

## Lecture 4





# Outline



1. Introduction



2. State Lattice Planning



3. Kinodynamic RRT\*



4. Hybrid A\*



5. Homework

# Introduction



# What is kinodynamic

## Kinodynamic : Kinematic + Dynamic

The *kinodynamic planning* problem is to synthesize a robot motion subject to simultaneous *kinematic* constraints, such as *avoiding obstacles*, and *dynamics* constraints, such as modulus *bounds on velocity, acceleration, and force*. A kinodynamic solution is a mapping from time to generalized forces or accelerations.

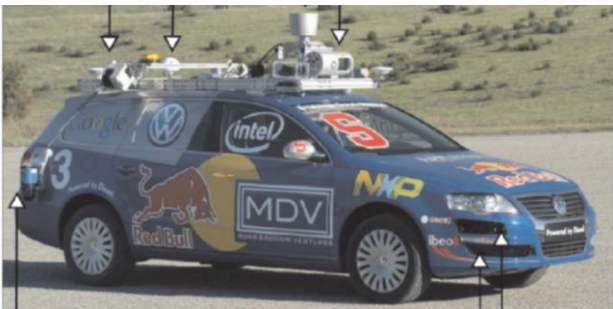
—— *Kinodynamic Motion Planning*, Bruce Donald, Patrick Xavier, John Canny, John Reif

- Differentially constrained
- Up to force (acceleration)

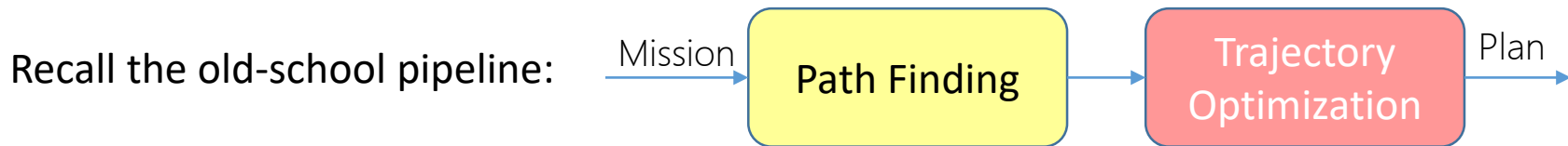


# Why kinodynamic planning ?

Straight-line connections between pairs of states are typically not valid trajectories due to the system's **differential constraints**.



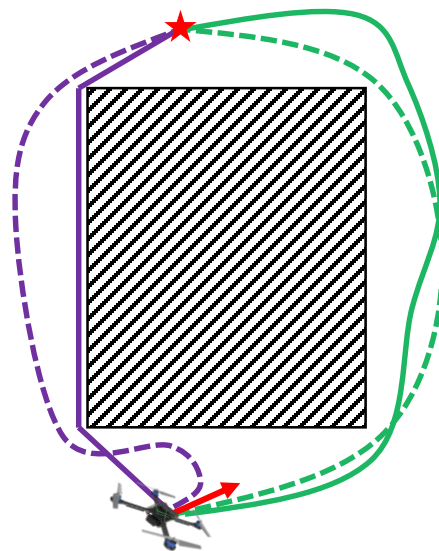
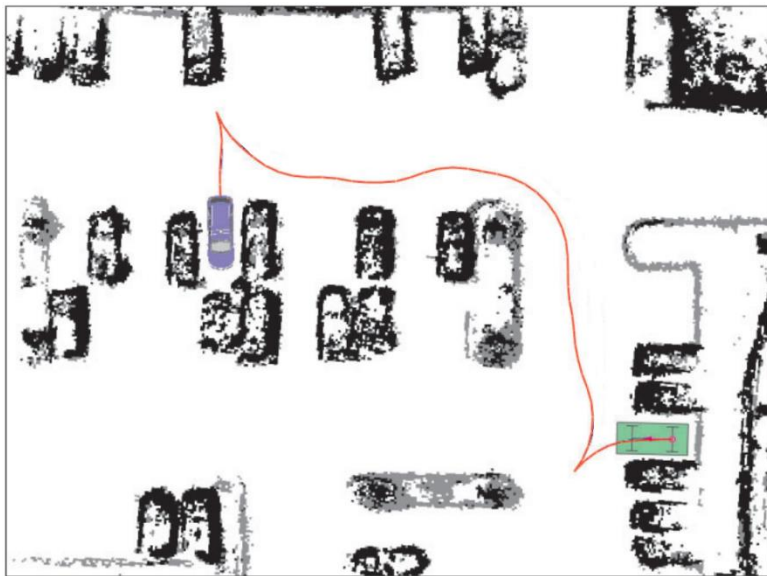
Ask: We have the back-end optimization, why kinodynamic planning?





# Why kinodynamic planning ?

- Coarse-to-fine process
- Trajectory only optimizes locally
- Infeasible path means nothing to nonholonomic system

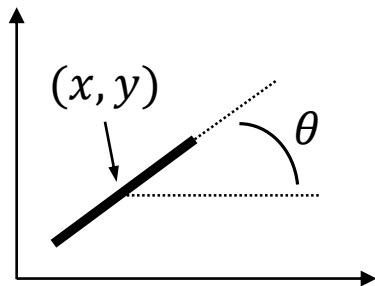




# Examples

Unicycle and differential drive models:

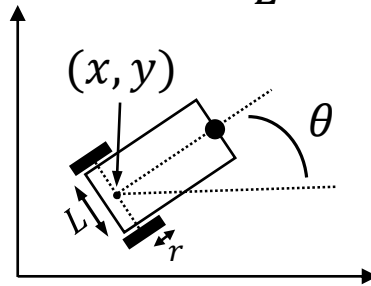
$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} \cos\theta \\ \sin\theta \\ 0 \end{pmatrix} \cdot v + \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \cdot \omega$$



Unicycle

$$\begin{aligned} |v| &\leq v_{max} \\ |\omega| &\leq \omega_{max} \end{aligned}$$

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} \frac{r}{2} (u_l + u_r) \cos\theta \\ \frac{r}{2} (u_l + u_r) \sin\theta \\ \frac{r}{L} (u_r - u_l) \end{pmatrix}$$



Differential drive

$$\begin{aligned} |u_l| &\leq u_{l,max} \\ |u_r| &\leq u_{r,max} \end{aligned}$$

$$v = \frac{r}{2} (u_l + u_r)$$

$$\omega = \frac{r}{L} (u_r - u_l)$$



# Examples

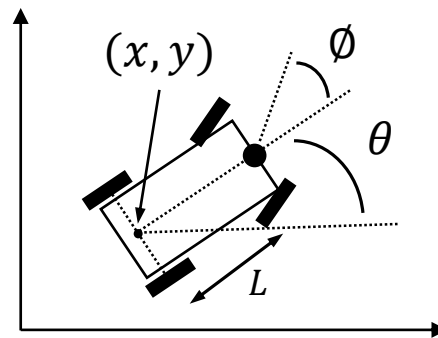
Simplified car model

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} v \cos \theta \\ v \sin \theta \\ \frac{r}{L} \tan \phi \end{pmatrix}$$

$$|v| \leq v_{max}, \quad |\phi| \leq \phi_{max} < \frac{\pi}{2}$$

$$v \in \{-v_{max}, v_{max}\}, \quad |\phi| \leq \phi_{max} < \frac{\pi}{2}$$

$$v = v_{max}, \quad |\phi| \leq \phi_{max} < \frac{\pi}{2}$$



→ Simple car model

→ Reeds & Shepp's car

→ Dubin's car



# State Lattice Planning



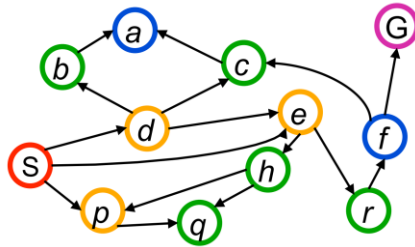
# Workflow



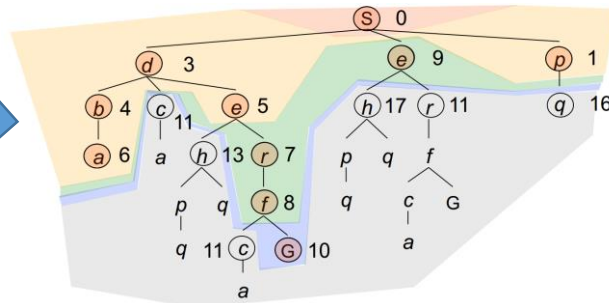
# Basic idea

- Recall the search-based path finding method in L2
- For planning, how to build a graph?
- Is this graph doable for our real robot?

Search graph



Search tree

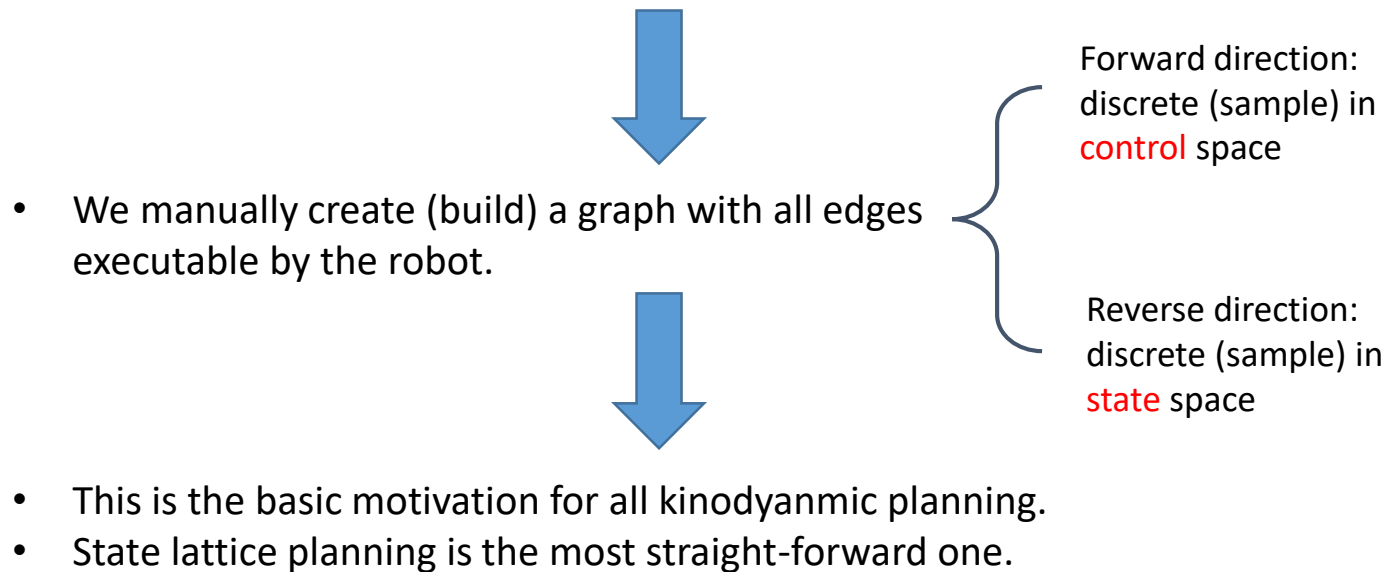


- Maintain a **priority queue** to store all the nodes to be expanded
- The heuristic function  $h(n)$  for all nodes are pre-defined
- The priority queue is initialized with the start state  $X_s$
- Assign  $g(X_s)=0$ , and  $g(n)=\text{infinite}$  for all other nodes in the graph
- Loop
  - If the 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  $g(m) = \text{infinite}$ 
      - $g(m) = g(n) + C_{nm}$
      - Push node "m" into the queue
    - If  $g(m) > g(n) + C_{nm}$ 
      - $g(m) = g(n) + C_{nm}$
  - end
- End Loop



# Basic idea

- We have many weapons to attack graph search.
- Assume the robot a mass point is not satisfactory any more.
- We now require a graph with **feasible motion connections**.





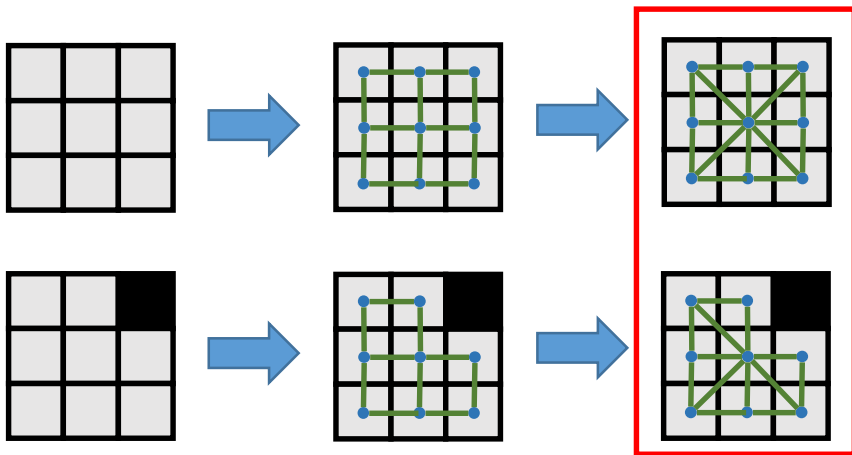
# Connection with previous lectures

- We manually create (build) a graph with all edges executable by the robot.

