线被完全限制在由其控制点 c^i 组成的凸包中.

Hodograph. Bezier 曲线 $B(t)$ 的导数曲线 $B'(t)$ 被称为 hodograph, 它也是一条 Bezier 曲线, 控制点为 $n \cdot (c^{i+1} - c^i)$, 其中 n 为原曲线阶数.

固定的时间间隔. Bezier 曲线总是定义在 $[0, 1]$ 上. (不灵活的, 负面影响)

凸包性质: 走廊由多个凸多边形组成. 每个 cube 对应一条 Bezier 曲线. 此曲线的控制点被限制在多边形内. Bezier 曲线轨迹完全在凸包内. 因此轨迹完全被限制在了走廊内.

由 Hodograph 性质, 速度和加速度曲线也可以被限制在凸包中.

轨迹生成表达式.

Define higher order (l^{th}) control points:

$$a_{\mu j}^{0,i} = c_{\mu j}^i, a_{\mu j}^{l,i} = \frac{n!}{(n-l)!} \cdot (a_{\mu j}^{l-1,i+1} - a_{\mu j}^{l-1,i}), \quad l \geq 1$$

- Boundary Constraints:

$$a_{\mu j}^{l,0} \cdot s_j^{(1-l)} = d_{\mu j}^{(l)}$$

- Continuity Constraints:

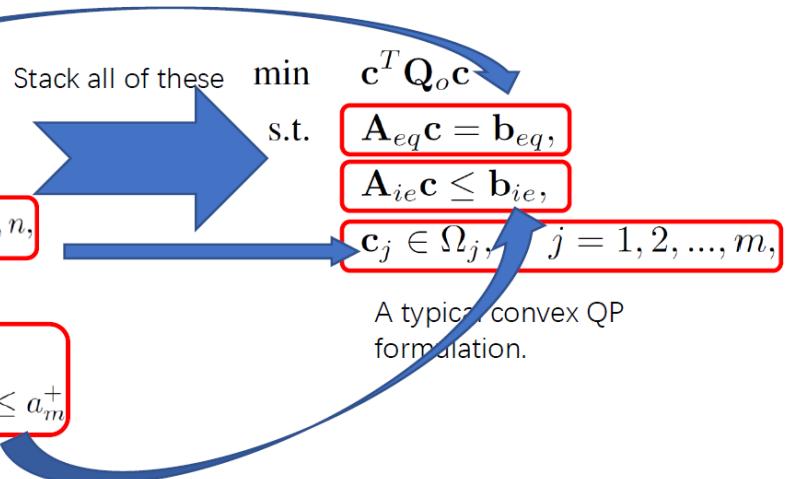
$$a_{\mu j}^{\phi,n} \cdot s_j^{(1-\phi)} = a_{\mu,j+1}^{\phi,0} \cdot s_{j+1}^{(1-\phi)}, \quad a_{\mu j}^{0,i} = c_{\mu j}^i.$$

- Safety Constraints:

$$\beta_{\mu j}^- \leq c_{\mu j}^i \leq \beta_{\mu j}^+, \quad \mu \in \{x, y, z\}, \quad i = 0, 1, 2, \dots, n,$$

- Dynamical Feasibility Constraints:

$$\begin{aligned} v_m^- \leq n \cdot (c_{\mu j}^i - c_{\mu j}^{i-1}) &\leq v_m^+, \\ a_m^- \leq n \cdot (n-1) \cdot (c_{\mu j}^i - 2c_{\mu j}^{i-1} + c_{\mu j}^{i-2}) / s_j &\leq a_m^+ \end{aligned}$$



只需要解一次即可确定是否存在一个合格的轨迹.

6.1.3 其他选择

Dense constraints: 在离散时间点添加大量约束. Piecewise constant accelerations at each tick. 用 QP 解决. 缺点: 总是产生过于保守的轨迹. 约束太多, 计算量大.

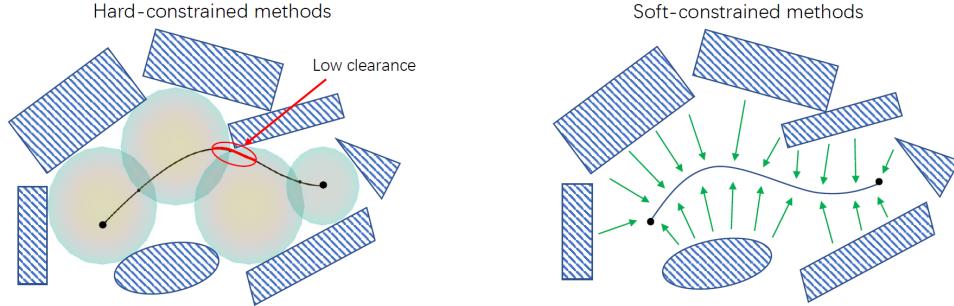
Mixed integer optimization: mixed-integer QP. 大'M'方法. 运行超慢.

6.2 软约束优化 soft-constrained optimization - 基于距离的轨迹优化

若机器人到障碍物的距离很近，则受到斥力。

硬约束方法等价地看待所有的环境。因此有可能生成的轨迹离障碍物距离很近。当有控制和反馈误差时，机器人可能会产生碰撞。所以硬约束方法对噪声是很敏感的。

基于视觉的无人机：感知范围和质量有限。



软约束优化：把路径更自然地规划在环境中，但十分依赖于代价函数的设计，即不一定会使得某一个约束被满足。

6.2.1 Differential flatness property: $\{x, y, z, \dot{x}, \dot{y}, \dot{z}, \emptyset, \theta, \varphi, p, q, r\} \rightarrow \{x, y, z, \varphi\}$

分段多项式轨迹：

$$f_\mu(t) = \begin{cases} \sum_{j=0}^N p_{1j}(t - T_0)^j & T_0 \leq t \leq T_1 \\ \sum_{j=0}^N p_{2j}(t - T_1)^j & T_0 \leq t \leq T_1 \\ \vdots & \vdots \\ \sum_{j=0}^N p_{Mj}(t - T_{M-1})^j & T_0 \leq t \leq T_1 \end{cases}$$

目标函数: $J = J_s + J_c + J_d = \lambda_1 J_1 + \lambda_2 J_2 + \lambda_3 J_3$

Smoothness cost Collision cost Dynamical cost

其中：

Smoothness cost 项: 最小化 snap. $J_s = \sum_{\mu \in \{x, y, z\}} \int_0^T \left(\frac{d^k f_\mu(t)}{dt^k} \right)^2 dt = [\mathbf{d}_F]^T \mathbf{C}^T \mathbf{M}^{-T} \mathbf{Q} \mathbf{M}^{-1} \mathbf{C} [\mathbf{d}_F] = [\mathbf{d}_F]^T [\mathbf{R}_{FF} \quad \mathbf{R}_{FP}] [\mathbf{d}_F]$

(注意：上文的 minimum snap 中 waypoints 的位置是固定的，这里的 \mathbf{d}_P 则包含了位置信息。)

对 free 导数 $\mathbf{d}_{p\mu}$ 求偏导，Jacobian 为 $\frac{\partial J_s}{\partial \mathbf{d}_{p\mu}} = 2\mathbf{d}_F^T \mathbf{R}_{FP} + 2\mathbf{d}_P^T \mathbf{R}_{PP}$

Collision cost 项: 到最近障碍物距离的惩罚. $J_c = \int_{T_0}^{T_M} c(p(t)) ds = \sum_{k=0}^{T/\delta t} c(p(T_k)) \|v(t)\| \delta t, T_k = T_0 + k\delta t$

(其中 $p(t)$ 表示 t 时刻机器人的位置， $c(p(t))$ 指该点的距离惩罚。这里是 ds 而不是 dt ，因为要使轨迹远离障碍物而不是经过障碍物的时间变短。)

对 free 导数 $\mathbf{d}_{p\mu}$ 求偏导，Jacobian 为 $\frac{\partial J_c}{\partial \mathbf{d}_{p\mu}} = \sum_{k=0}^{T/\delta t} \left\{ \nabla_\mu c(p(T_k)) \|v\| \mathbf{F} + c(p(T_k)) \frac{v_\mu}{\|v\|} \mathbf{G} \right\} \delta t, \mu \in \{x, y, z\}$

其中 $\mathbf{F} = \mathbf{T} \mathbf{L}_{dp}$, $\mathbf{G} = \mathbf{T} \mathbf{V}_m \mathbf{L}_{dp}$. \mathbf{L}_{dp} 是矩阵的右块 $\mathbf{M}^{-1} \mathbf{C}$ ，因为其对应了 μ 轴上的 free 导数 $\mathbf{d}_{p\mu}$ 。

