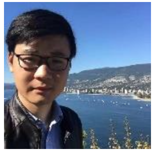
深蓝学院
shenlanxueyuan.com

## Lecture 4

# KINODYNAMIC PATH FINDING



主讲人 Fei Gao

Ph.D. in Robotics
Hong Kong University of Science and Technology
Assistant Professor，Zhejiang University

# 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

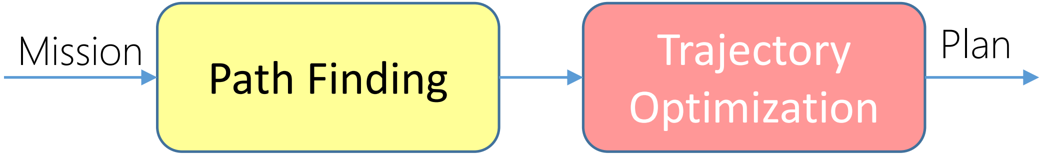
- Differentially constrained
- Up to force (acceleration)

# Why kinodynamic planning ?

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

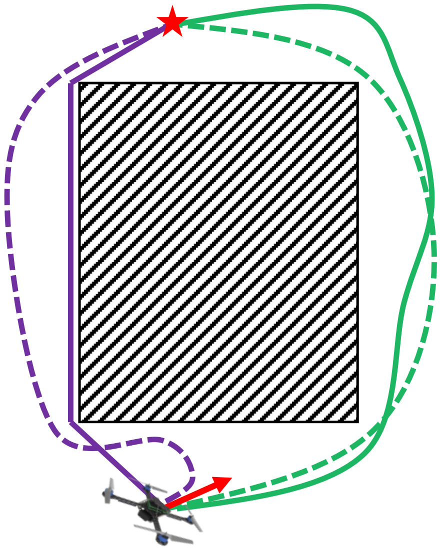

Ask: We have the back-end optimization, why kinodynamic planning?

Recall the old-school pipeline:



# Why kinodynamic planning ?

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

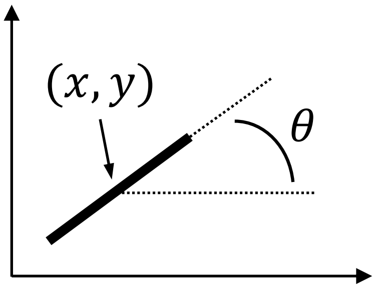


# Examples

Unicycle and differential drive models:

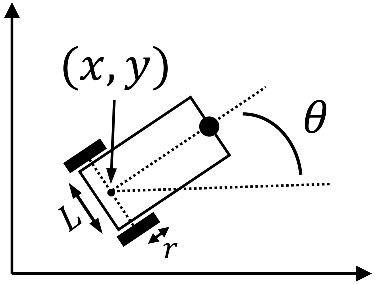
$$\begin{pmatrix}\dot{x}\\ \dot{y}\\ \dot{\theta}\end{pmatrix} = \begin{pmatrix}cos\theta\\ sin\theta\\ 0\end{pmatrix}\cdot v + \begin{pmatrix}0\\ 0\\ 1\end{pmatrix}\cdot \omega$$

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




Unicycle

$|v| \leq v_{max}$
$|\omega| \leq \omega_{max}$




Differential drive

$|\omega_l| \leq \omega_{l,max}$
$|\omega_r| \leq \omega_{r,max}$

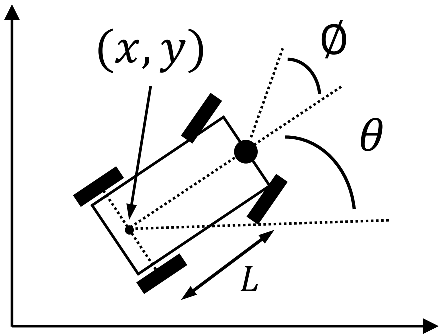
$$v = \frac{r}{2}(\omega_l + \omega_r)cos\theta \quad \omega = \frac{r}{L}(\omega_r - \omega_l)$$

http://planning.cs.uiuc.edu/node659.html

# Examples

Simplified car model

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




$|v| \leq v_{max}, \quad |\emptyset| \leq \emptyset_{max} < \frac{\pi}{2}$ → Simple car model