Forward direction:  
discrete (sample) in  
**control** space

Reverse direction:  
discrete (sample) in  
**state** space

Things in L2



4 connection

8 connection

- Actually, this is a discretization of control space!
- We assume the robot can move in 4/8 directions



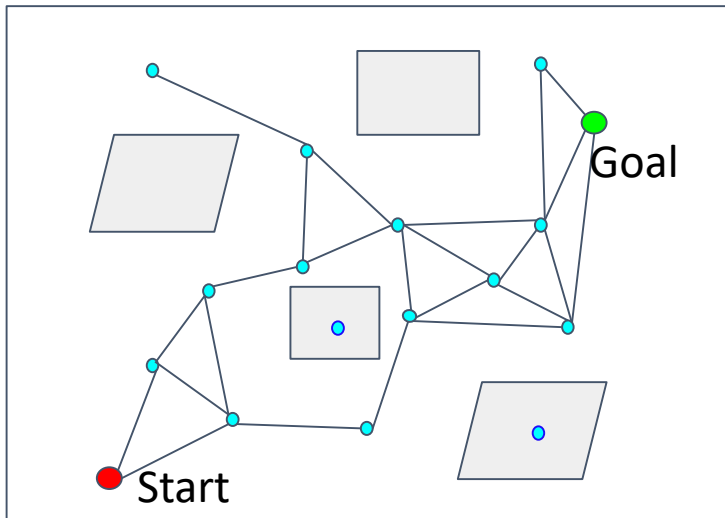
# Connection with previous lectures

- We manually create (build) a graph with all edges executable by the robot.

Forward direction:  
discrete (sample) in  
**control** space

Reverse direction:  
discrete (sample) in  
**state** space

Things in L3



- Actually, this is a discretization of state space!
- Here the state is  $R^2$ , only position (x, y) is considered here.

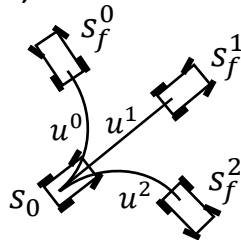


# Build the graph, sample in control vs. state space

For a robot model:  $\dot{s} = f(s, u)$

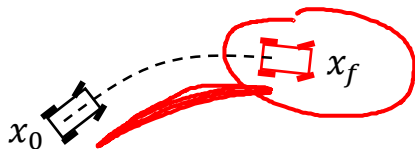
- The robot is differentially driven.
- We have an initial state  $s_0$  of the robot.
- Generate feasible local motions by:

❑ Select a  $u$ , fix a time duration  $T$ , forward simulate the system (numerical integration).



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

❑ Select a  $s_f$ , find the connection (a trajectory) between  $s_0$  and  $s_f$ .



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



# Sample in control space



$$\text{State: } s = \begin{pmatrix} x \\ y \\ z \\ \dot{x} \\ \dot{y} \\ \dot{z} \end{pmatrix} \quad \text{Input: } u = \begin{pmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{pmatrix}$$

$$\text{System equation: } \dot{s} = A \cdot s + B \cdot u$$

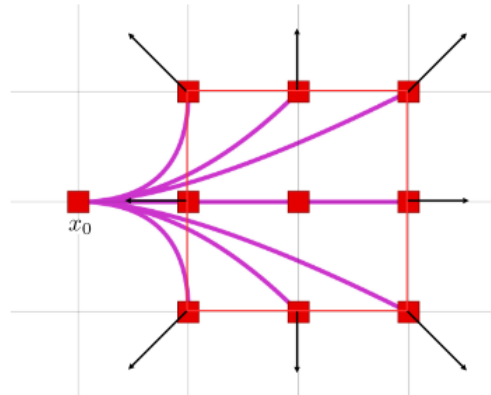
$$A = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

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



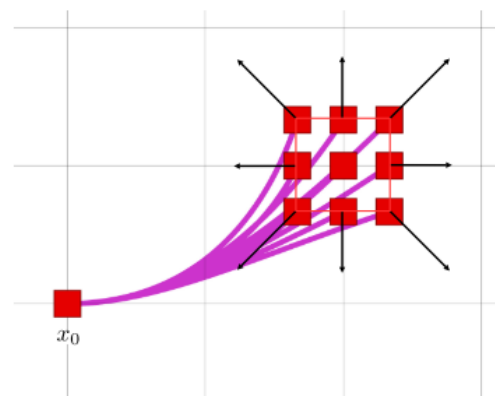
$$\dot{s} = A \cdot s + B \cdot u$$

$$A = \begin{bmatrix} 0 & I_3 & 0 & \dots & 0 \\ 0 & 0 & I_3 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & \dots & 0 & I_3 \\ 0 & \dots & \dots & 0 & 0 \end{bmatrix} \quad B = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ I_3 \end{bmatrix}$$



Discretize acceleration

$$v_0 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$



Discretize jerk

$$v_0 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, a_0 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

- Several-order integrator
- $A$  **nilpotent**
- Very special, remember it





# Sample in control space

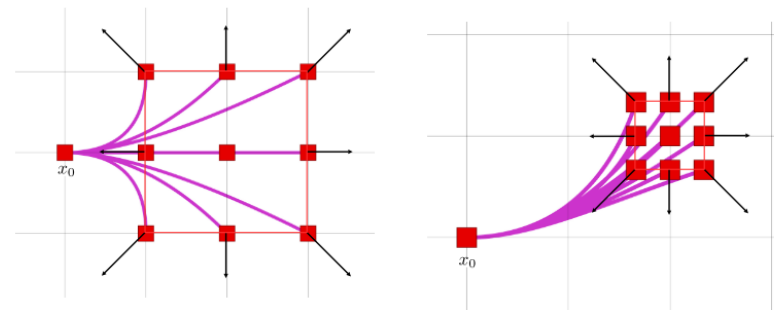


$$\text{State: } s = \begin{pmatrix} x \\ y \\ z \\ \dot{x} \\ \dot{y} \\ \dot{z} \end{pmatrix} \quad \text{Input: } u = \begin{pmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{pmatrix}$$

$$\text{System equation: } \dot{s} = A \cdot s + B \cdot u$$

$$\dot{s} = A \cdot s + B \cdot u$$

$$A = \begin{bmatrix} 0 & I_3 & 0 & \cdots & 0 \\ 0 & 0 & I_3 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & 0 & I_3 \\ 0 & \cdots & \cdots & 0 & 0 \end{bmatrix} \quad B = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ I_3 \end{bmatrix}$$



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

$e^{At}$  : state transition matrix,  
critical to the integration.

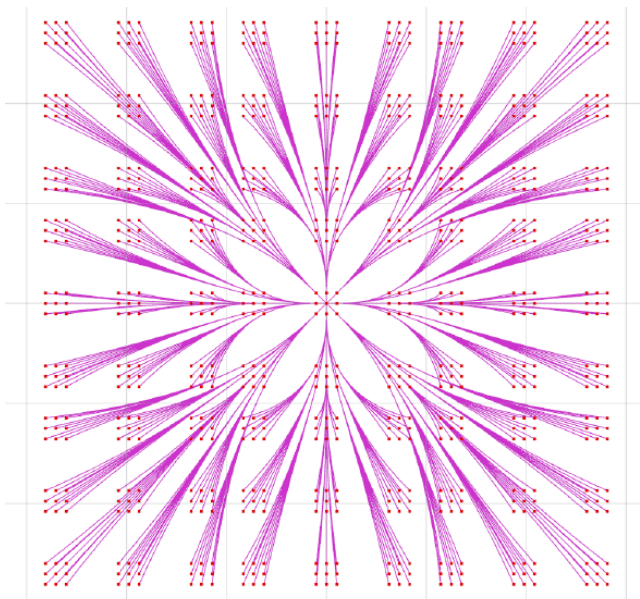
$$e^{At} = I + \frac{At}{1!} + \frac{(At)^2}{2!} + \frac{(At)^3}{3!} + \cdots + \frac{(At)^k}{k!} + \cdots$$

If matrix  $A \in \mathbb{R}^{n \times n}$  is **nilpotent**, i.e.  $A^n = 0$ ,  $e^{At}$  has a closed-form expression in the form of an  $(n - 1)$  degree matrix polynomial in  $t$ .

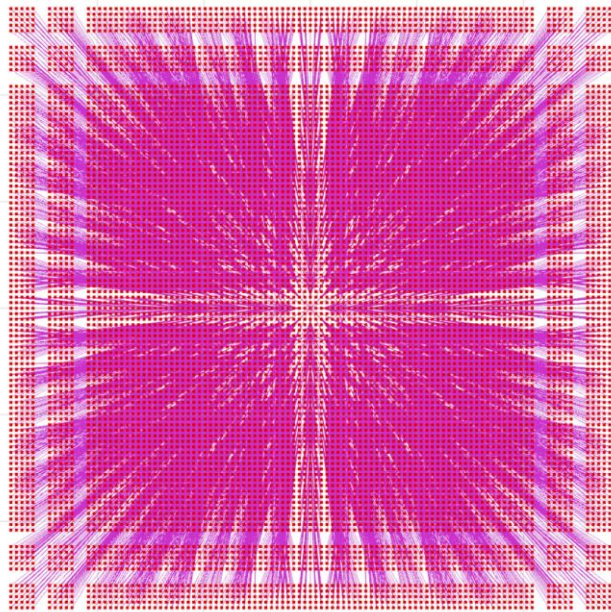


# Sample in control space

The lattice graph obtained by searching



9 discretization



25 discretization

## 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.



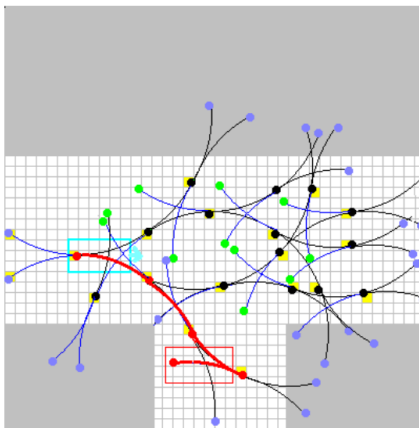
# Sample in control space



**State:**  $s = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix}$       **Input:**  $u = \begin{pmatrix} v \\ \phi \end{pmatrix}$

**System equation:**  $\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} v \cdot \cos\theta \\ v \cdot \sin\theta \\ \frac{r}{L} \cdot \tan\phi \end{pmatrix}$

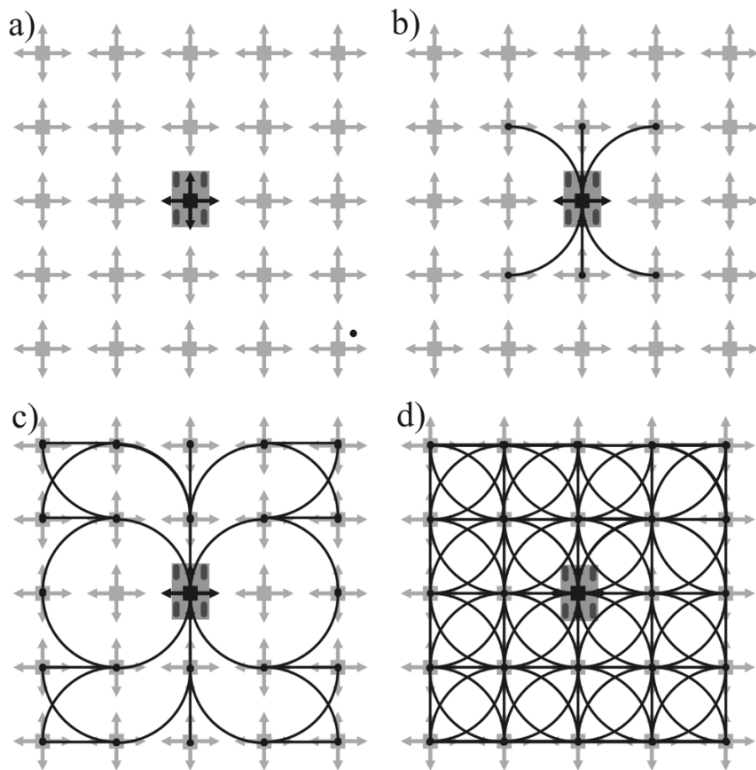
- For every  $s \in T$  from the search tree
- Pick a control vector  $u$
- Integrate the equation over short duration
- Add collision-free motions to the search tree



- 1) Select a  $s \in T$
- 2) Pick  $v, \phi$  and  $\tau$
- 3) Integrate motion from  $s$
- 4) Add result if collision-free