$\nabla_\mu c(\cdot)$ 是 Collision cost 项沿着 μ 轴方向上的梯度。

V_m 将位置系数映射为速度系数。

$$\mathbf{T} = [T_k^0, T_k^1, \dots, T_k^n].$$

$$\mathbf{H}_o = \left[\frac{\partial^2 f_o}{\partial \mathbf{d}_{P_x}^2}, \frac{\partial^2 f_o}{\partial \mathbf{d}_{P_y}^2}, \frac{\partial^2 f_o}{\partial \mathbf{d}_{P_z}^2} \right]$$

二阶导得到 Hessian 矩阵

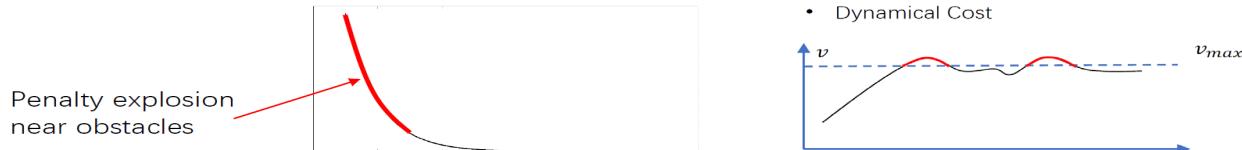
$$\text{其中 } \frac{\partial^2 f_o}{\partial \mathbf{d}_{P_\mu}^2} = \sum_{k=0}^{\tau/\delta t} \left\{ \mathbf{F}^T \nabla_\mu c(p(\mathcal{T}_k)) \frac{v_\mu}{\|v\|} \mathbf{G} + \mathbf{F}^T \nabla_\mu^2 c(p(\mathcal{T}_k)) \|v\| \mathbf{F} + \mathbf{G}^T \nabla_\mu c(p(\mathcal{T}_k)) \frac{v_\mu}{\|v\|} \mathbf{F} + \mathbf{G}^T c(p(\mathcal{T}_k)) \frac{v_\mu^2}{\|v\|^3} \mathbf{G} \right\} \delta t$$

Dynamical cost 项: 对超过极限的速度和加速度进行惩罚(类似于碰撞项).

这不是一个凸优化问题, 因为 ESDF 地图不一定是凸的. 所以只能用非线性优化的方法. 因此先把目标函数的导数求出.

6.2.2 ESDF: 欧氏有符号的距离场 Euclidean signed distance field.

用指数函数作为代价函数 c , 防止轨迹接近障碍物. 对 dynamical cost 同理.



6.2.3 数值优化方法

一阶方法 first-order method: 梯度下降法.

General descent method with $\Delta x = -\nabla f(x)$

given a starting point $x \in \text{dom } f$.

Repeat

1. $\Delta x := -\nabla f(x)$.
2. Line search. Choose step size t via exact or backtracking line search.
3. Update. $x := x + t\Delta x$.

until stopping criterion is satisfied.

- Stopping criterion usually of the form $\|\nabla f(x)\|_2 \leq \epsilon$

二阶方法 second-order method: 牛顿法, LM 法.

Newton method

Taylor expansion at $x^{(k)}$, and second – order similarity $\Delta x = -\nabla^2 f(x)^{-1} \nabla f(x)$

given a starting point $x \in \text{dom } f$.

Repeat

1. $\Delta x := -\nabla^2 f(x)^{-1} \nabla f(x)$.
2. Line search. $t = \text{argmin}_{t>0} f(x^{(k)} - t\nabla^2 f(x)^{-1} \nabla f(x))$.
3. Update. $x := x + t\Delta x$.

until stopping criterion is satisfied.

- Stopping criterion usually of the form $\|\nabla f(x)\|_2 \leq \epsilon$

- $\nabla^2 f(x)$ is the Hessian matrix of the $f(x)$ at x

- For Gauss-newton: $\nabla^2 f(x) \approx J_f^T J_f$, J_f is Jacobi matrix, $\Delta x = -(J_f^T J_f)^{-1} J_f^T$

已经有很多成熟的软件包可以运用: Ceres / NLOpt.

6.2.4 规划策略

Receding horizon re-planning

前端在导航开始处搜索, 规划出全局路径(黑色折线).

由于视野限制, 前端查找局部路径.

使用后端生成局部轨迹(蓝色).

沿着执行范围内的轨迹(红线)运动, 之后再进行下一次规划.

好处在于若机器人一直以高频的速度做轨迹的重规划, 这样可提高一致性.

Levenberg-Marquardt method

Improvement of Gauss-newton method: $\Delta x = -(J_f^T J_f + \lambda I)^{-1} J_f^T$

given a starting point $x \in \text{dom } f$, start $\lambda_0 > 0$.

Repeat

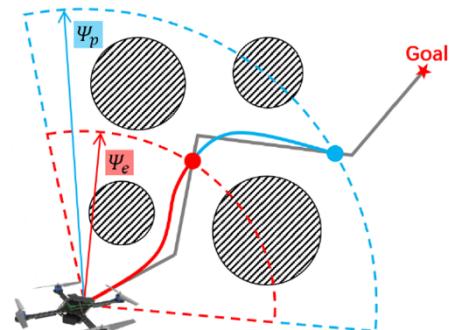
1. $\Delta x = -(J_f^T J_f + \lambda I)^{-1} J_f^T$.
2. Update. λ , the updating is controlled by the gain ratio
3. Update. $x := x + \Delta x$.

until stopping criterion is satisfied.

- Stopping criterion usually of the form $\|\nabla f(x)\|_2 \leq \epsilon$

- When $\lambda \rightarrow 0$, LM method \rightarrow Gauss-newton method

- When $\lambda \rightarrow \infty$, LM method \rightarrow Gradient descent method



ψ_p : planning horizon ψ_e : execution horizon

Exploration strategy

当建图 mapping 模块中的异常值(噪声, 分辨率)没有探索到路径时, 激活该策略.
在附近区域生成安全但较短的轨迹. 悬停并观察.

基于梯度优化的策略 - 初值的选择

若初始化为 minimum snap 的轨迹. 好处在于有良好的平滑度. 但可能会由于 overshoot 造成碰撞. 这是一个不安全的初始值.

若初始化为优化前沿着路径的直线轨迹. 其平滑度差, 但无碰撞, 是一个安全的初始值.

工程实现时一般认为安全性是最重要的. 因此采用第二种初始值. 给定无碰撞初始轨迹, 通过使用 collision cost 的指数型惩罚函数实现安全性.

6.2.5 两步优化策略

第一步时只优化 $J = J_s + J_c + J_d$ 中的碰撞代价 J_c . 先尽可能地让轨迹远离障碍物. 仅以 J_c 优化轨迹.

第二步再重新分配时间并重新参数化轨迹, 通过增加光滑度项 J_s 和动态惩罚项 J_d 对目标进行优化.

7 基于 MDP 的规划

7.1 规划中的不确定性 uncertainties

到目前为止, 规划都是在理想情况下进行的. 即 完美的动作执行 和 完备的状态估计 . 但在实际场景中会存在不确定性.

动作执行不确定性: 打滑, 地形不平, 风, 空气阻力, 控制误差等.

状态估计不确定性: 传感器噪声, 校准误差, 不完全估计, 部分可观测性等.

不确定性可分为 2 类.

非决定性 Nondeterministic: 机器人不知道下一时刻其行为会受到什么类型的不确定性或干扰.

概率性 Probabilistic: 机器人可通过观察和收集统计数据来估计执行动作后大致受到的干扰的程度.

在之前的规划中通常只有机器人一个参与者. 现在我们首先引入两个决策者 decision makers 对不确定性的产生进行建模, 然后引入不确定性规划的类型.

机器人是主要的决策者, 在完全已知状态和完美执行的基础上执行规划.

Nature 给机器人规划的执行增加了不确定性, 这对机器人来说是不可预测的.

定义 1: A Game Against Nature (Independent Game), U 和 Θ 相互独立.

定义一个非空集合 U 称为机器人的动作空间. 每个 $u \in U$ 被称为机器人的动作.

定义一个非空集合 Θ 称为自然动作空间. 每个 $\theta \in \Theta$ 被称为自然的动作.

定义函数 $L: U \times \Theta \rightarrow \mathbb{R} \cup \{\infty\}$ 称为成本函数 cost function, 或负奖励函数 negative reward function. 用来衡量机器人动作和自然动作相互作用的结果. 通过优化 L 来为机器人找到最优的动作 u .