$v \in \{-v_{max}, v_{max}\}, \quad |\emptyset| \leq \emptyset_{max} < \frac{\pi}{2}$ → Reeds & Shepp's car

$v = v_{max}, \quad |\emptyset| \leq \emptyset_{max} < \frac{\pi}{2}$ → Dubin's car

http://planning.cs.uiuc.edu/node658.html

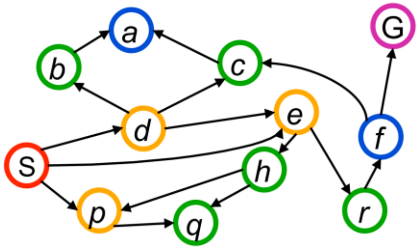
# State Lattice Planning

# Workflow

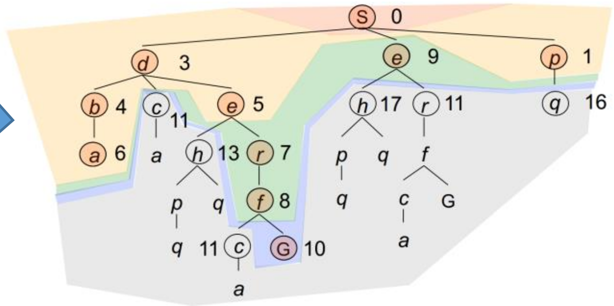
# Basic idea

- 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)=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) = infinite
      - g(m)= g(n) + Cnm
      - Push node "m" into the queue
    - If $g(m) > g(n) + C_{nm}$
      - g(m)= g(n) + Cnm
  - 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.

- 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

- This is the basic motivation for all kinodyanmic planning.
- State lattice planning is the most straight-forward one.