# Sample in state space



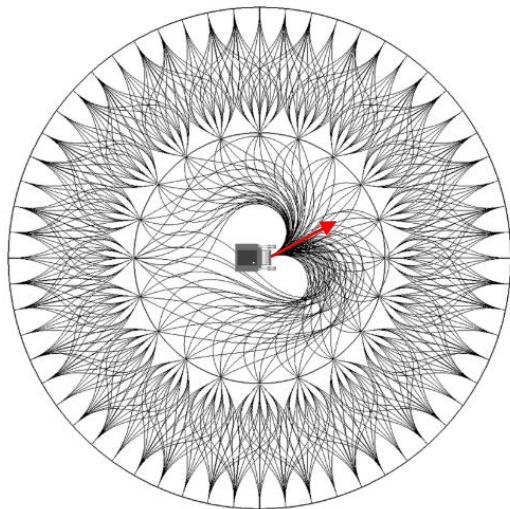
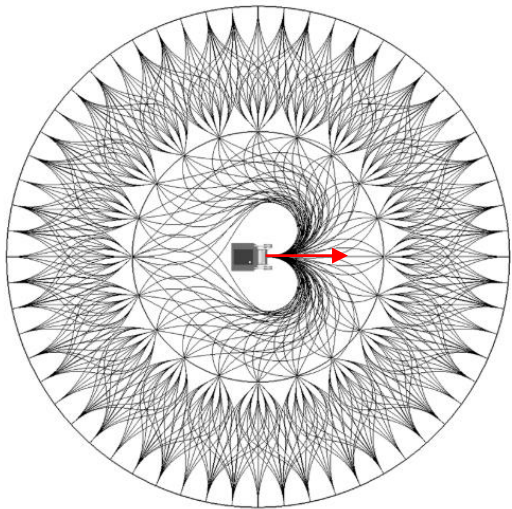
Build a lattice graph:

- 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



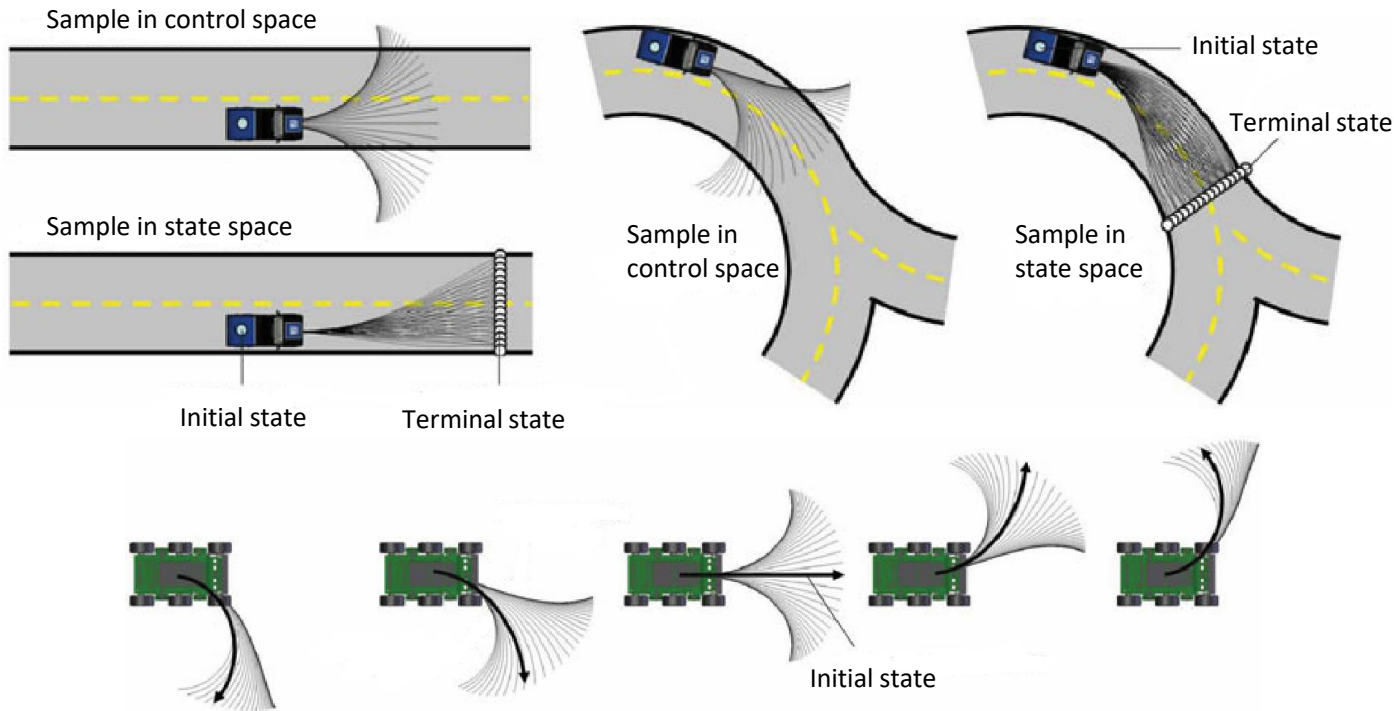
# Sample in state space



- Two layer lattice graph
- Only first layer is different
- Different initial states



# Comparison



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



# Boundary Value Problem





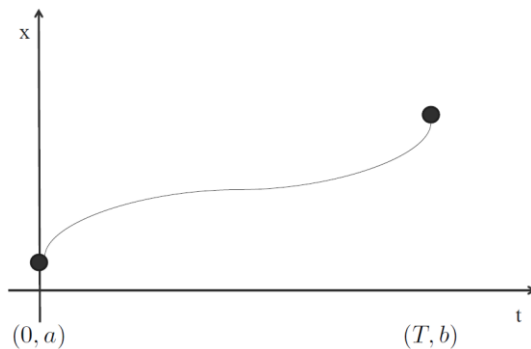
# Boundary Value Problem (BVP)

- BVP is the basis of state sampled lattice planning.
- No general solution. Case by case design.
- Often evolve complicated numerical optimization.



- Design a trajectory  $x(t)$  such that:

- $x(0) = a$
- $x(T) = b$







# Boundary Value Problem (BVP)

- 5<sup>th</sup> order polynomial trajectory:
  - $x(t) = c_5 t^5 + c_4 t^4 + c_3 t^3 + c_2 t^2 + c_1 t + c_0$

- Boundary conditions

	Position	Velocity	Acceleration
t = 0	a	0	0
t = T	b	0	0

- Solve:

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



# Optimal Boundary Value Problem (OBVP)

## Modelling

Objective, minimize the integral of squared jerk:

$$J_{\Sigma} = \sum_{k=1}^3 J_k, \quad J_k = \frac{1}{T} \int_0^T j_k(t)^2 dt.$$

State:  $s_k = (p_k, v_k, a_k)$     Input:  $u_k = j_k$

System model:  $\dot{s}_k = f_s(s_k, u_k) = (v_k, a_k, j_k)$

## Solving

By **Pontryain's minimum principle**, we first introduce the costate:  $\lambda = (\lambda_1, \lambda_2, \lambda_3)$

Define the Hamiltonian function:

$$\begin{aligned} H(s, u, \lambda) &= \frac{1}{T} j^2 + \lambda^T f_s(s, u) \\ &= \frac{1}{T} j^2 + \lambda_1 v + \lambda_2 a + \lambda_3 j \end{aligned}$$

## minimum principle

$\dot{s}^*(t) = f(s^*(t), u^*(t))$ , given:  $s^*(0) = s(0)$

$\lambda(t)$  is the solution of:

$$\dot{\lambda}(t) = -\nabla_s H(s^*(t), u^*(t), \lambda(t))$$

with the boundary condition of:

$$\lambda(T) = -\nabla_h h(s^*(T))$$

and the optimal control input is:

$$u^*(t) = \arg \min_{u(t)} H(s^*(t), u(t), \lambda(t))$$



# Optimal Boundary Value Problem (OBVP)

## Modelling

Objective, minimize the integral of squared jerk:

$$J_{\Sigma} = \sum_{k=1}^3 J_k, \quad J_k = \frac{1}{T} \int_0^T j_k(t)^2 dt.$$

State:  $s_k = (p_k, v_k, a_k)$     Input:  $u_k = j_k$

System model:  $\dot{s} = f_s(s, u) = (v, a, j)$

## Solving

By **Pontryain's minimum principle**, we first introduce the costate:  $\lambda = (\lambda_1, \lambda_2, \lambda_3)$

Define the Hamiltonian function:

$$\begin{aligned} H(s, u, \lambda) &= \frac{1}{T} j^2 + \lambda^T f_s(s, u) \\ &= \frac{1}{T} j^2 + \lambda_1 v + \lambda_2 a + \lambda_3 j \end{aligned}$$

## minimum principle

$\dot{s}^*(t) = f(s^*(t), u^*(t))$ , given:  $s^*(0) = s(0)$

$\lambda(t)$  is the solution of:

$$\dot{\lambda}(t) = -\nabla_s H(s^*(t), u^*(t), \lambda(t))$$

with the boundary condition of:

$$\lambda(T) = -\nabla_h h(s^*(T))$$

and the optimal control input is:

$$u^*(t) = \arg \min_{u(t)} H(s^*(t), u(t), \lambda(t))$$



# Pontryagin's minimum principle

Generally:

$$J = \underbrace{h(s(T))}_{\text{final state}} + \underbrace{\int_0^T g(s(t), u(t)) \cdot dt}_{\text{transition cost}}$$

final state

transition cost

Write the Hamiltonian and costate:

$$H(s, u, \lambda) = g(s, u) + \lambda^T f(s, u)$$

$$\lambda = (\lambda_1, \lambda_2, \lambda_3)$$

We have



## minimum principle

$$\dot{s}^*(t) = f(s^*(t), u^*(t)), \text{ given: } s^*(0) = s(0)$$

$\lambda(t)$  is the solution of:

$$\dot{\lambda}(t) = -\nabla_s H(s^*(t), u^*(t), \lambda(t))$$

with the boundary condition of:

$$\lambda(T) = -\nabla h(s^*(T))$$

and the optimal control input is:

$$u^*(t) = \arg \min_{u(t)} H(s^*(t), u(t), \lambda(t))$$

Suppose:

$s^*$ : Optimal state

$u^*$ : Optimal input



# Optimal Boundary Value Problem (OBVP)

## Modelling

Objective, minimize the integral of squared jerk:

$$J_{\Sigma} = \sum_{k=1}^3 J_k, \quad J_k = \frac{1}{T} \int_0^T j_k(t)^2 dt.$$

State:  $s_k = (p_k, v_k, a_k)$  Input:  $j_k$

System equation:  $\dot{s} = f_s(s, u) = (v, a, j)$

## Solving

By Pontryain's minimum principle, we first introduce the costate:  $\lambda = (\lambda_1, \lambda_2, \lambda_3)$

Define the Hamiltonian function:

$$\begin{aligned} H(s, u, \lambda) &= \frac{1}{T} j^2 + \lambda^T f_s(s, u) \\ &= \frac{1}{T} j^2 + \lambda_1 v + \lambda_2 a + \lambda_3 j \end{aligned}$$

$$\dot{\lambda} = -\nabla_s H(s^*, u^*, \lambda) = (0, -\lambda_1, -\lambda_2)$$

Optimal state

Optimal input

The costate is solved as:

$$\lambda(t) = \frac{1}{T} \begin{bmatrix} -2\alpha \\ 2\alpha t + 2\beta \\ -\alpha t^2 - 2\beta t - 2\gamma \end{bmatrix}$$

The optimal input is solved as:

$$\begin{aligned} u^*(t) &= j^*(t) = \arg \min_{j(t)} H(s^*(t), j(t), \lambda(t)) \\ &= \frac{1}{2} \alpha t^2 + \beta t + \gamma \end{aligned}$$

The optimal state trajectory is solved as:

$$s^*(t) = \begin{bmatrix} \frac{\alpha}{120} t^5 + \frac{\beta}{24} t^4 + \frac{\gamma}{6} t^3 + \frac{a_0}{2} t^2 + v_0 t + p_0 \\ \frac{\alpha}{24} t^4 + \frac{\beta}{6} t^3 + \frac{\gamma}{2} t^2 + a_0 t + v_0 \\ \frac{\alpha}{6} t^3 + \frac{\beta}{2} t^2 + \gamma t + a_0 \end{bmatrix}$$

Initial state:  $s(0) = (p_0, v_0, a_0)$



# Optimal Boundary Value Problem (OBVP)

The cost:

$$J = \gamma^2 + \beta\gamma T + \frac{1}{3}\beta^2 T^2 + \frac{1}{3}\alpha\gamma T^2 + \frac{1}{4}\alpha\beta T^3 + \frac{1}{20}\alpha^2 T^4$$

$\alpha, \beta, \gamma$  is solved as:

$$\begin{bmatrix} \frac{1}{120}T^5 & \frac{1}{24}T^4 & \frac{1}{6}T^3 \\ \frac{1}{24}T^4 & \frac{1}{6}T^3 & \frac{1}{2}T^2 \\ \frac{1}{6}T^3 & \frac{1}{2}T^2 & T \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} = \begin{bmatrix} \Delta p \\ \Delta v \\ \Delta a \end{bmatrix}$$

$$\begin{bmatrix} \Delta p \\ \Delta v \\ \Delta a \end{bmatrix} = \begin{bmatrix} p_f - p_0 - v_0 T - \frac{1}{2}a_0 T^2 \\ v_f - v_0 - a_0 T \\ a_f - a_0 \end{bmatrix}$$



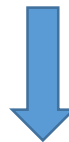
$$\begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} = \frac{1}{T^5} \begin{bmatrix} 720 & -360T & 60T^2 \\ -360T & 168T^2 & -24T^3 \\ 60T^2 & -24T^3 & 3T^4 \end{bmatrix} \begin{bmatrix} \Delta p \\ \Delta v \\ \Delta a \end{bmatrix}$$

This derivation holds for fixed final state:  $s(T) = (p_f, v_f, a_f)$

Similar solution can also be found when  $s(T)$  is partially defined

Same solving process holds for  $J_k = \int_0^T j_k(t)^2 dt + T$ .