定义 2: Nature Knows the Robot Action (Dependent Game), Θ 依赖于 U .

定义一个非空集合 U 称为机器人的动作空间. 每个 $u \in U$ 被称为机器人的动作.

对于每个机器人动作 $u \in U$, 定义一个非空集合 $\Theta(u)$ 称为自然动作空间.

定义函数 $L: U \times \Theta \rightarrow \mathbb{R} \cup \{\infty\}$ 称为成本函数 cost function, 或负奖励函数 negative reward function.

针对以上 2 种对不确定性的模型和定义，也有 2 种不同的解法.

解法 1: One-step Worst-Case Analysis.

在 Nondeterministic 模型下, Independent Game 中的 $P(\theta)$ 和 Dependent Game 中的 $P(\theta|u_k)$ 是未知的. 机器人无法预测自然的行为, 假定它恶意地选择了使成本尽可能高的行为(即对机器人最不利的行为).

因此, 做出假定是最坏的决策是合理的. 即 $u^* = \arg \min_{u \in U} \left\{ \max_{\theta \in \Theta} L(u, \theta) \right\}$.

解法 2: One-step Expected-Case Analysis.

在 Probabilistic 模型下, Independent Game 中的 $P(\theta)$ 和 Dependent Game 中的 $P(\theta|u_k)$ 是已知的. 尽管不能精准地知道自然下一步要做什么动作, 但我们知道其服从一定的分布.

假设自然动作已被观察到, 自然在动作的选择中应用了随机策略.

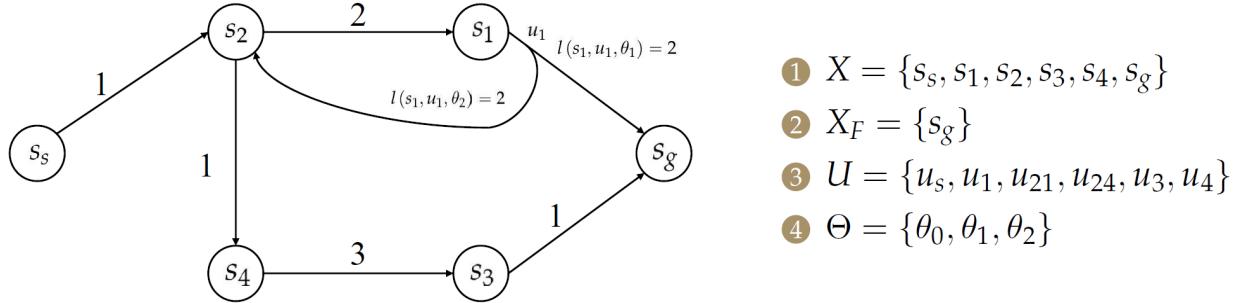
因此, 我们要优化 the average cost to be received. 即 $u^* = \arg \min_{u \in U} \{E_\theta [L(u, \theta)]\}$.

以上是 one-step. 对于 multi-step, 有以下定义.

定义: Multi-Step Discrete Planning with Nature.

1 一个非空的状态空间 X , 初始状态为 s_s , 目标集合为 $X_F \subset X$.

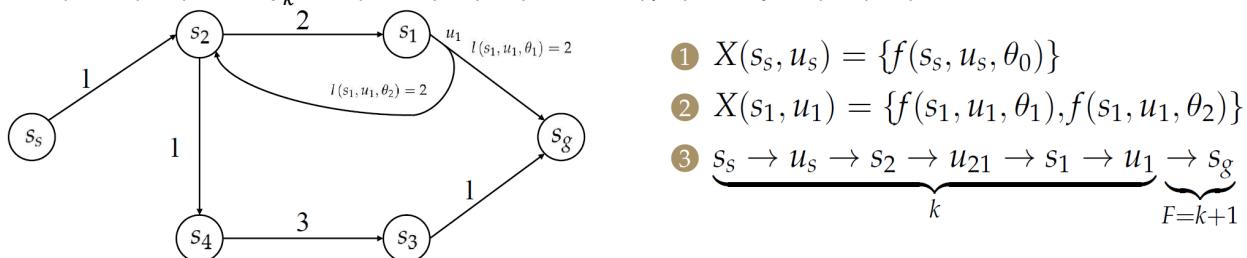
2 对于每个状态 $x \in X$, 有一个有限的非空的机器人动作空间 $U(x)$. 对于每个 $x \in X$ 和 $u \in U(x)$, 有一个有限的非空的自然动作空间 $\Theta(x, u)$.



3 对于每个 $x \in X$, $u \in U(x)$ 和 $\theta \in \Theta(x, u)$, 定义一个状态转移函数 $f(x, u, \theta)$. 即:

$$X_{k+1}(x_k, u_k) = \{x_{k+1} \in X \mid \exists \theta_k \in \Theta(x_k, u_k) \text{ s.t. } x_{k+1} = f(x_k, u_k, \theta_k)\}.$$

$$P(x_{k+1}|x_k, u_k) = \sum_{\theta_k} P(x_{k+1}, \theta_k|x_k, u_k) \text{ s.t. } \{\theta_k | x_{k+1} = f(x_k, u_k, \theta_k)\}.$$



4 一个阶段 stage 的集合, 说明机器人的动作由连续的 stages 组成. 每个阶段用 k 表示, 从 $k = 1$ 开始无限期地继续或在最大阶段 $k = K + 1 = F$ 结束.

5 一个 stage-additive cost functional L . \tilde{x}_F 表示状态到阶段 K 的历史, \tilde{u}_k 表示机器人动作到阶段 K 的历史, $\tilde{\theta}_K$ 表示自然动作到阶段 K 的历史: $\tilde{x}_F = (x_1, x_2, \dots, x_F)$, $\tilde{u}_k = (u_1, u_2, \dots, u_k)$, $\tilde{\theta}_K = (\theta_1, \theta_2, \dots, \theta_K)$. 其中 cost functional 是评估所有可能的规划(或路径)的指标: $L(\tilde{x}_F, \tilde{u}_k, \tilde{\theta}_K) = \sum_{k=1}^K l(x_k, u_k, \theta_k) + l_F(x_F)$,

$$l_F(x_F) = \begin{cases} 0, & \text{if } x_F \in X_G, \\ \infty, & \text{otherwise.} \end{cases}$$

7.2 Markov Decision Process (MDP)

事实上, 上述定义的标准形式是 Markov Decision Process (MDP).

MDP 在 learning field 是 (S, A, P, R) , 在 planning field 是 (X, U, P, L) . 其中:

S 或 X 是状态空间, A 或 U 是机器人的动作空间.

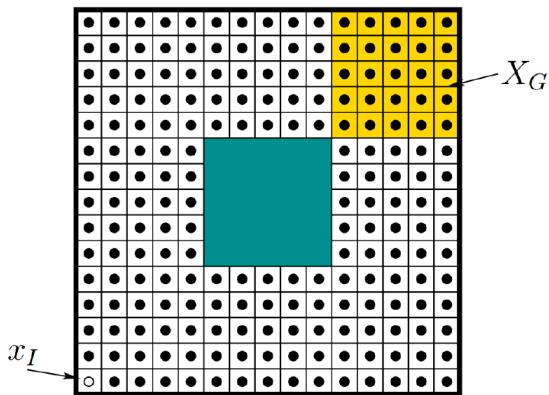
$P(x_{k+1}|x_k, u_k)$ 是 probabilistic 模型下的状态转移函数, 在 nondeterministic 模型下会退化为集合 $X_{k+1}(x_k, u_k)$.

因为动作 u 和 θ 而从状态 x_k 转移到 x_{k+1} , $R(x_k, x_{k+1})$ 称即时奖励 immediate reward, 或称负的一步成本 negative one-step cost $-l(x_k, u_k, \theta_k)$.

不确定规划的第一个困难在于如何用 MDP 模型恰当地形式化我们的问题.

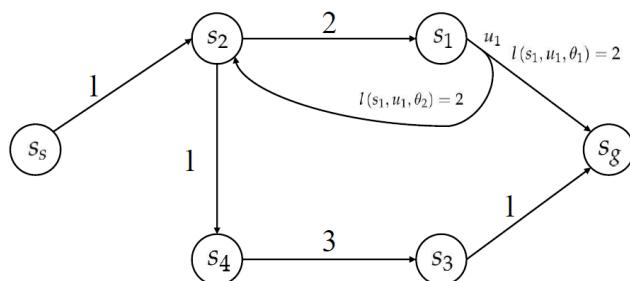
例子: 3 为连续的建模方式, 把机器人动作的执行建模成一个高斯分布. 4 为离散的建模方式, 机器人在自身动作以外随机加上一种自然动作.