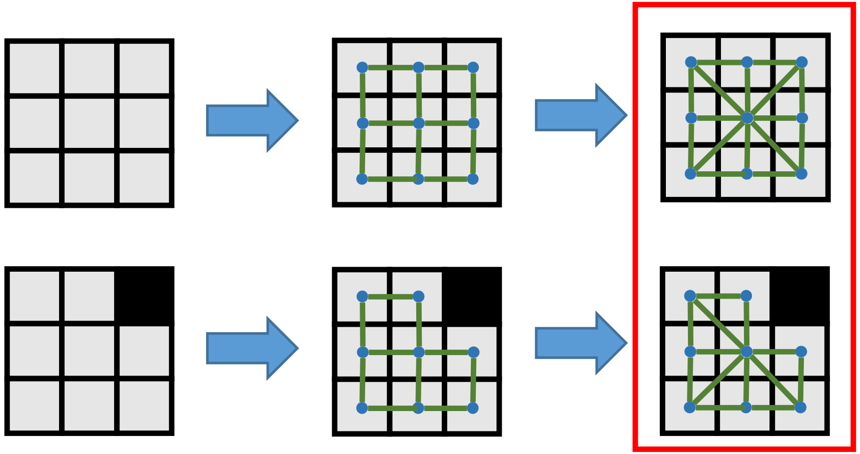
# Connection with previous lectures

- 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




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

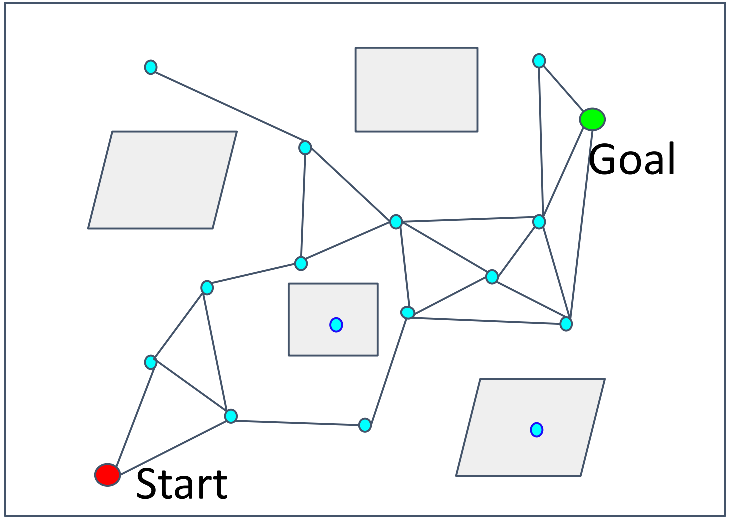
# Connection with previous lectures

- 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



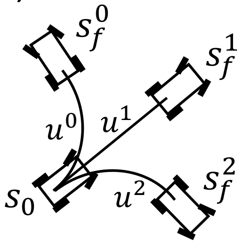

- Actually, this is a discretization of state space!
- Here the state is $R^2$, only position (x, y) is considered here.

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

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

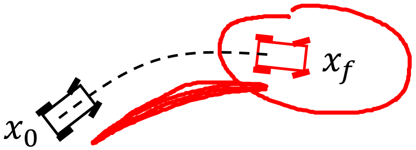
- The robot is differentially driven.
- We have an initial state $s_0$ of the robot.
- Generate feasible local motions by:

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




- Forward simulation
- Fixed $u, T$
- No mission guidance,
- East 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

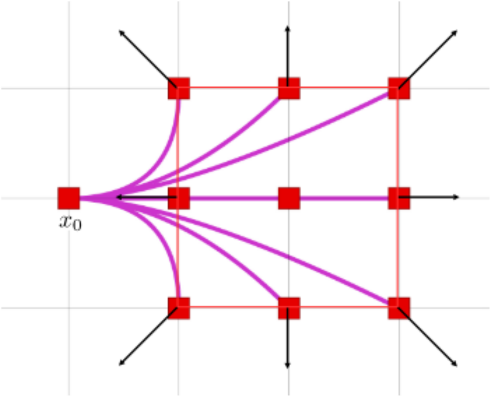
**State**: $s = \begin{pmatrix} x \\ y \\ z \\ \dot{x} \\ \dot{y} \\ \dot{z} \end{pmatrix}$ **Input**: $u = \begin{pmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{pmatrix}$

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

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

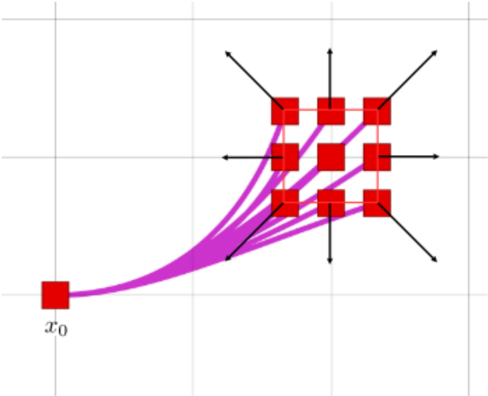
$$\dot{s} = A \cdot s + B \cdot u$$

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




Discretize acceleration

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




Discretize jerk

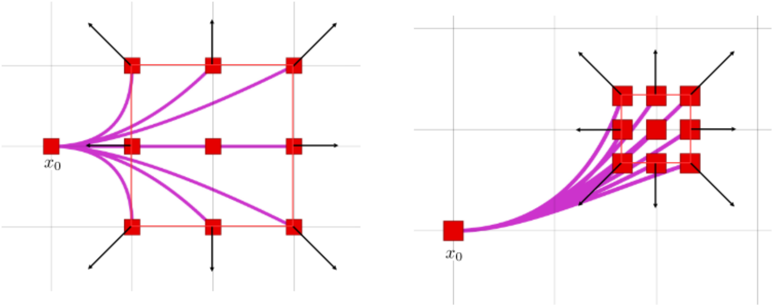
$$v_0 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, a_0 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