$J$  only depends on  $T$ , and the boundary states (known), so we can even get an optimal  $T$ !



How?

Polynomial function root finding problem.



# Optimal Boundary Value Problem (OBVP)

- Previous slides are about fixed final state problem.
- How about the final state is (partially)-free?
  - Did you notice where is the boundary condition?

$$\lambda(t) = -\nabla h(s^*(t))$$

For fixed final state problem:

$$h(s(T)) = \begin{cases} 0, & \text{if } s = s(T) \\ \infty, & \text{otherwise} \end{cases} \quad \text{Not differentiable}$$

So we discard this condition, and use given  $x(T)$  to directly solve for unknown variables

$$s^*(t) = \begin{bmatrix} \frac{\alpha}{120}t^5 + \frac{\beta}{24}t^4 + \frac{\gamma}{6}t^3 + \frac{a_0}{2}t^2 + v_0t + p_0 \\ \frac{\alpha}{24}t^4 + \frac{\beta}{6}t^3 + \frac{\gamma}{2}t^2 + a_0t + v_0 \\ \frac{\alpha}{6}t^3 + \frac{\beta}{2}t^2 + \gamma t + a_0 \end{bmatrix}$$

For (partially)-free final state problem:

$$\text{given } s_i(T), i \in I$$

We have boundary condition for other costate:

$$\lambda_j(T) = \frac{\partial h(s^*(T))}{\partial s_j}, \text{ for } j \neq i$$

Then we solve this problem again.



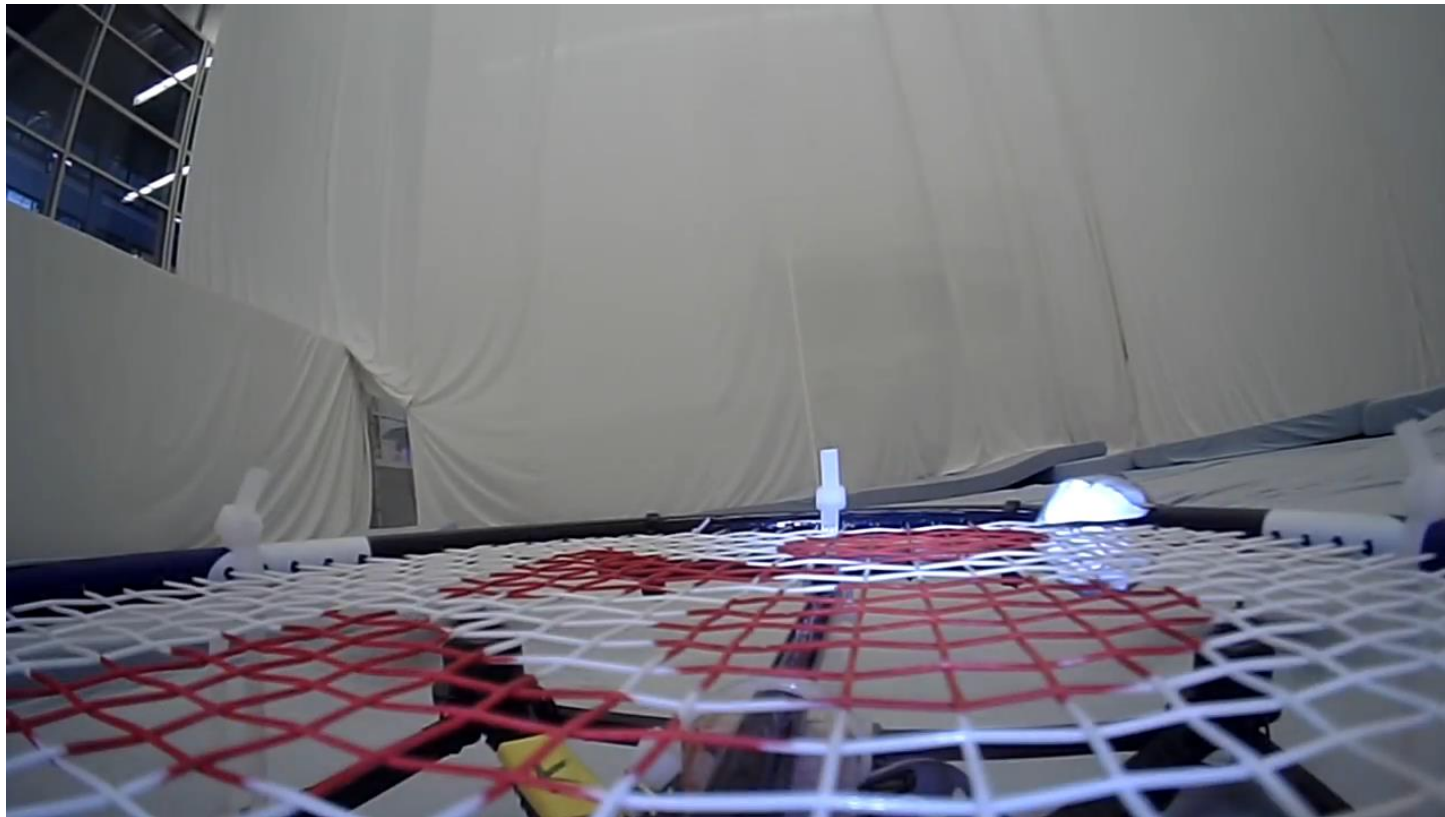
# Application







# Application

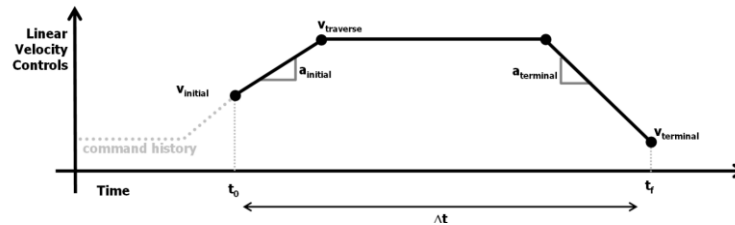




# Another example



- Parametrize the control input
  - $\omega(t) = a + bt + ct^2 + dt^3 + \dots$
  - $v(t) =$

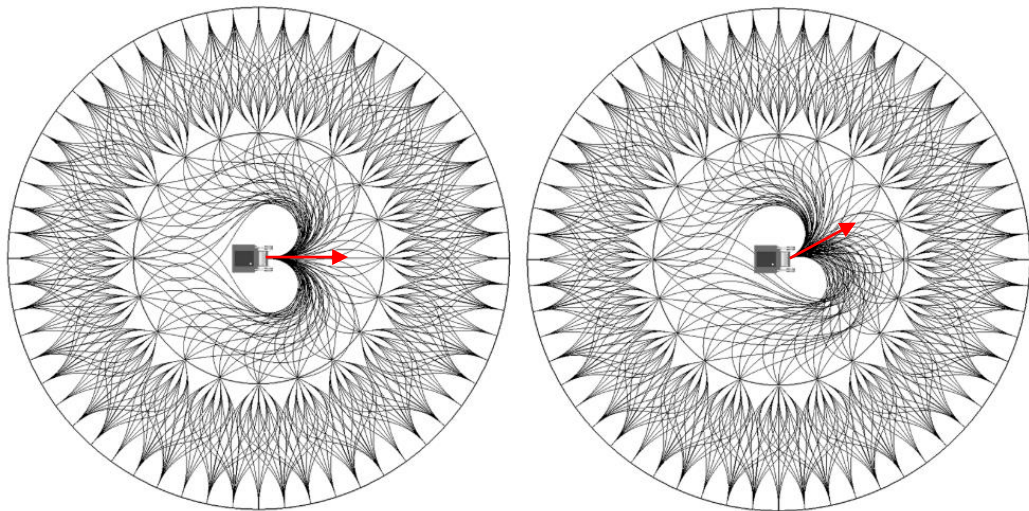


- Constrained Trajectory Generation
  - Numerical difference evaluated Jacobian
- 
- Offline BVP, trajectory generation.
  - Online search.

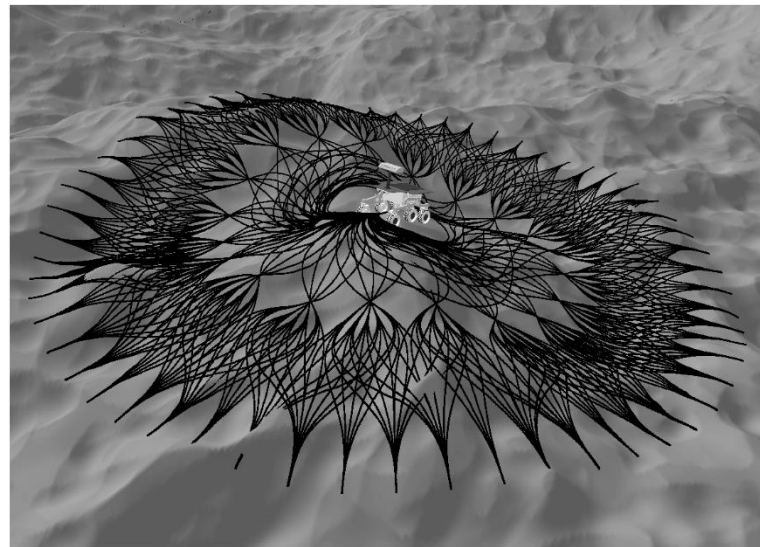
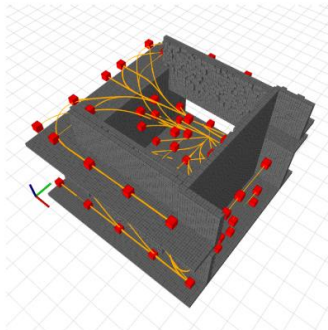
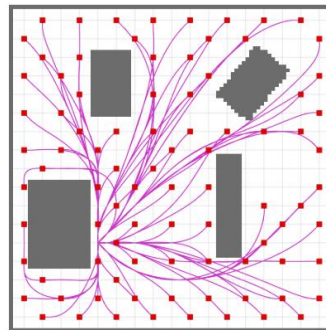




# Graph search problem



- Up to now, it has become a graph search problem.
- Every techniques learned in L2 can be applied here.