A grid-based shortest path problem with interference from nature.



- ① $X = \{(i, j) \mid 1 \leq i, j \leq 15\}$
- ② $U = \{u_{stay}, u_{right}, u_{up}, u_{left}, u_{down}\}.$
- ③ $\Theta = \{\theta_1, \theta_2\}$
 $f(x_{k+1}, x_k, u_k) = \mathcal{N}(x_k + u_k, \sigma(\theta_1, \theta_2))$
- ④ $\Theta = \{\theta_0 = [0, 0]^T, \theta_1 = [0, 1]^T, \theta_2 = [0, -1]^T\}$
 $x_{k+1} = x_k + u_k + \theta_k, k \sim \{0, 1, 2\}$
- ⑤ $l(x_k, u_k, \theta_k) = \|x_{k+1} - x_k\|$

定义一个 plan (或反馈): $\pi: X \rightarrow U$, 是从状态空间到动作空间的映射. 它规定了在什么状态下应该执行什么动作. 定义从起点 x_s 的由 π 触发的所有轨迹的集合为 $\mathcal{H}(\pi, x_s)$. 每一条轨迹或每一次执行表示为 $(\tilde{x}, \tilde{u}, \tilde{\theta}) \in \mathcal{H}(\pi, x_s)$.



- ① $\mathcal{H}(\pi_1, s_s) :$
 $s_s \rightarrow u_s \rightarrow s_2 \rightarrow u_{21} \rightarrow s_1 \rightarrow u_1 \rightarrow s_g$
 $s_s \rightarrow u_s \rightarrow s_2 \rightarrow u_{21} \rightarrow s_1 \rightarrow u_1 \rightarrow s_2 \rightarrow \dots$
- ② $\mathcal{H}(\pi_2, s_s) :$
 $s_s \rightarrow u_s \rightarrow s_2 \rightarrow u_{24} \rightarrow s_4 \rightarrow u_4 \rightarrow s_3 \rightarrow u_3 \rightarrow s_g$

定义对特定的 plan π 的代价, 即衡量该 policy 好坏(而不是单独一条轨迹), 为 $G_\pi(x_s)$: cost-to-goal.

1 worst-case analysis for nondeterministic model: $G_\pi(x_s) = \max_{(\tilde{x}, \tilde{u}, \tilde{\theta}) \in \mathcal{H}(\pi, x_s)} \{L(\tilde{x}, \tilde{u}, \tilde{\theta})\}.$

2 expected-case analysis for probabilistic model: $G_\pi(x_s) = E_{\mathcal{H}(\pi, x_s)}[L(\tilde{x}, \tilde{u}, \tilde{\theta})].$

对不确定性的规划进行求解, 主要有 2 种方式: Minimax Cost Planning 和 Expected Cost Planning.

7.3 Minimax Cost Planning

已知一个 MDP 的 nondeterministic model. 一般用 worst-case analysis.

最优的 plan π^* 满足: $G_{\pi^*}(x_s) = \min_{\pi} \{G_{\pi}(x_s)\} = \min_{\pi} \left\{ \max_{(\tilde{x}, \tilde{u}, \tilde{\theta}) \in \mathcal{H}(\pi, x_s)} \{L(\tilde{x}, \tilde{u}, \tilde{\theta})\} \right\}$.

直接求解是很困难的, 可用动态规划解决, 即去寻找最优 plan 中的 stage $k+1$ 到 k 的递推关系.

Solution:

1 终末状态 F 的最优代价 cost-to-goal 可直接得到: $G_F^* = l_F(x_F)$.

2 从第 K 阶段到第 $F = K+1$ 阶段的所有最优 one-step plans 的 costs 为

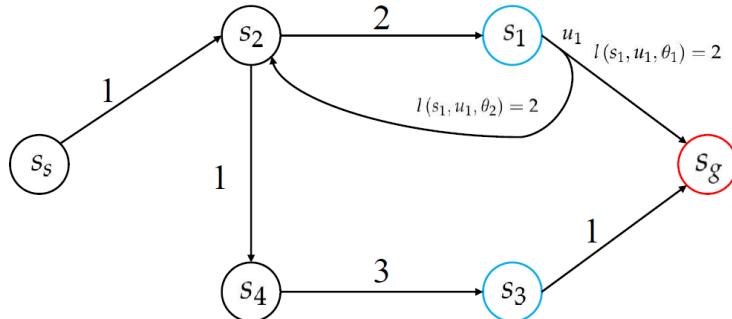
$$G_K^*(x_K) = \min_{u_K} \max_{\theta_K} \{l(x_K, u_K, \theta_K) + G_F^*(f(x_K, u_K, \theta_K))\}.$$

3 更一般的情况, 若 G_{k+1}^* 给定, 可计算出 G_k^* :

$$G_k^*(x_k) = \min_{u_k} \max_{\theta_k} \left\{ \min_{u_{k+1}} \max_{\theta_{k+1}} \left\{ \dots \min_{u_K} \max_{\theta_K} (l(x_K, u_K, \theta_K) + \sum_{i=k+1}^K l(x_i, u_i, \theta_i) + l_F(x_F)) \right\} \right\}.$$

$$G_k^*(x_k) = \min_{u_k} \max_{\theta_k} [l(x_k, u_k, \theta_k) + \underbrace{\min_{u_{k+1}} \dots \min_{u_K} \max_{\theta_K} (\sum_{i=k+1}^K l(x_i, u_i, \theta_i) + l_F(x_F))}_{G_{k+1}^*(x_{k+1})}]$$

因此, 求解 minimax cost plan 的递推为: $G_k^*(x_k) = \min_{u_k \in U(x_k)} \left\{ \max_{\theta_k \in \Theta(x_k, u_k)} \{l(x_k, u_k, \theta_k) + G_{k+1}^*(x_{k+1})\} \right\}$.



- ① suppose $G_{k+1}^*(x_{k+1} = s_g) = 0$
- ② $G_k^*(x_k = s_3) = \min\{1 + 0\}$
- ③ $G_k^*(x_k = s_1) = \min\{\max\{2 + 0, 2 + \infty\}\}$

算法 1: Nondeterministic Dijkstra

$G(x_F) \leftarrow 0$; all other G values are infinite; $\text{OPEN} = \{x_F\}$; $\text{CLOSED} = \emptyset$;

while x_s is not expanded **do**

$x_{k+1} \leftarrow$ remove x with the smallest G value from OPEN;

 insert x_{k+1} to CLOSED;

for every $x_k \notin \text{CLOSED}$ s.t. $x_{k+1} \in X_{k+1}(x_k, u_k)$ **do**

if $G(x_k) > \max_{\theta_k \in \Theta(x_k, u_k)} \{l(x_k, u_k, \theta_k) + G(x_{k+1})\}$ **then**

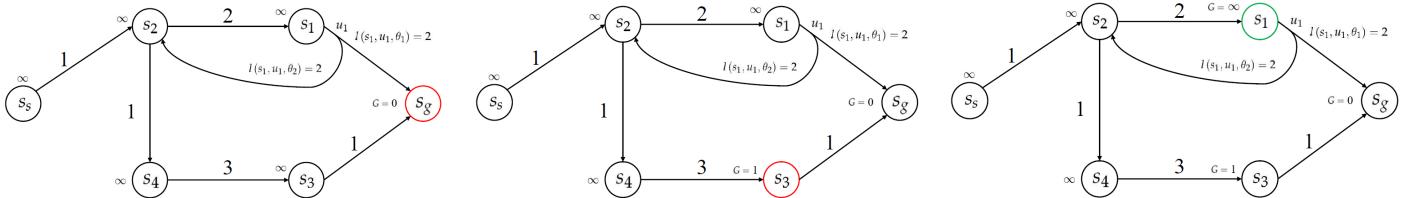
$G(x_k) = \max_{\theta_k \in \Theta(x_k, u_k)} \{l(x_k, u_k, \theta_k) + G(x_{k+1})\}$;

 insert x_k into OPEN;

end

end

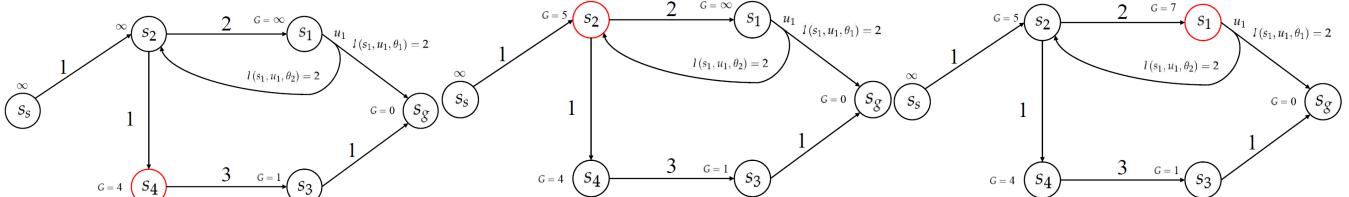
end



- ① CLOSED = {}
- ② OPEN = {s_g}
- ③ next state to expand: s_g
- ④ G(x_{k+1} = s_g) = 0