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



*Search-based Motion Planning for Quadrotors using Linear Quadratic Minimum Time Control*, Sikang Liu, Nikolay Atanasov, Kartik Mohta, and Vijay Kumar

# Sample in control space




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

**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.

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

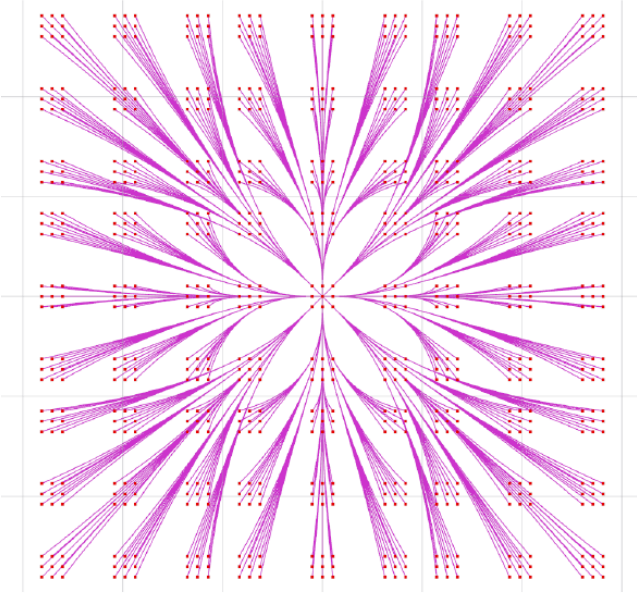
If matrix A $\in R^{n \times n}$ is nilpotent, i.e. $A^n = 0$, $e^{At}$ has a closed-form expression in the form of an $(n-1)$ degree matrix polynomial in $t$.



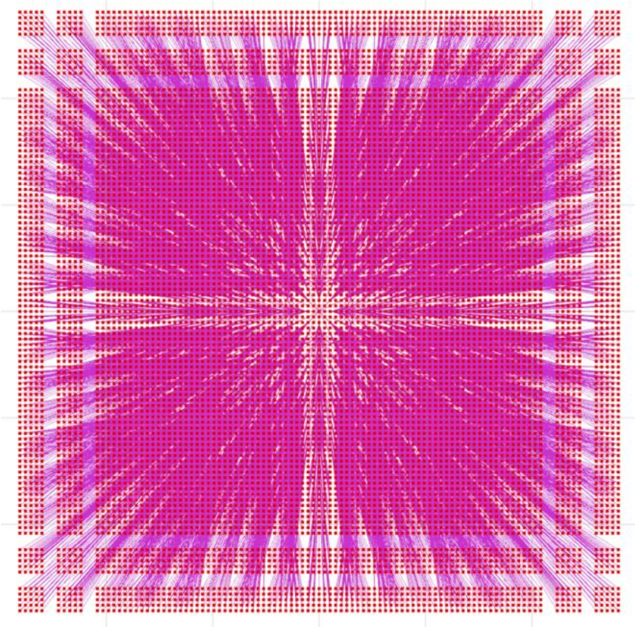
*Search-based Motion Planning for Quadrotors using Linear Quadratic Minimum Time Control*, Sikang Liu, Nikolay Atanasov, Kartik Mohta, and Vijay Kumar