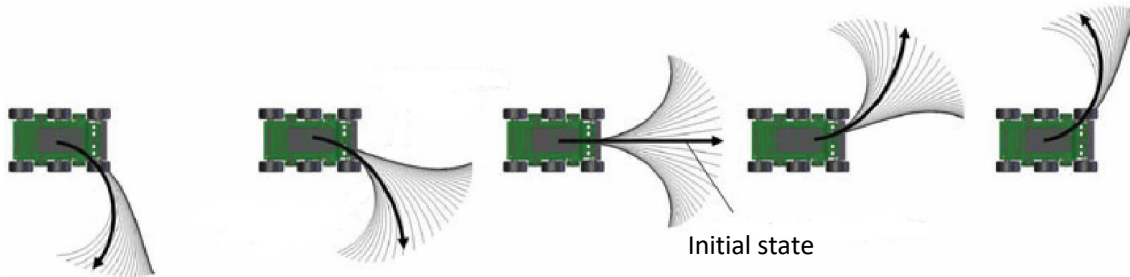




# Trajectory library

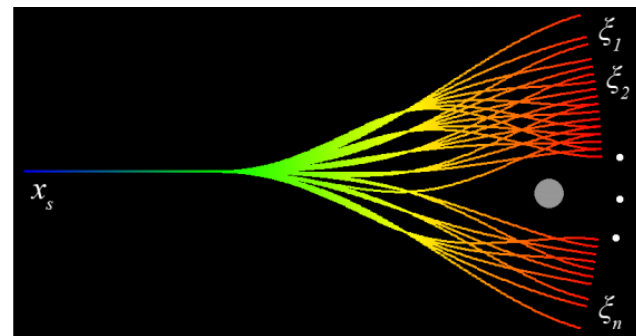
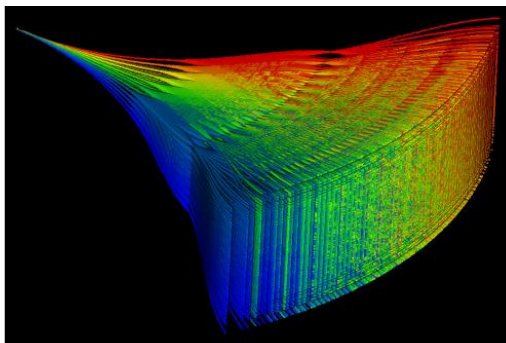
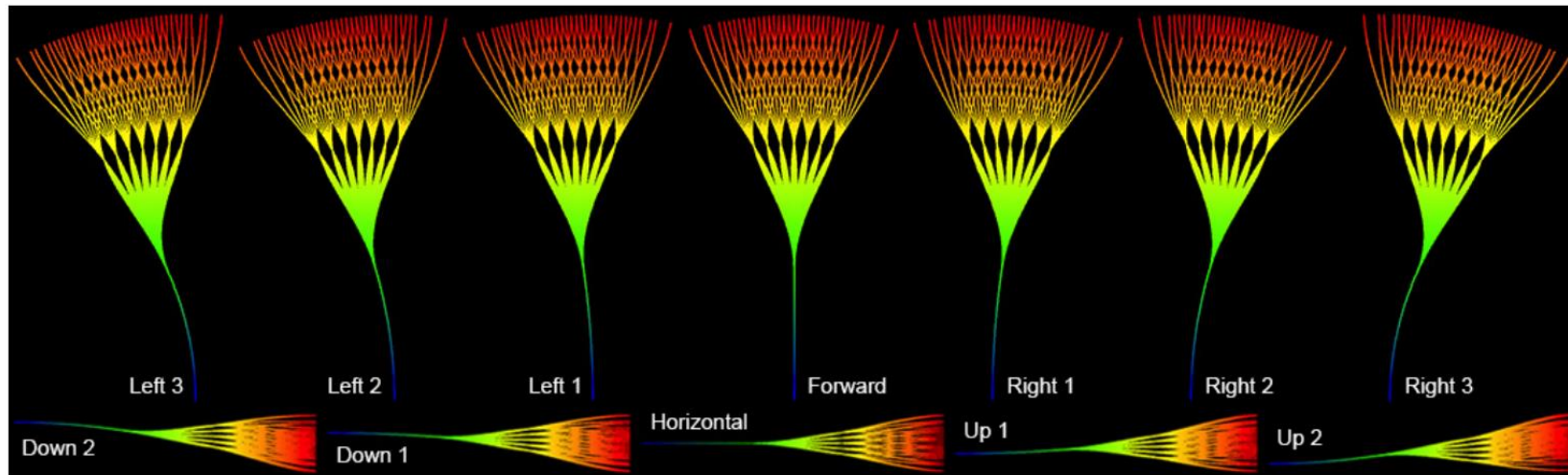
- Single layer lattice planning is a common option for local collision avoidance.
- No graph search, only trajectory selection.
- Rating each trajectory based on a **multi-term** cost function.

collision risk, information acquisition, comfort, energy, ...





# Trajectory library





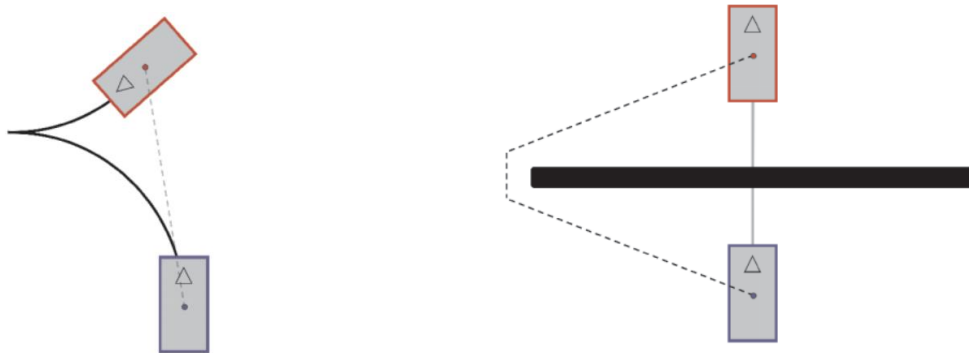
# Heuristic



# Heuristic design

Principle: solve an **easier** problem

- Assume no obstacle existence
- Assume no dynamic existence

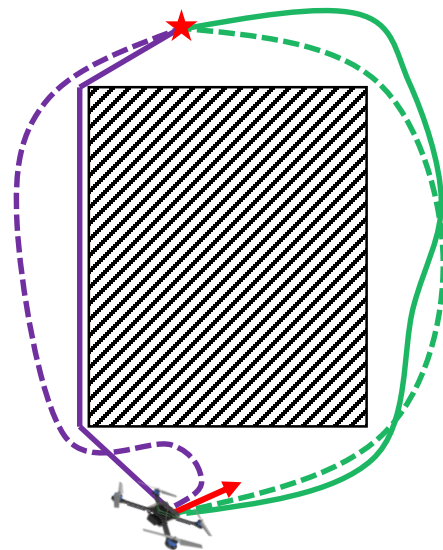
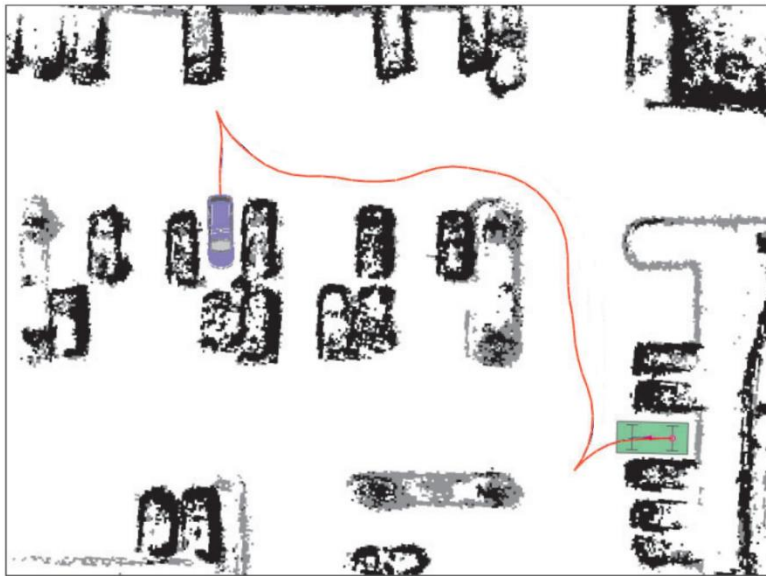






# Heuristic design

For every node (state), Ignoring the dynamic model and search the **shortest path** for it







# Assume no obstacle existence

For every node (state), solve the **OBVP** to the planning target state as heuristic function  $h$

- 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  $g(n)=\text{infinite}$  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  $g(m) = \text{infinite}$ 
      - $g(m)=g(n) + C_{nm}$
      - Push node "m" into the queue
    - If  $g(m) > g(n) + C_{nm}$ 
      - $g(m)=g(n) + C_{nm}$
  - end
- End Loop

Accumulate cost



# Comparison



Euclidean 2D distance



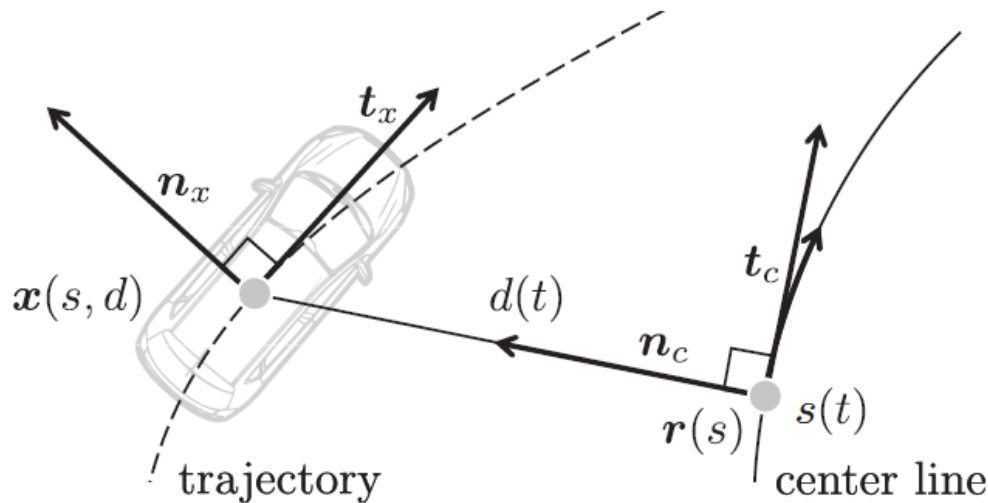
non-holonomic-without-obstacles



# Planning in Frenet-serret Frame



# Frenet-serret frame



- dynamic reference frame.
- lateral and longitudinal independently.
- For lane following problem, the problem is decoupled.

- ✓ Motion/control parametrization:  
quintic polynomial.

$$d(t) = a_{d0} + a_{d1}t + a_{d2}t^2 + a_{d3}t^3 + a_{d4}t^4 + a_{d5}t^5$$

$$s(t) = a_{s0} + a_{s1}t + a_{s2}t^2 + a_{s3}t^3 + a_{s4}t^4 + a_{s5}t^5$$

- ✓ Solve the optimal control problem.

We only discuss the lateral planning here,  
for longitudinal planning, please refer to:

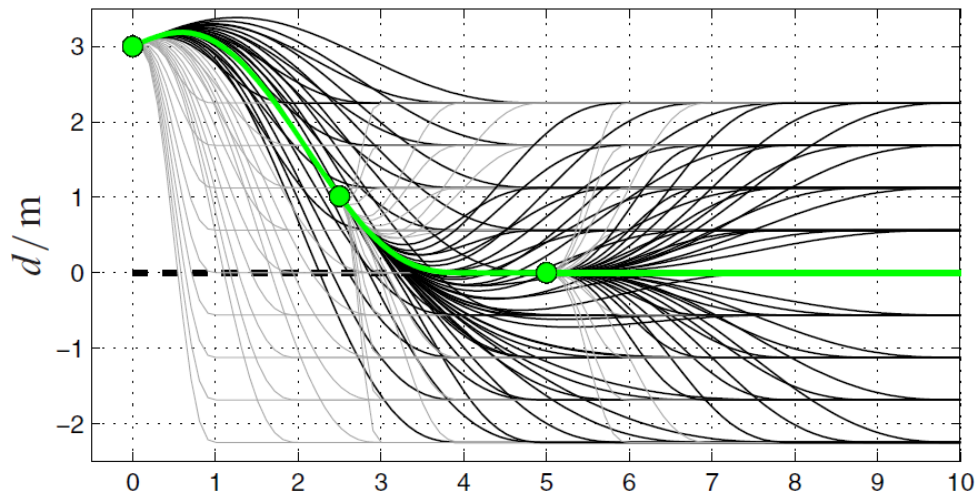


*Optimal Trajectory Generation for Dynamic Street Scenarios in a Frenet Frame*, Moritz Werling, Julius Ziegler, Sören Kammel, and Sebastian Thrun

*Optimal trajectories for time-critical street scenarios using discretized terminal manifolds*, Moritz Werling, Sören Kammel, Julius Ziegler and Lutz Gröll