- ① CLOSED = {s_g}
- ② G(x_k = s₃) = 1 + G(x_{k+1} = s_g)
- ③ OPEN = {s₃}
- ④ next state to expand: -

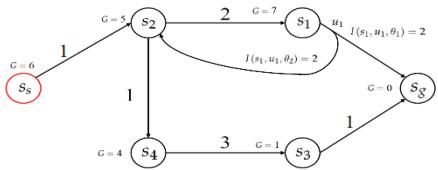
- ① CLOSED = {s_g}
- ② G(x_k = s₁) cannot be reduced
- ③ OPEN = {s₃}
- ④ next state to expand: s₃



- ① CLOSED = {s_g, s₃}
- ② G(x_k = s₄) = 3 + G(x_{k+1} = s₃)
- ③ OPEN = {s₄}
- ④ next state to expand: s₄

- ① CLOSED = {s_g, s₃, s₄}
- ② G(x_k = s₂) = 1 + G(x_{k+1} = s₄)
- ③ OPEN = {s₂}
- ④ next state to expand: s₂

- ① CLOSED = {s_g, s₃, s₄, s₂}
- ② G(x_k = s₁) = 2 + G(x_{k+1} = s₂)
- ③ OPEN = {s₁}
- ④ next state to expand: -



- ① CLOSED = {s_g, s₃, s₄, s₂}
- ② G(x_k = s₅) = 1 + G(x_{k+1} = s₂)
- ③ OPEN = {s₁, s₅}
- ④ next state to expand: s₅

Minimax cost planning 的优缺点:

对不确定性的鲁棒性.

过于悲观.

比正常路径更难计算. 尤其是 nondeterministic Dijkstra 或 A*不适用时. 即使 nondeterministic A*适用, 仍然比用 A*计算单个路径更昂贵.

7.4 Expected Cost Planning

已知一个 MDP 的 probabilistic model. 一般用 expected-case analysis.

最优的 plan π^* 满足: $G_{\pi^*}(x_s) = \min_{\pi} \{G_{\pi}(x_s)\} = \min_{\pi} \{E_{\mathcal{H}(\pi, x_s)}[L(\tilde{x}, \tilde{u}, \tilde{\theta})]\}$.

直接求解是很困难的, 可用动态规划解决, 即去寻找最优 plan 中的 stage $k+1$ 到 k 的递推关系.

Solution:

1 终末状态 F 的最优代价 cost-to-goal 可直接得到: $G_F^* = l_F(x_F)$.

2 从第 K 阶段到第 $F=K+1$ 阶段的所有最优 one-step plans 的 costs 为

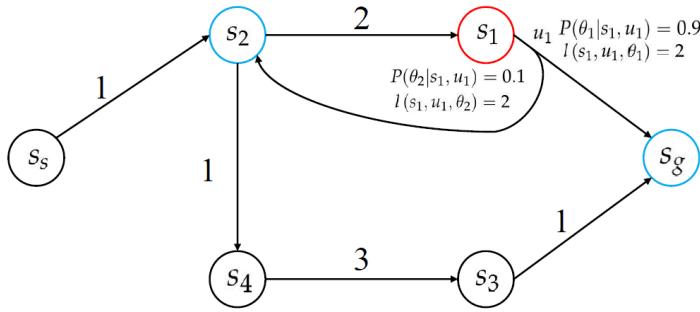
$$G_K^*(x_K) = \min_{u_K} \{E_{\theta_K}[l(x_K, u_K, \theta_K) + G_F^*(f(x_K, u_K, \theta_K))]\}.$$

3 更一般的情况, 若 G_{k+1}^* 给定, 可计算出 G_k^* :

$$G_k^*(x_k) = \min_{u_k} \left\{ E_{\theta_k} \left[\min_{u_{k+1}, \dots, u_K} \{E_{\theta_{k+1}, \dots, \theta_K}[l(x_k, u_k, \theta_k) + \sum_{i=k+1}^K l(x_i, u_i, \theta_i) + l_F(x_F)]\} \right] \right\}.$$

$$G_k^*(x_k) = \min_{u_k} \{E_{\theta_k}[l(x_k, u_k, \theta_k) + \underbrace{\min_{u_{k+1}, \dots, u_K} \{E_{\theta_{k+1}, \dots, \theta_K} [\sum_{i=k+1}^K l(x_i, u_i, \theta_i) + l_F(x_F)]\}}_{G_{k+1}^*(x_{k+1})}\}]$$

因此, 求解 expected cost plan 的递推为: $G_k^*(x_k) = \min_{u_k \in U(x_k)} \{E_{\theta_k} \{l(x_k, u_k, \theta_k) + G_{k+1}^*(x_{k+1})\}\}$, 也称为 Bellman Optimality Equation.



- ① suppose $G_{k+1}^*(x_{k+1} = s_g) = 0$
- ② suppose $G_{k+1}^*(x_{k+1} = s_2) = 2$
- ③ $G_k^*(x_k = s_1) = \min\{(2 + 0) * 0.9 + (2 + 2) * 0.1\}$

算法 2: Value Iteration (VI)

Initialize G values of all states to finite values;

while not converge **do**

for all the states x **do**

$$G(x_F) = 0$$

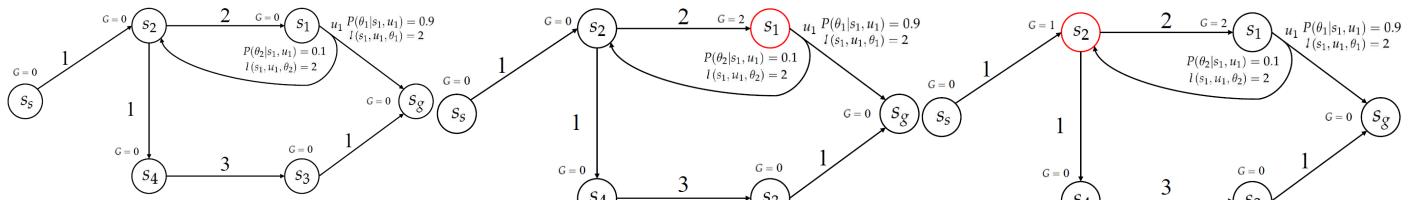
$$G_k(x_k) = \min_{u_k \in U(x_k)} \{E_{\theta_k} [l(x_k, u_k, \theta_k) + G_{k+1}(x_{k+1})]\}, x_k \neq x_F$$

Bellman Update Equation

end
 end

通过值迭代得到最优值. 最优性与迭代顺序无关. 收敛速度取决于迭代顺序.

Bellman 更新方程是实现 Bellman 最优方程的一种方法.



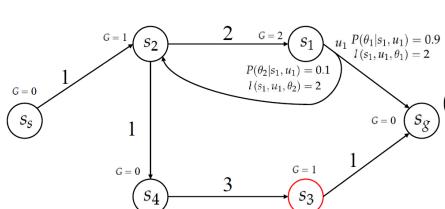
- ① initialize all G value with zero.