# Sample in control space

The lattice graph obtained by searching



9 discretization



25 discretization

**Note**

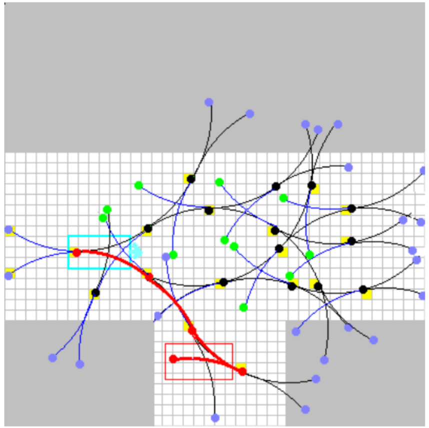
- During searching, the graph can be built when necessary.
- Create nodes (state) and edges (motion primitive) when nodes are newly discovered.
- Save computational time/space.

# Sample in control space


- 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

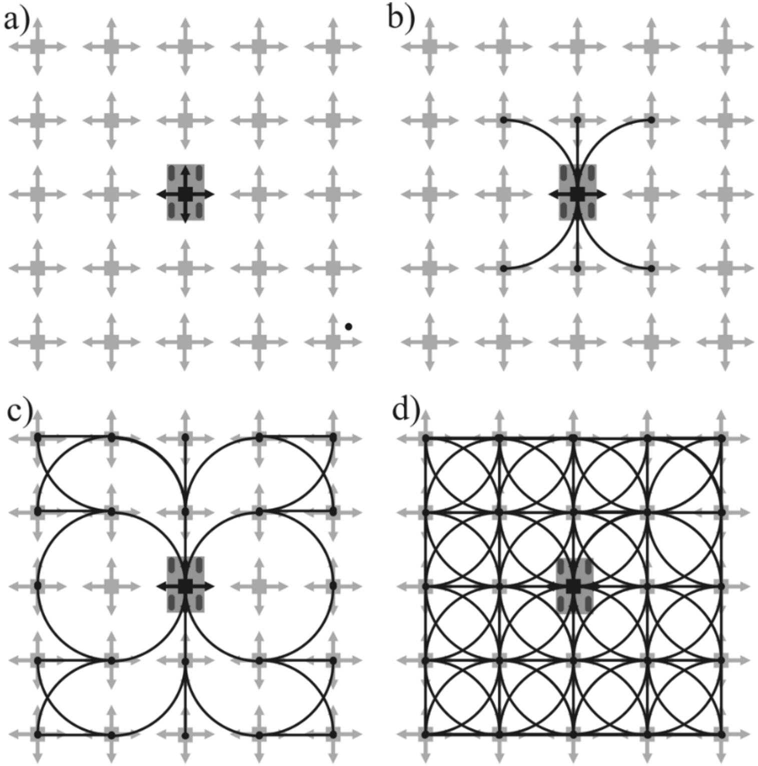
**State**: $s = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix}$ **Input**: $u = \begin{pmatrix} v \\ \emptyset \end{pmatrix}$