# Planning in Frenet-serret frame



Lateral trajectory

$$\begin{bmatrix} T^3 & T^4 & T^5 \\ 3T^2 & 4T^3 & 5T^4 \\ 6T & 12T^2 & 20T^3 \end{bmatrix} \begin{bmatrix} a_{d3} \\ a_{d4} \\ a_{d5} \end{bmatrix} = \begin{bmatrix} \Delta p \\ \Delta v \\ \Delta a \end{bmatrix} \quad \begin{bmatrix} \Delta p \\ \Delta v \\ \Delta a \end{bmatrix} = \begin{bmatrix} d_f - (d_0 + \dot{d}_0 T + \frac{1}{2} \ddot{d}_0 T^2) \\ \dot{d}_f - (\dot{d}_0 + \ddot{d}_0 T) \\ \ddot{d}_f - \ddot{d}_0 \end{bmatrix}$$

$$d(t) = a_{d0} + a_{d1}t + a_{d2}t^2 + a_{d3}t^3 + a_{d4}t^4 + a_{d5}t^5$$

Initial condition:

$$D(0) = \begin{pmatrix} d_0 & \dot{d}_0 & \ddot{d}_0 \end{pmatrix}$$

Terminate condition:

$$D(T) = \begin{pmatrix} d_f & \dot{d}_f & \ddot{d}_f \end{pmatrix}$$

Lane following:

$$D(T) = \begin{pmatrix} d_f & 0 & 0 \end{pmatrix}$$

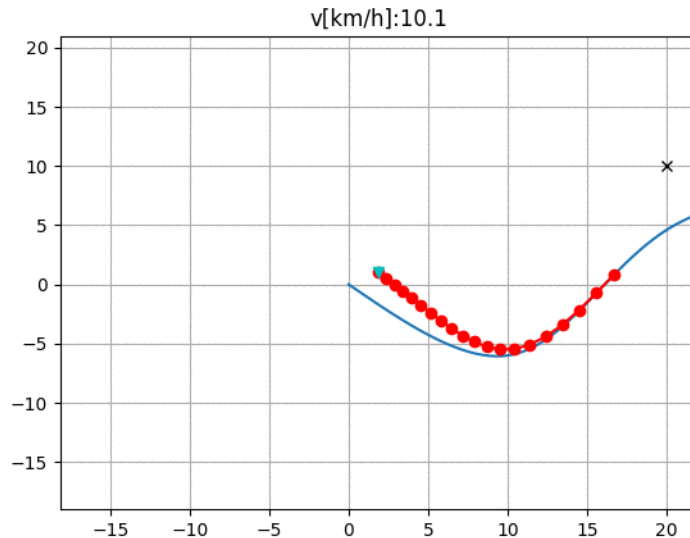
Recall what we learn previously:





# Planning in Frenet-serret frame

## Example



# Hybrid A\*



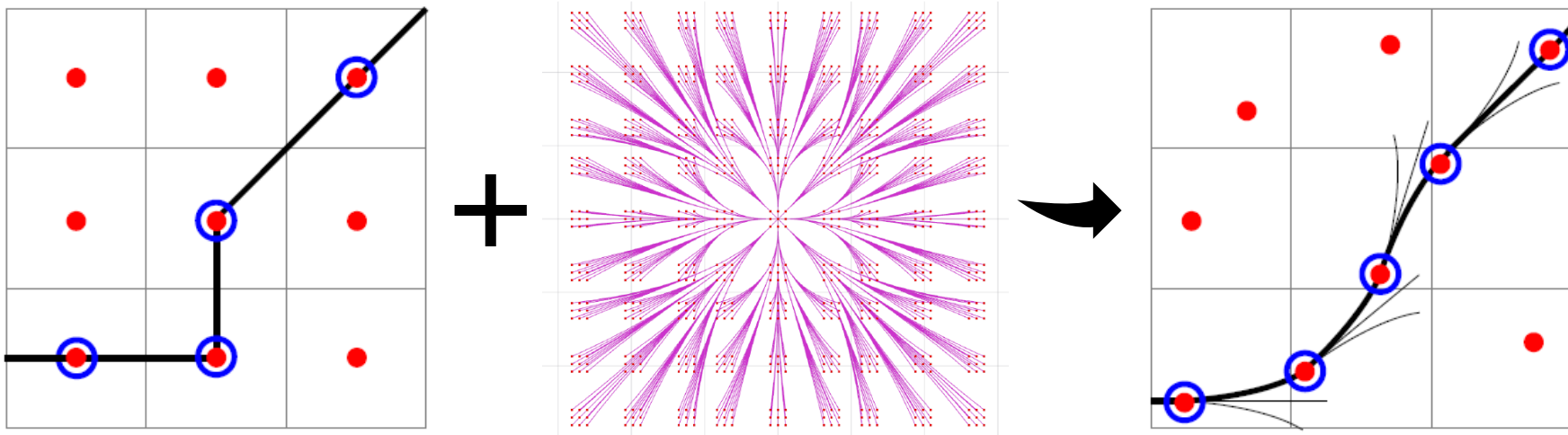
# Workflow





# Basic idea

- Online generate a dense lattice costs too much time.
- How about **prune** some nodes?
- Define a rule to prune: use the grid map.





# Detail

- 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  $g(n)=\text{infinite}$  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  $g(m) = \text{infinite}$ 
      - $g(m) = g(n) + C_{nm}$
      - Push node "m" into the queue
    - If  $g(m) > g(n) + C_{nm}$ 
      - $g(m) = g(n) + C_{nm}$
- end
- End Loop

Choose a proper heuristic according to previous slides

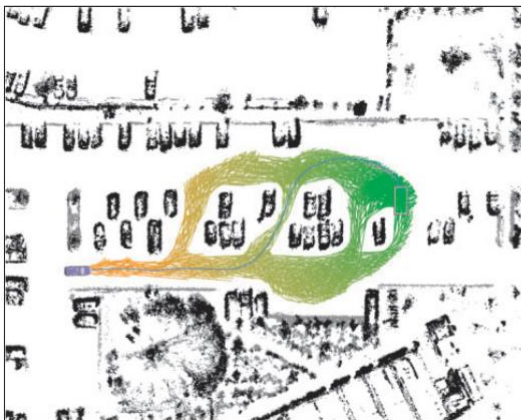
Find neighbors by forward integrating the state in the node.

Record the state inside node "m"

Update the state inside node "m"



# Heuristic design



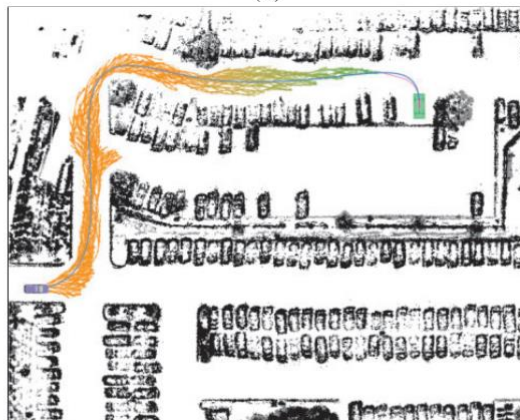
(a)



(b)



(c)



(d)

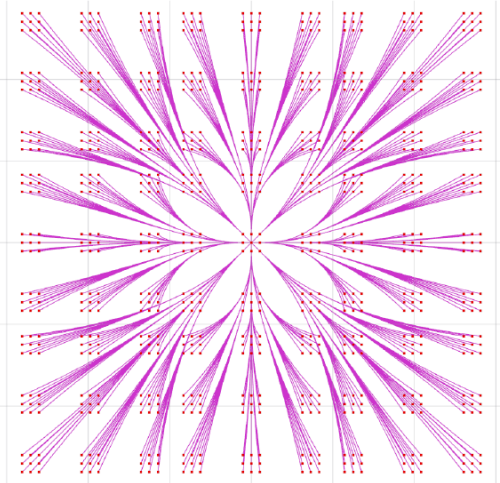
- (a) 2D-Euclidean distance
- (b) non-holonomic-without-obstacles
- (c) non-holonomic-without-obstacles, bad performance in dead ends
- (d) non-holonomic-without-obstacles + holonomic-with-obstacles (2D shortest path)



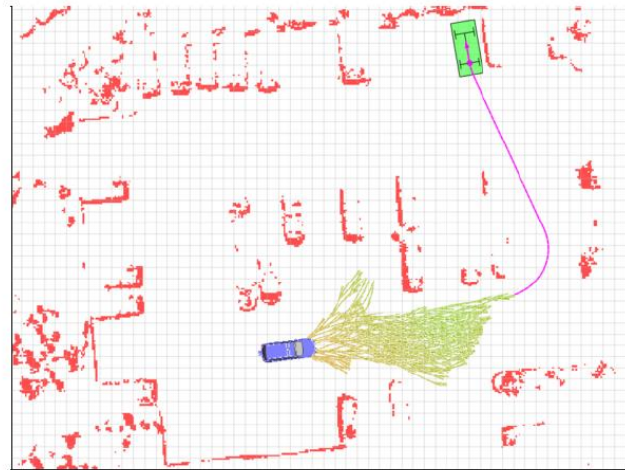
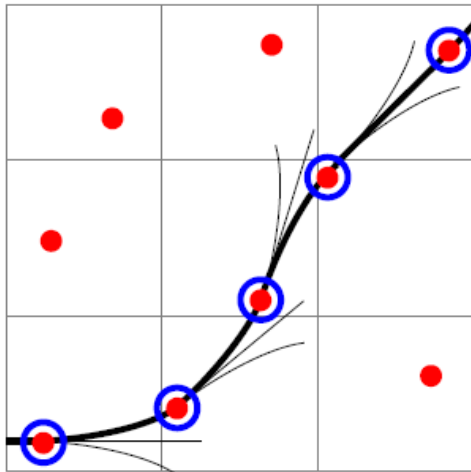
# Other tricks

**Analytic Expansions:** One shot heuristic

Add a state-driven bias towards the searching process



Control space sample (discretization) is kind of low-efficient, since no target biasing is encoded



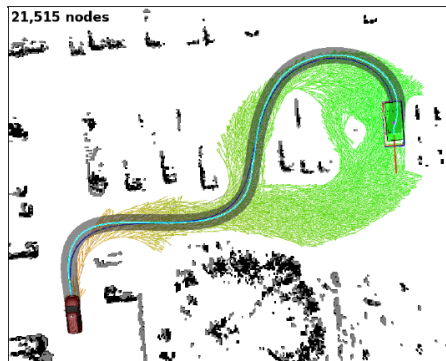
How about we manually add (try) state space sample?



# Application



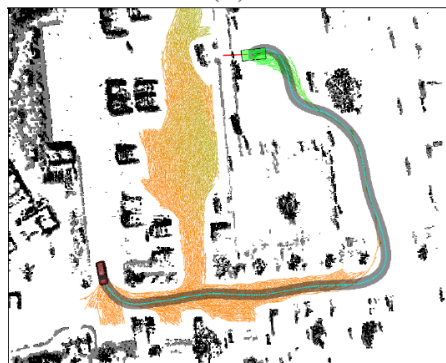
# Autonomous car



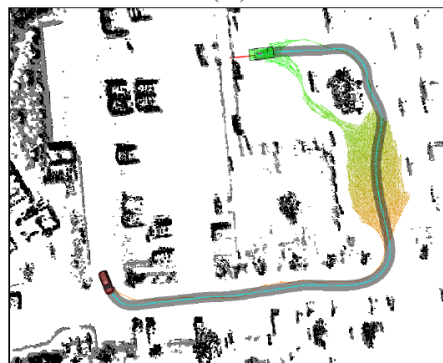
(a)



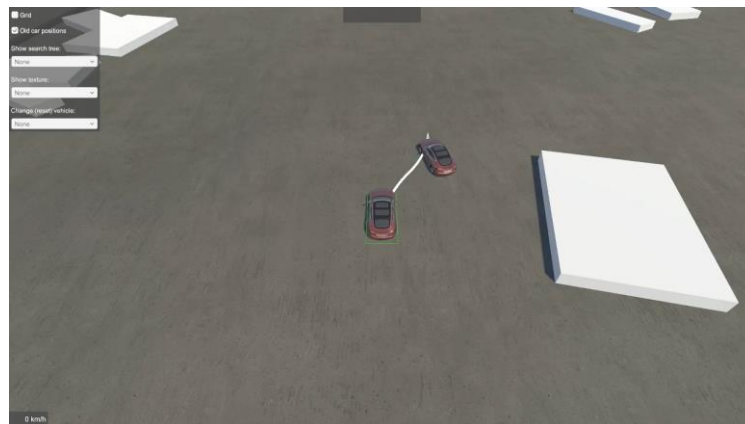
(b)



(c)



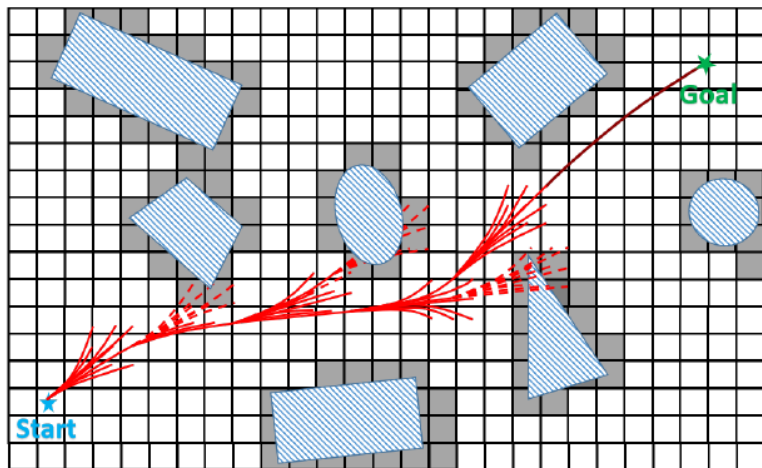
(d)