- ② iteration order:

$$s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow s_s$$

$$\textcircled{1} \quad G_k^*(x_k = s_1) = \min\{(2 + 0) * 0.9 + (2 + 2) * 0.1\}$$

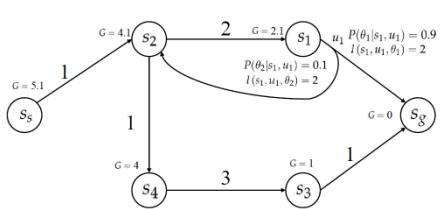
$$\textcircled{1} \quad G_k^*(x_k = s_2) = \min\{(1 + 0) * 1.0, (2 + 2) * 1.0\}$$



$$\textcircled{1} \quad G_k^*(x_k = s_3) = \min\{(1 + 0) * 1.0\}$$

$$\textcircled{1} \quad G_k^*(x_k = s_4) = \min\{(3 + 1) * 1.0\}$$

$$\textcircled{1} \quad G_k^*(x_k = s_s) = \min\{(1 + 1) * 1.0\}$$



① after second iteration

Expected cost planning 的优缺点:

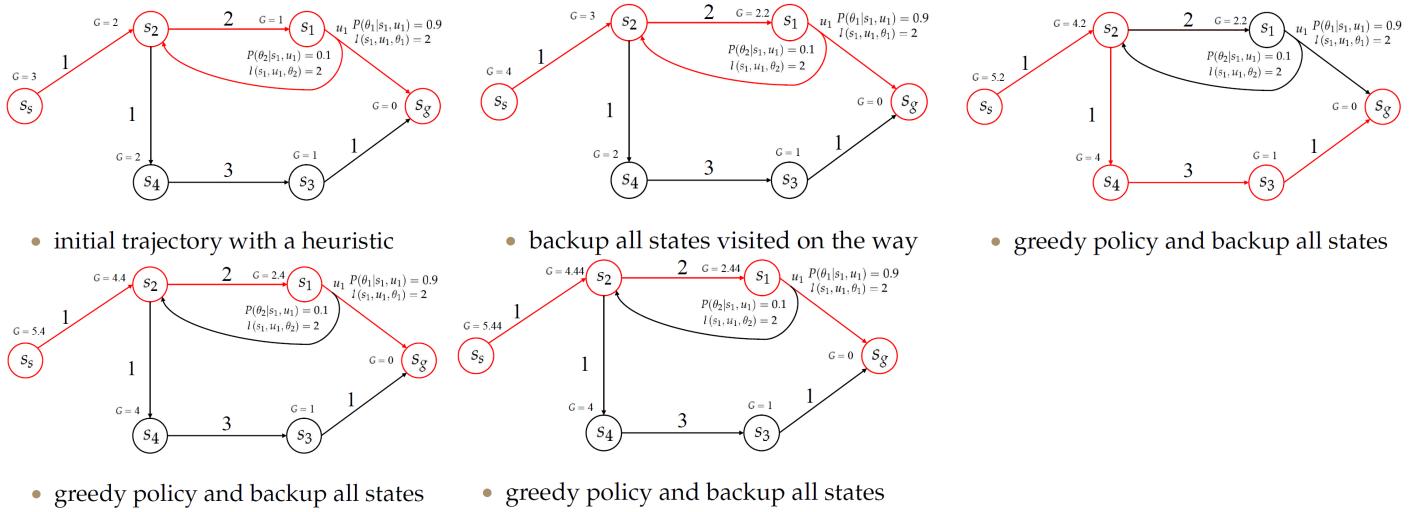
- 概率最优: 反映了平均性能. 一次特定的执行可能不是最优的.
- 需要不确定性的分布.
- 比正常路径更难计算: 在整个地图(状态)上迭代. 受初始化和迭代顺序的影响.

7.5 实时动态规划(RTDP)

算法 3: RTDP

- 1 将所有状态的 G 初始化为 admissible values.
- 2 遵循贪婪策略随机挑选结果, 直到达到目标.
- 3 备份途中访问的所有状态.
- 4 重置为 x_s 并重复步骤 2~4, 直到当前贪婪策略上的所有状态的 Bellman 误差 $< \Delta$, 其中 $\Delta(x_k) = \|G(x_k) - G(x_{k+1})\|$.

RTDP 的优点在于: 是一个非常有效的 value 迭代的替代方案. 不计算所有状态的值. 计算都集中在相关的关系上.



8 模型预测控制(Model Predictive Control)在运动规划中的应用

线性 MPC 解决凸优化问题, 大多数工程问题, 都需要用到解决非凸优化的非线性 MPC.

模型: system model / problem model

预测: 状态空间 / 输入空间 / 参数空间

控制: 选择最好策略的过程. 即在参数空间内选择一组最好的参数.

例子:

$$\left\{ \begin{array}{l} \min_u C_F(x(t_f)) + \int_{t=t_0}^{t_f} C_R(x, u) dt \\ \dot{x} = f(x, u) \\ g(x, u) < 0 \\ h(x, u) = 0 \\ x \notin \text{Obstacle} \end{array} \right.$$

式 1: 问题模型(也是优化目标): final cost $C_F(x(t_f))$ 和 running cost

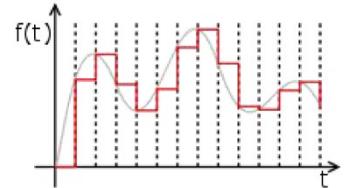
$\int_{t=t_0}^{t_f} C_R(x, u) dt$. 约束条件为式 2~5. 式 2: dynamic constraint, 描述了动力学模型. 式 3: 最大限制. 如最大速度限制, 最大加速度限制等. 式 4: 等式限制. 如终点位置等. 式 5: collision avoidance constraint. 可以归类于不等式约束中, 但由于其性质特殊(非凸的). 为提高实时性所以单独考虑.

参数空间: 令无限维度的系统输入用有限维度的参数来描述. 方法:

Zero order hold (direct discretization): 如图

多项式 / 样条: $u(t) = at^3 + bt^2 + ct + d$

数值映射: 有限 jerk 的轨迹 / 基于神经网络的方法



优化

搜索: 图搜索 / 基于随机采样的搜索

凸优化: QP (quadratic programming)

非凸优化: sequential QP / particle swarm optimization (可解决非凸, 非线性, 不连续问题)

控制

1. 设置优化问题.

2. 获取当前状态, 确定优化问题.

3. 求解优化问题, 得到 u^* .

4. 由于模型不确定性和外界的干

扰, 只在短时间内使用 u^* (如

0.1s). 再回到步骤 2 重新优化. 这个过程称为 resetting horizontal control (RHC). 是 MPC 的基础.

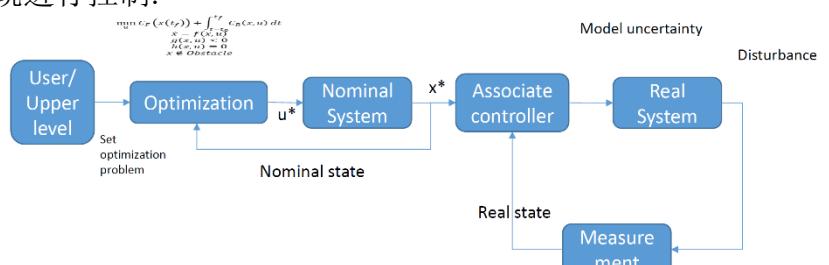
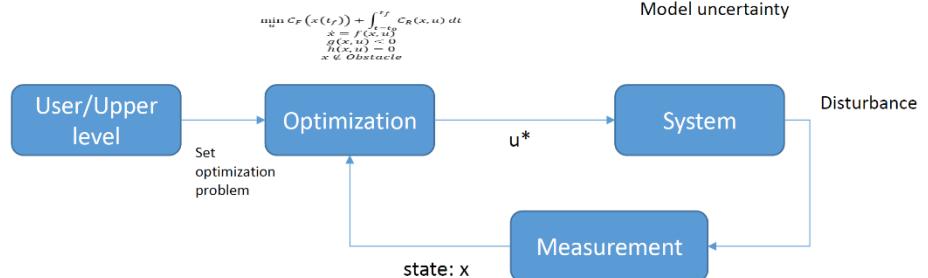
Tube-based MPC: 传统的 MPC 要求系统的控制频率非常高(如 200~1000Hz), 要求求解速度很快. 该方法先求解"Optimization→Nominal system"得到 x^* , 因为只是数学模型, 不涉及干扰, 这样就可以以非常低的频率对 Nominal system 进行控制. 我们希望 real state 与 x^* 保持一致. 于是引入 Associate controller 跟踪 x^* . 该 controller 可以转化为一个线性/简单的非线性/MPC 控制器. Real system 处理模型的不确定性和外界的干扰, 使系统可以跟踪 x^* . 整个系统可以以极低的频率解决优化问题, 从而节省计算量, 但又可以较高的频率对真实系统进行控制.

1. x^* 称"参考轨迹", 也称 nominal state.

2. Nominal system 只是一个数学模型
 $\dot{x} = f(x, u)$.

3. 最小化优化问题的求解速度.

4. 将鲁棒跟踪问题交给相关控制器处理.