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



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

# Sample in state space



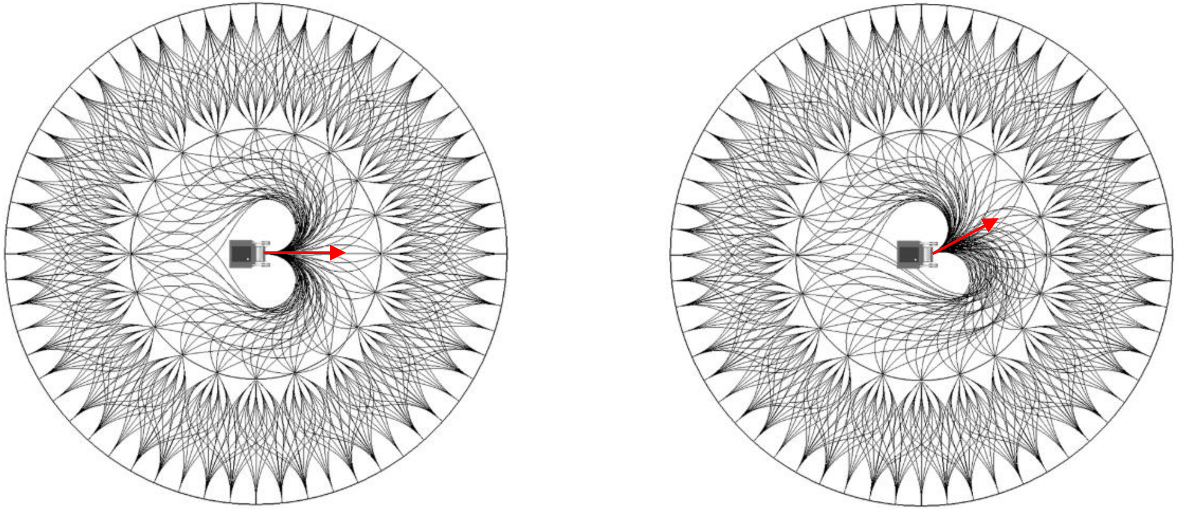

Build a lattice graph:

- 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

*Generating Near Minimal Spanning Control Sets for Constrained Motion Planning in Discrete State Spaces*, Mihail Pivtoraiko and Alonzo Kelly

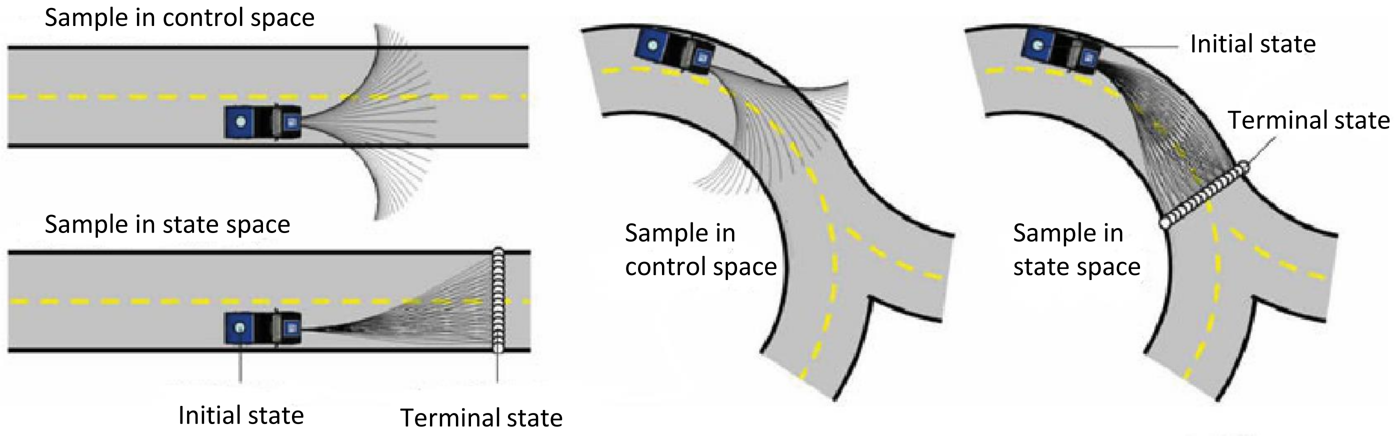
# Sample in state space

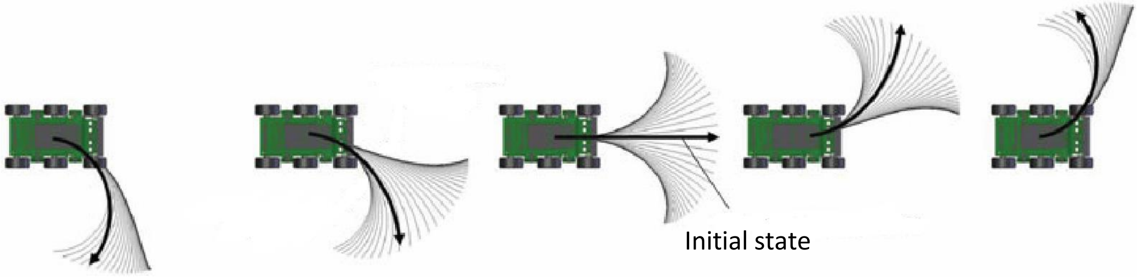


- Two layer lattice graph
- Only first layer is different
- Different initial states