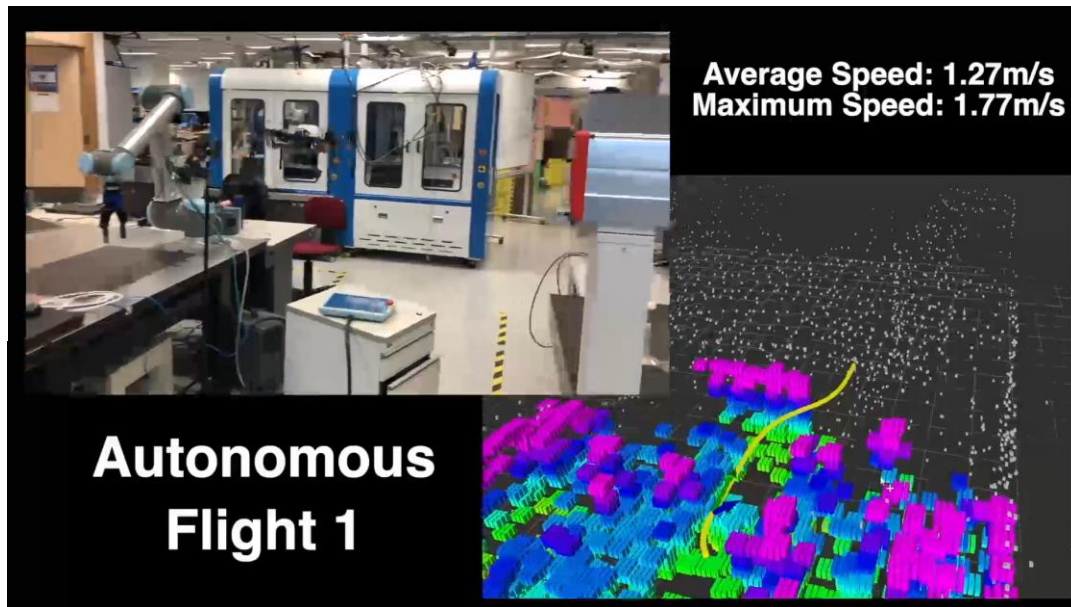




# Autonomous UAV



- As a promising front-end
- Careful engineering considerations
- Linear UAV model: **nilpotent!**
- Sophisticated C++ implementation



*Robust and Efficient Quadrotor Trajectory Generation for Fast Autonomous Flight*, Boyu Zhou, Fei Gao

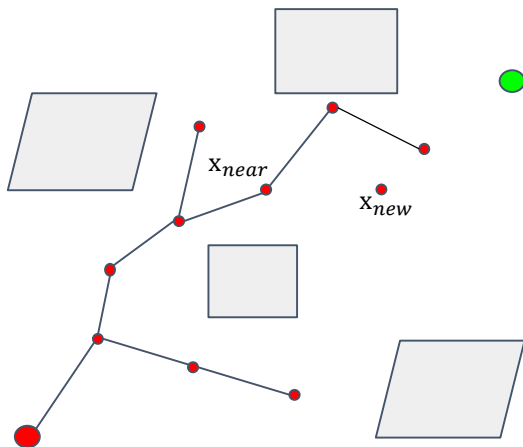
<https://github.com/HKUST-Aerial-Robotics/Fast-Planner>

# Kinodynamic RRT\*





# Review of RRT\*



---

## 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**

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

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

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

**if**  $\text{CollisionFree}(x_{new})$  **then**

$X_{near} \leftarrow \text{NearC}(\mathcal{T}, x_{new})$ ;

$x_{min} \leftarrow \text{ChooseParent}(X_{near}, x_{near}, x_{new})$  ;

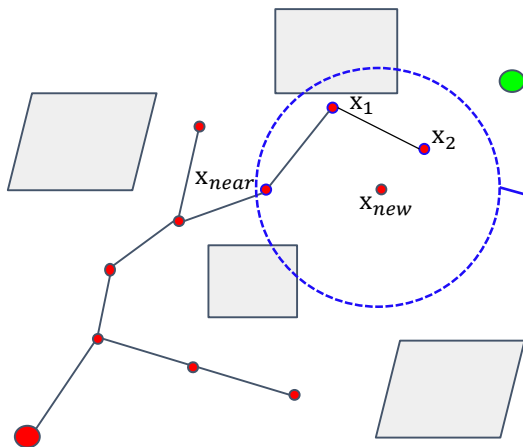
$\mathcal{T}.addNodeEdge(x_{min}, x_{new})$ ;

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

---



# Review of RRT\*



---

## 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**

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

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

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

**if**  $\text{CollisionFree}(x_{new})$  **then**

$X_{near} \leftarrow \text{NearC}(\mathcal{T}, x_{new})$ ;

$x_{min} \leftarrow \text{ChooseParent}(X_{near}, x_{near}, x_{new})$  ;

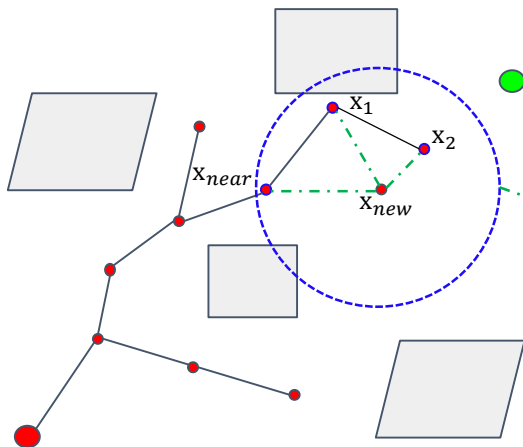
$\mathcal{T}.addNodeEdge(x_{min}, x_{new})$ ;

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

---



# Review of RRT\*



---

## 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**

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

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

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

**if**  $\text{CollisionFree}(x_{new})$  **then**

$X_{near} \leftarrow \text{NearC}(\mathcal{T}, x_{new})$ ;

$x_{min} \leftarrow \text{ChooseParent}(X_{near}, x_{near}, x_{new})$ ;

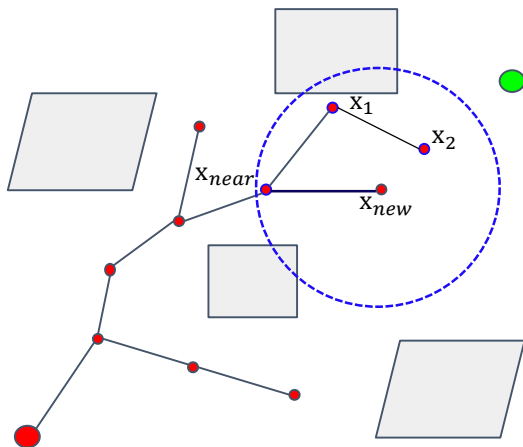
$\mathcal{T}.addNodeEdge(x_{min}, x_{new})$ ;

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

---



# Review of RRT\*



---

## 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**

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

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

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

**if**  $\text{CollisionFree}(x_{new})$  **then**

$X_{near} \leftarrow \text{NearC}(\mathcal{T}, x_{new})$ ;

$x_{min} \leftarrow \text{ChooseParent}(X_{near}, x_{near}, x_{new})$  ;

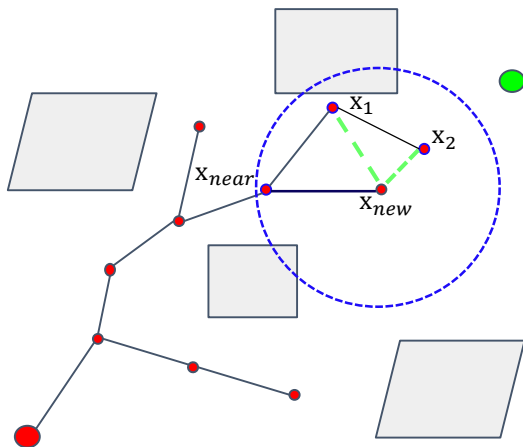
$\mathcal{T}.addNodeEdge(x_{min}, x_{new})$ ;

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

---



# Review of RRT\*



---

## 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**

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

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

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

**if**  $\text{CollisionFree}(x_{new})$  **then**

$X_{near} \leftarrow \text{NearC}(\mathcal{T}, x_{new})$ ;

$x_{min} \leftarrow \text{ChooseParent}(X_{near}, x_{near}, x_{new})$  ;

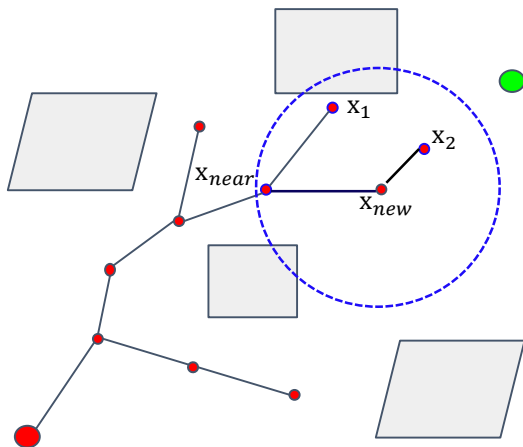
$\mathcal{T}.addNodeEdge(x_{min}, x_{new})$ ;

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

---



# Review of RRT\*



---

## 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**

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

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

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

**if**  $\text{CollisionFree}(x_{new})$  **then**

$X_{near} \leftarrow \text{NearC}(\mathcal{T}, x_{new})$ ;

$x_{min} \leftarrow \text{ChooseParent}(X_{near}, x_{near}, x_{new})$  ;

$\mathcal{T}.addNodeEdge(x_{min}, x_{new})$ ;

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

---



## Workflow

Similar to RRT\* but different in details

---

### 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**

$x_{rand} \leftarrow Sample(E);$

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

$x_{min} \leftarrow ChooseParent(X_{near}, x_{rand});$

$T.addNode(x_{rand});$

$T.rewire();$

---



# Problems when it comes to motion constraints

## 1. How to “Sample”

---

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 ← Sample(E);

    X\_near ← Near(T, x\_rand);

    x\_min ← ChooseParent(X\_near, x\_rand);

    T.addNode(x\_rand);

    T.rewire();

---

LTI system state-space equation:

$$\dot{x}(t) = Ax(t) + Bu(t) + c$$

For example for double integrator systems,

$$x = \begin{bmatrix} p \\ v \end{bmatrix}, A = \begin{bmatrix} 0 & I \\ 0 & 0 \end{bmatrix}, B = \begin{bmatrix} 0 \\ I \end{bmatrix}$$

Instead of sampling in Euclidean space like RRT, it requires to **sample in full state space**.





# Problems when it comes to motion constraints

## 2. How to define “Near”

---

Kinodynamic RRT\*

---

**Input:**  $E$ ,  $x_{\text{init}}$ ,  $x_{\text{goal}}$

**Output:** A trajectory  $T$  from  $x_{\text{init}}$  to  $x_{\text{goal}}$

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

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

$x_{\text{rand}} \leftarrow \text{Sample}(E);$

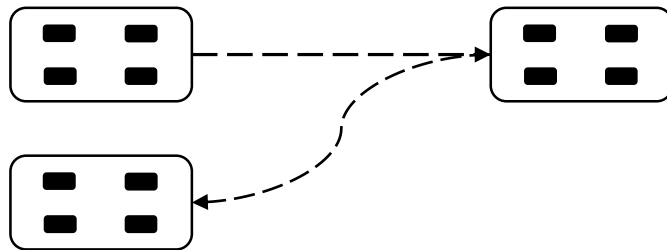
$X_{\text{near}} \leftarrow \text{Near}(T, x_{\text{rand}});$

$x_{\text{min}} \leftarrow \text{ChooseParent}(X_{\text{near}}, x_{\text{rand}});$

$T.\text{addNode}(x_{\text{rand}});$

$T.\text{rewire}();$

---



A car can not move sideways

If without motion constraints, Euclidean distance or Manhattan distance can be used.

In state space with motion constraints, bringing in **optimal control**.



## Problems when it comes to motion constraints

### 2. How to define “Near”

If bring optimal control, we can define **cost functions** of transferring from states to states.

$$c[\pi] = \int_0^{\tau} (1 + u(t)^T R u(t)) dt$$

Typically, a quadratic form of time-energy optimal is adopted.

**Two states are near if the cost of transferring from one state to the other is small.**  
(Note that the cost may be different if transfer reversely)



## Problems when it comes to motion constraints

### 2. How to define “Near”

$$c[\pi] = \int_0^{\tau} (1 + u(t)^T R u(t)) dt$$

If we know the arriving time  $\tau$  and the control policy  $u(t)$  of transferring, we can calculate the cost.