相关资源

Matlab MPC toolbox: <https://www.mathworks.com/products/mpc.html>

μ AO-MPC: <http://ifatwww.et.uni-magdeburg.de/syst/muAO-MPC/>

Acado toolkit: <https://acado.github.io/>

YANE: <http://www.nonlinearmpc.com/>

Multi-Parametric Toolbox 3: <https://www.mpt3.org/>

8.1 线性 MPC

1 连续模型: $\dot{p} = v, \dot{v} = a, \dot{a} = j$, 考虑优化 4s 内(Prediction horizon)的系统状态. 先把时间离散化:
 $i = [0, 1, \dots, 19]$, $dt = 0.2$. 有 $p_i = p(i \cdot dt)$, $v_i = v(i \cdot dt)$, $a_i = a(i \cdot dt)$. 故模型被离散为:

$$\begin{cases} p_{i+1} = p_i + v_i dt + \frac{1}{2} a_i dt^2 + \frac{1}{6} j_i dt^3 \\ v_{i+1} = v_i + a_i dt + \frac{1}{2} j_i dt^2 \\ a_{i+1} = a_i + j_i dt \end{cases} \rightarrow \text{矩阵化} \rightarrow \begin{cases} \mathbf{P} = \mathbf{T}_p \mathbf{J} + \mathbf{B}_p \\ \mathbf{V} = \mathbf{T}_v \mathbf{J} + \mathbf{B}_v, \text{ 称预测模型.} \\ \mathbf{A} = \mathbf{T}_a \mathbf{J} + \mathbf{B}_a \end{cases}$$

$P = [p_1, p_2, p_3, \dots, p_{20}]^T$
 $V = [v_1, v_2, v_3, \dots, v_{20}]^T$
 $A = [a_1, a_2, a_3, \dots, a_{20}]^T$
 $J = [j_0, j_1, j_2, \dots, j_{19}]^T$

其中 $\mathbf{B}_p, \mathbf{B}_v, \mathbf{B}_a$ 是关于 p_0, v_0, a_0 的函数, \mathbf{J} 是参数空间.

2 问题模型.

目标 1: 0 位置, 0 速度, 0 加速度. 优化目标 1: $\min_J w_1 \mathbf{P}^T \mathbf{P} + w_2 \mathbf{V}^T \mathbf{V} + w_3 \mathbf{A}^T \mathbf{A}$

目标 2: 光滑轨迹. 优化目标 2: $\min_J w_4 \mathbf{J}^T \mathbf{J}$

3 优化. $\min_J (w_1 \mathbf{T}_p^T \mathbf{T}_p + w_2 \mathbf{T}_v^T \mathbf{T}_v + w_3 \mathbf{T}_a^T \mathbf{T}_a + w_4 \mathbf{I}) \mathbf{J} + 2(w_1 \mathbf{B}_p^T \mathbf{T}_p + w_2 \mathbf{B}_v^T \mathbf{T}_v + w_3 \mathbf{B}_a^T \mathbf{T}_a) \mathbf{J} + c$,

其中 c 为常数. 这是一个 QP 问题. 其中 \mathbf{I} 是正定的, $\mathbf{T}^T \mathbf{T}$ 也至少是半正定的, 故该 Hessian 矩阵是正定的, 为凸优化问题. 可以用优化工具库 quadprog 解决.

4 控制. 每隔 0.2s 用求解出的最优控制信号来驱动实际系统.

```

1 p_0 = 10;
2 v_0 = 0;
3 a_0 = 0;
4 K=20;
5 dt=0.2;
6 log=[0 p_0 v_0 a_0];
7 w1 = 1;
8 w2 = 1;
9 w3 = 1;
10 w4 = 1;
11 for t=0.2:0.2:10
12 % Construct the prediction matrix
13 [Tp, Tv, Ta, Bp, Bv, Ba] = getPredictionMatrix(K,dt,p_0,v_0,a_0);
14
15 %% Construct the optimization problem
16 H = w4*eye(K)+w1*(Tp'*Tp)+w2*(Tv'*Tv)+w3*(Ta'*Ta);
17 F = w1*Bp'*Tp+w2*Bv'*Tv+w3*Ba'*Ta;
18
19 %% Solve the optimization problem
20 J = quadprog(H,F,[],[]);
21
22 %% Apply the control
23 j = J(1);
24 p_0 = p_0 + v_0*dt + 0.5*a_0*dt^2 + 1/6*j*dt^3;
25 v_0 = v_0 + a_0*dt + 0.5*j*dt^2;
26 a_0 = a_0 + j*dt;
27
28 %% Log the states
29 log = [log; t p_0 v_0 a_0];
30 end

```

5 硬约束. 限制为 $-1 \leq v_i \leq 1, -1 \leq a_i \leq 1, \forall i \in \{1, 2, \dots, 20\}$. 矩阵形式为

$$\begin{cases} -1_{20 \times 1} \leq \mathbf{V} = \mathbf{T}_v \mathbf{J} + \mathbf{B}_v \leq 1_{20 \times 1} \\ -1_{20 \times 1} \leq \mathbf{A} = \mathbf{T}_a \mathbf{J} + \mathbf{B}_a \leq 1_{20 \times 1} \end{cases}, \text{ 化简为 } \begin{cases} \mathbf{T}_v \mathbf{J} \leq 1_{20 \times 1} - \mathbf{B}_v \\ -\mathbf{T}_v \mathbf{J} \leq 1_{20 \times 1} + \mathbf{B}_v \\ \mathbf{T}_a \mathbf{J} \leq 1_{20 \times 1} - \mathbf{B}_a \\ -\mathbf{T}_a \mathbf{J} \leq 1_{20 \times 1} + \mathbf{B}_a \end{cases}.$$

```

A = [Tv;-Tv;Ta;-Ta];
b = [ones(20,1)-Bv;ones(20,1)+Bv;ones(20,1)-Ba;ones(20,1)+Ba];

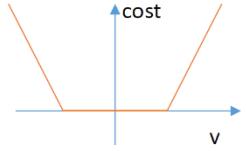
```

在 Matlab 中默认 $\mathbf{AJ} < \mathbf{b}$.

MPC 和运动规划问题的不同点: MPC 受限于 prediction horizon, 运动时间很短, 因此终末状态设置不合理的话很有可能使得问题无解. 因此大多数 MPC 把终末状态放入目标函数内而不是作为一个约束条件. 这样至少能得到一个解.

较长的 prediction horizon 需要更强的计算能力. 太短的 prediction horizon 不能对系统起到正定的作用, 结果会发散.

6 软约束. 向优化目标中加入一个惩罚项. $\min_{\mathbf{J}} w_1 \mathbf{P}^T \mathbf{P} + w_2 \mathbf{V}^T \mathbf{V} + w_3 \mathbf{A}^T \mathbf{A} + w_4 \mathbf{J}^T \mathbf{J} + S(\mathbf{V})$. 其中惩罚项 $S(\mathbf{V}) = \sum_{i=1}^{20} s(v_i)$, $s(v_i) = \begin{cases} 0 & \text{if } \|v_i\| \leq 1 \\ M \cdot (\|v_i\| - 1) & \text{else}, \end{cases}$, M 是一个较大的正数.



以将 $-\mathbf{T}_v \mathbf{J} \leq \mathbf{1}_{20 \times 1} + \mathbf{B}_v$ 更改为软约束为例. 添加一个松弛变量 slack variable \mathbf{L} : $-\mathbf{T}_v \mathbf{J} \leq \mathbf{1}_{20 \times 1} + \mathbf{B}_v + \mathbf{L}$, 其中 $-\mathbf{L} \leq 0$, $\mathbf{L} = [l_1, l_2, \dots, l_{20}]^T$.

新的优化目标为: $\min_{\mathbf{J}, \mathbf{L}} w_1 \mathbf{P}^T \mathbf{P} + w_2 \mathbf{V}^T \mathbf{V} + w_3 \mathbf{A}^T \mathbf{A} + w_4 \mathbf{J}^T \mathbf{J} + w_5 \mathbf{L}^T \mathbf{L}$, 其中 w_5 一般会设定为比较大的值.

设一个新的规划变量 $\bar{\mathbf{J}} = \begin{bmatrix} \mathbf{J} \\ \mathbf{L} \end{bmatrix}$, 所以所有的 $\mathbf{H}, \mathbf{F}, \mathbf{A}, \mathbf{b}$ 都要做相应调整.

```
%>> Construct the prediction matrix
[Tp, Tv, Ta, Bp, Bv, Ba] = getPredictionMatrix(K, dt, p_0, v_0, a_0);

%>> Construct the optimization problem
H = blkdiag(w4*eye(K)+w1*(Tp'*Tp)+w2*(Tv'*Tv)+w3*(Ta'*Ta), w5*eye(K));
F = [w1*Bp'*Tp+w2*Bv'*Tv+w3*Ba'*Ta zeros(1,K)];
A = [Tv zeros(K); -Tv -eye(K); Ta zeros(K); -Ta zeros(K); zeros(size(Ta)) -eye(K)];
b = [ones(20,1)-Bv; ones(20,1)+Bv; ones(20,1)-Ba; ones(20,1)+Ba; zeros(K,1)];
%>> Solve the optimization problem
J = quadprog(H,F,A,b);
```



哪些约束适用于硬约束? 哪些约束适用于软约束?

涉及状态的约束应考虑设为软约束, 因为状态往往收到测量噪声和外界干扰的影响使得其初始值违反了约束条件.

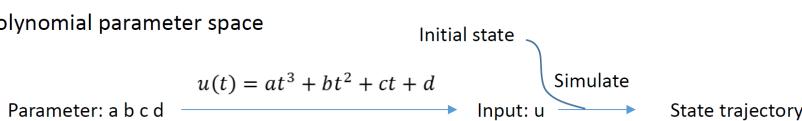
输入变量则应考虑设为硬约束, 因为其不受干扰, 并且超过了约束的话会对系统产生物理性的损伤.

线性 MPC 的缺陷: 通常需要线性模型或可被线性化(adaptative MPC), 而且障碍物约束通常情况下为非凸的. (Tips: 设计到'或'条件时, 该条件往往就是非凸的)

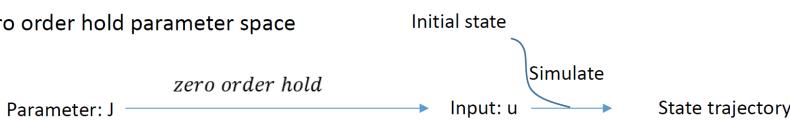
8.2 参数空间: Boundary constrained motion primitives (BSCP)

与传统的多项式参数空间不同的是, 在从参数到轨迹的转化过程中要求解 BVP. 例如, 多项式参数空间有 4 个参数, 用多项式即可表示成输入 u , 搭配初始状态即可仿真出状态轨迹. 而 BSCP 的参数是初始状态和期望的终末状态. 把这 2 个参数放进两点边界问题的算子内, 可得到一条系统轨迹, 使系统在满足约束条件的同时从初始状态和期望的终末状态.

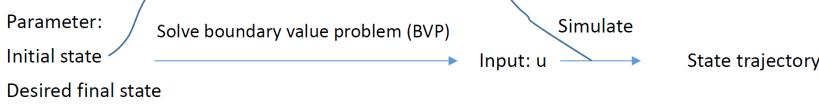
- Polynomial parameter space



- Zero order hold parameter space



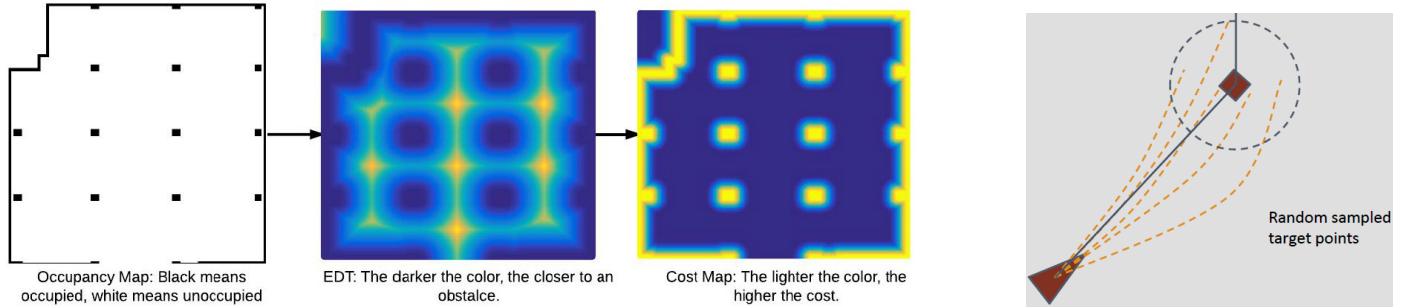
- BSCP



8.3 非线性 MPC

Jerk limited trajectory (JLT)与非线性 MPC 结合以解决本地的(而不是全局的)运动规划问题. 因为 MPC 的本质是把运动规划问题变成优化问题来解决, 而优化问题的轨迹的复杂度是有限的, 而全局规划可能有无限自由度的轨迹. 随着自由度的增加, 解决速度也会变慢.

1 环境观测: 像素化环境 occupancy map, 转化为 cost map (包含进入每个像素点的 cost 和到最近障碍物的距离).



2 Two level guidance. 全局规划(A*/JPS) + 轨迹规划(右上图) + 评估.

3 事件管理器 event manager. MPC 每隔一段时间要重新优化一次. Event manager 可决定什么时候使用新的轨迹, 一般搭配 Tube-based MPC. 共有 3 种决定策略: 快撞上障碍物时更新 / 轨迹快走完时更新 / 用户更改终末状态. 可节约计算量, 也可使轨迹更光滑.

$$\left\{ \begin{array}{l} \min_u C_F(x(t_f)) + \int_{t=t_0}^{t_f} C_R(x, u) dt \\ \dot{x} = f(x, u) \\ g(x, u) < 0 \\ h(x, u) = 0 \\ x \notin \text{Obstacle} \end{array} \right. . \text{ 其中 } \left\{ \begin{array}{l} C_R(x, u) = (x - x_d)^T Q (x - x_d) + u^T R u + s(x) \\ C_F = (x - x_d)^T W (x - x_d) + s(x) \end{array} \right. . \text{ JLT} -$$

定满足式 2~4, 故硬约束为 $x \notin \text{Obstacle}$. 软约束为 $s(x) = \begin{cases} 0, & \text{if } x \notin \text{Obstacle} \\ M, & \text{otherwise} \end{cases}$.

用粒子群优化方法 PSO.

```

Input:  $x_{\text{ini}}$ ,  $\mathcal{M}(\text{map})$ 
Output:  $\theta^*$  (best end state constraint)
1:  $\Theta \leftarrow \text{Particle\_Initialization}();$ 
2:  $c_i^* \leftarrow \infty, \theta_i^* \leftarrow \theta_i, \delta_i \leftarrow \text{rand}, \forall i \in [1, \text{size}(\Theta)]$ 
3: for  $m = 1$  to MAX_ITERS do
4:   for each  $\theta_i \in \Theta$  do
5:      $[x(t), u(t)] = \mathcal{S}_{\text{NN}}(x_{\text{ini}}, \theta_i)$ 
6:      $c_i = J(x(t), u(t), \mathcal{M})$ 
7:     if  $c_i < c_i^*$  then
8:        $c_i^* = c_i$ 
9:        $\theta_i^* = \theta_i$ 
10:       $i^* = \underset{i}{\operatorname{argmin}}(c_i^*)$ 
11:       $\theta^* = \theta_{i^*}$ 
12:      for each  $\theta_i \in \Theta$  do
13:         $\delta_i = \delta_i + k_1 \cdot \text{rand} \cdot (\theta_i^* - \theta_i) + k_2 \cdot \text{rand} \cdot$ 
           $(\theta^* - \theta_i)$ 
14:         $\theta_i = \theta_i + \delta_i$ 

```

8.4 General BSCP

一般的 BVP 通常需要数值优化, 解析解存在但很少. 有时也可以通过动态规划求解一个控制器. 在 BSCP 中, 选定了初始状态 x_0 和终止状态参数 θ 后就能得到一组轨迹 $\mathcal{S}: \langle x_0, \theta \rangle \rightarrow \langle \hat{u}(t), \hat{x}(t), \hat{t}_f \rangle$. 选择合理的 θ 使得 $\min_{\theta} L_F(\theta) + \int_{t=t_0}^{t_f} L_R(\theta) dt$. 可用 PSO 算法找出 θ , 但速度很慢.

可用神经网络来解决.

更新 x_0 .

优化 $\min_{\theta} L_F(\theta) + \int_{t=t_0}^{t_f} L_R(\theta) dt$, 得到 θ^* .

1) 用神经网络近似输入项 u .

2) 求解原始的 BVP.

$$\min_{u(t), x(t), t_f} G(x(t), u(t), t_f)$$

$$g(x(t_f), \theta) = 0 \quad \dot{x} = f(x, u)$$

$$h(x, u) = 0, \tilde{h}(x, u) \leq 0$$