And thankfully, it's all in classic optimal control solutions.(OBVP)



# Unbounded Optimal control solutions

## 2.1 Fixed final state $x_1$ , fixed final time $\tau$

optimal control policy  $u^*(t)$

$$u^*(t) = R^{-1}B^T \exp[A^T(\tau - t)]G(\tau)^{-1}[x_1 - \bar{x}(\tau)].$$

Where  $G(t)$  is the weighted controllability Gramian:

$$G(t) = \int_0^t \exp[A(t - t')]BR^{-1}B^T \exp[A^T(t - t')]dt'.$$

Which is the solution to the Lyapunov equation:

$$\dot{G}(t) = AG(t) + G(t)A^T + BR^{-1}B^T, G(0) = 0.$$



# Unbounded Optimal control solutions

## 2.1 Fixed final state $x_1$ , fixed final time $\tau$

$$u^*(t) = R^{-1}B^T \exp[A^T(\tau - t)]G(\tau)^{-1}[x_1 - \bar{x}(\tau)].$$

And  $\bar{x}(t)$  describe what the state  $x$  would be at time  $t$  if no control input were applied:

$$\bar{x}(t) = \exp(At)x_0 + \int_0^t \exp[A(t - t')]cdt'.$$

Which is the solution to the differential equation:

$$\dot{\bar{x}}(t) = A\bar{x}(t) + c, \bar{x}(0) = x_0$$



# Unbounded Optimal control solutions

## 2.2 Fixed final state $x_1$ , free final time $\tau$

If we want to find the optimal arrival time  $\tau$ , we do this by filling in the control policy  $u^*(t)$  into the cost function  $c[\pi]$  and evaluating the integral:

$$c[\tau] = \tau + [x_1 - \bar{x}(\tau)]^T G(t)^{-1} [x_1 - \bar{x}(\tau)].$$

The optimal  $\tau$  is found by taking the derivative of  $c[\tau]$  with respect to  $\tau$ :

$$\dot{c}[\tau] = 1 - 2(Ax_1 + c)^T d(\tau) - d(\tau)^T B R^{-1} B^T d(\tau).$$

Where

$$d(\tau) = G(t)^{-1} [x_1 - \bar{x}(\tau)].$$



# Unbounded Optimal control solutions

## 2.2 Fixed final state $x_1$ , free final time $\tau$

Solve  $\dot{c}[\tau] = 0$  for  $\tau^*$ .

Noted that the function  $c[\tau]$  may have multiple local minima.

And for a double integrator system, it's a 4<sup>th</sup> order polynomial.

Given the optimal arrival time  $\tau^*$  as defined above, it again turns into a fixed final state, fixed final time problem.



## Problems when it comes to motion constraints

### 3. How to “ChooseParent”

---

Kinodynamic RRT\*

---

**Input:**  $E$ ,  $x_{\text{init}}$ ,  $x_{\text{goal}}$

**Output:** A trajectory  $T$  from  $x_{\text{init}}$  to  $x_{\text{goal}}$

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

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

$x_{\text{rand}} \leftarrow \text{Sample}(E);$

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

$x_{\text{min}} \leftarrow \text{ChooseParent}(X_{\text{near}}, x_{\text{rand}});$

$T.\text{addNode}(x);$

$T.\text{rewire}();$

---

Now if we sample a random state, we can calculate control policy and cost from those state-nodes in the tree to the sampled state.

Choose one with the minimal cost and **check  $x(t)$  and  $u(t)$  are in bounds**.

If no qualified parent found, sample another state.





# Problems when it comes to motion constraints

## 4. How to find near nodes efficiently

---

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\_rand);

    T.addNode(x);

    T.rewire();

---

Every time we sample a random state *x\_rand*, it requires to check every node in the tree to find its parent, that is solving a OBVP for each node, which is not efficient.



## Problems when it comes to motion constraints

### 4. How to find near nodes efficiently

---

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**

$x_{rand} \leftarrow \text{Sample}(E)$ ;

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

$x_{min} \leftarrow \text{ChooseParent}(X_{near}, x_{rand})$ ;

    T.addNode(x);

    T.rewire();

---

If we set a **cost tolerance**  $r$ , we can actually calculate bounds of the states (forward-reachable set) that can be reached by  $x_{rand}$  and bounds of the states (backward-reachable set) that can reach  $x_{rand}$  with cost less than  $r$ .

And if we store nodes in form of a kd-tree, we can then do range query in the tree.



## Problems when it comes to motion constraints

### 4. How to find near nodes efficiently

$$c[\tau] = \tau + [x_1 - \bar{x}(\tau)]^T G(t)^{-1} [x_1 - \bar{x}(\tau)].$$

This formula describes how cost of transferring from state  $x_0$  to state  $x_1$  changes with arrival time  $\tau$ .

We can see that given initial state  $x_0$ , cost tolerance  $r$  and arrival time  $\tau$ , the forward-reachable set of  $x_0$  is:

$$\begin{aligned} & \{x_1 \mid \tau + [x_1 - \bar{x}(\tau)]^T G(t)^{-1} [x_1 - \bar{x}(\tau)] < r\} \\ &= \left\{x_1 \mid [x_1 - \bar{x}(\tau)]^T \frac{G(t)^{-1}}{r - \tau} [x_1 - \bar{x}(\tau)] < 1\right\}. \\ &= \mathcal{E}[\bar{x}(\tau), G(t)(r - \tau)]. \end{aligned}$$



## Problems when it comes to motion constraints

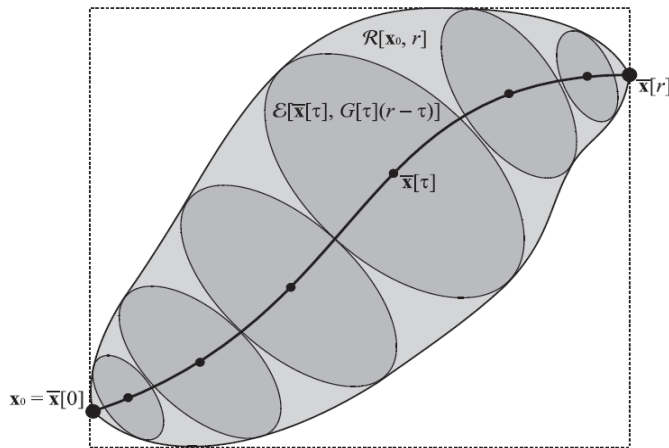
### 4. How to find near nodes efficiently

$$\begin{aligned} & \{x_1 \mid \tau + [x_1 - \bar{x}(\tau)]^T G(t)^{-1} [x_1 - \bar{x}(\tau)] < r\} \\ &= \left\{x_1 \mid [x_1 - \bar{x}(\tau)]^T \frac{G(t)^{-1}}{r - \tau} [x_1 - \bar{x}(\tau)] < 1\right\}. \\ &= \mathcal{E}[\bar{x}(\tau), G(t)(r - \tau)]. \end{aligned}$$

where  $\mathcal{E}[x, M]$  is an **ellipsoid** with center  $x$  and positive definite weight matrix  $M$ , formally defined as:

$$\mathcal{E}[x, M] = \{x' \mid (x' - x)^T M^{-1} (x' - x) < 1\}.$$

Hence, the forward-reachable set is the union of high dimensional ellipsoids for all possible arrival times  $\tau$ .



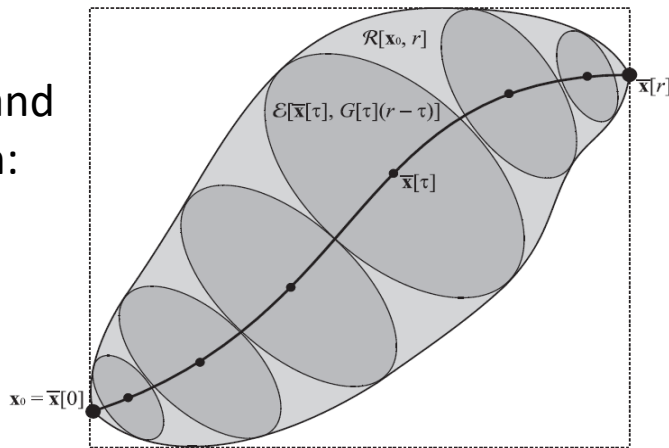


# Problems when it comes to motion constraints

## 4. How to find near nodes efficiently

For simplification, we sample several  $\tau$ s and calculate axis-aligned bounding box of the ellipsoids for each  $\tau$  and update the maximum and minimum in each dimension:

$$\prod_{k=1}^n \left[ \begin{array}{l} \min\{0 < \tau < r\}(\bar{x}(\tau)_k - \sqrt{G[\tau]_{(k,k)}(r - \tau)}), \\ \max\{0 < \tau < r\}(\bar{x}(\tau)_k + \sqrt{G[\tau]_{(k,k)}(r - \tau)}) \end{array} \right].$$



Similar for the calculation of the backward-reachable set.



# Problems when it comes to motion constraints

## 4. How to find near nodes efficiently

---

Kinodynamic RRT\*

---

**Input:**  $E$ ,  $x_{\text{init}}$ ,  $x_{\text{goal}}$

**Output:** A trajectory  $T$  from  $x_{\text{init}}$  to  $x_{\text{goal}}$

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

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

$x_{\text{rand}} \leftarrow \text{Sample}(E);$

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

$x_{\text{min}} \leftarrow \text{ChooseParent}(X_{\text{near}}, x_{\text{rand}});$

$T.\text{addNode}(x);$

$T.\text{rewire}();$

---

When do “Near” query and “ChooseParent”,  $X_{\text{near}}$  can be found from the backward-reachable set of  $x_{\text{rand}}$ .



# Problems when it comes to motion constraints

## 5. How to “Rewire”

---

Kinodynamic RRT\*

---

**Input:**  $E$ ,  $x_{\text{init}}$ ,  $x_{\text{goal}}$

**Output:** A trajectory  $T$  from  $x_{\text{init}}$  to  $x_{\text{goal}}$

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

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

$x_{\text{rand}} \leftarrow \text{Sample}(E);$

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

$x_{\text{min}} \leftarrow \text{ChooseParent}(X_{\text{near}}, x_{\text{rand}});$

$T.\text{addNode}(x);$

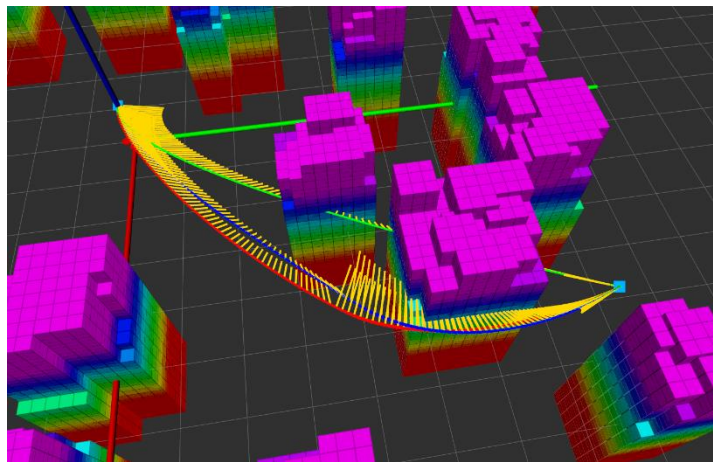
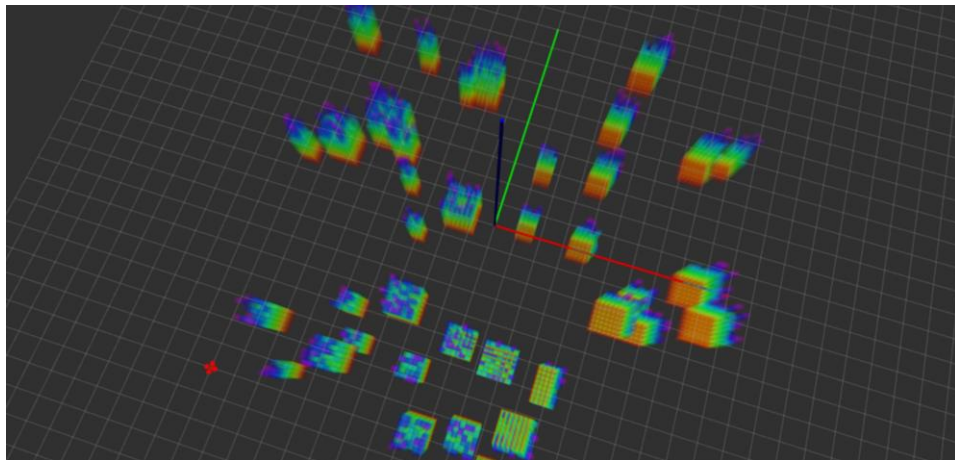
$T.\text{rewire}();$

---

When “Rewire”, we calculate the **forward-reachable set** of  $x_{\text{rand}}$ , and solve OBVPs.



## Demos



The green curve takes no account of the obstacles;  
The red curve is the result of the kinodynamic trajectory planner;  
The blue curve is the first feasible trajectory found by the kinodynamic trajectory planner;  
The yellow lines are the control inputs in every control points.



# Homework



# Local lattice planner

## Homework 1

- For the OBVP problem stated in slides p.25-p.29, please get the optimal solution (control, state, and time) for **partially free final state** case.
- Suppose the position is fixed, velocity and acceleration are free here.



# Local lattice planner

## Homework 2

- Build an ego-graph of the linear modeled robot.
- Select the best trajectory closest to the planning target.



**Thanks for Listening!**