*Optimal Rough Terrain Trajectory Generation for Wheeled Mobile Robots,* Thomas M. Howard Alonzo Kelly

# Comparison




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

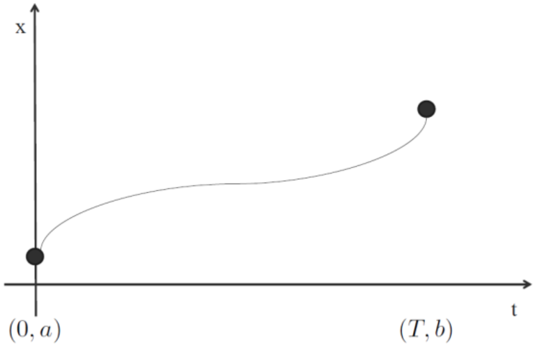
*State Space Sampling of Feasible Motions for High-Performance Mobile Robot Navigation in Complex Environments,*
Thomas M. Howard, Colin J. Green, and Alonzo Kelly

# 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)

- $5^{th}$ 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:

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

## minimum principle

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

$\lambda(t)$ 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:

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

*A Computationally Efficient Motion Primitive for Quadrocopter Trajectory Generation,* Mark W. Mueller, Markus Hehn, and Raffaello D'Andrea

*Dynamic Programming and Optimal Control,* D. P. Bertsekas

# 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:

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

## minimum principle

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

$\lambda(t)$ 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:

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

*A Computationally Efficient Motion Primitive for Quadrocopter Trajectory Generation,* Mark W. Mueller, Markus Hehn, and Raffaello D'Andrea

*Dynamic Programming and Optimal Control,* D. P. Bertsekas

# Pontryagin’s minimum principle

Generally:

$$J = \boxed{h(s(T))} + \boxed{\int_0^T g(s(t),u(t)) \cdot dt}$$

final state

transition cost

Write the Hamiltonian and costate:

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

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

Suppose:

$s^*$: Optimal state

$u^*$: Optimal input

We have

## minimum principle

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

$\lambda(t)$ 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:

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

*A Computationally Efficient Motion Primitive for Quadrocopter Trajectory Generation,* Mark W. Mueller, Markus Hehn, and Raffaello D’Andrea

*Dynamic Programming and Optimal Control,* D. P. Bertsekas

# 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:

$$H(s,u,\lambda) = \frac{1}{T} j^2 + \lambda^T f_s(s,u)$$

$$= \frac{1}{T} j^2 + \lambda_1 v + \lambda_2 a + \lambda_3 j$$

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

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:

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

$$= \frac{1}{2}\alpha t^2 + \beta t + \gamma$$

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


*A Computationally Efficient Motion Primitive for Quadrocopter Trajectory Generation,* Mark W. Mueller, Markus Hehn, and Raffaello D'Andrea

*Dynamic Programming and Optimal Control,* D. P. Bertsekas

# Optimal Boundary Value Problem (OBVP)

The cost:

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

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

How?

Polynomial function root finding problem.

$\alpha, \beta, \gamma$ 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 $J_k = \int_0^T j_k(t)^2 dt + T.$

# 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:

$$h(s(T)) = \begin{cases} 0, & if \quad s = s(T) \\ \infty, & otherwise \end{cases}$$

Not differentiable

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

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

For (partially)-free final state problem:

$$given \quad s_i(T), \; i \in I$$

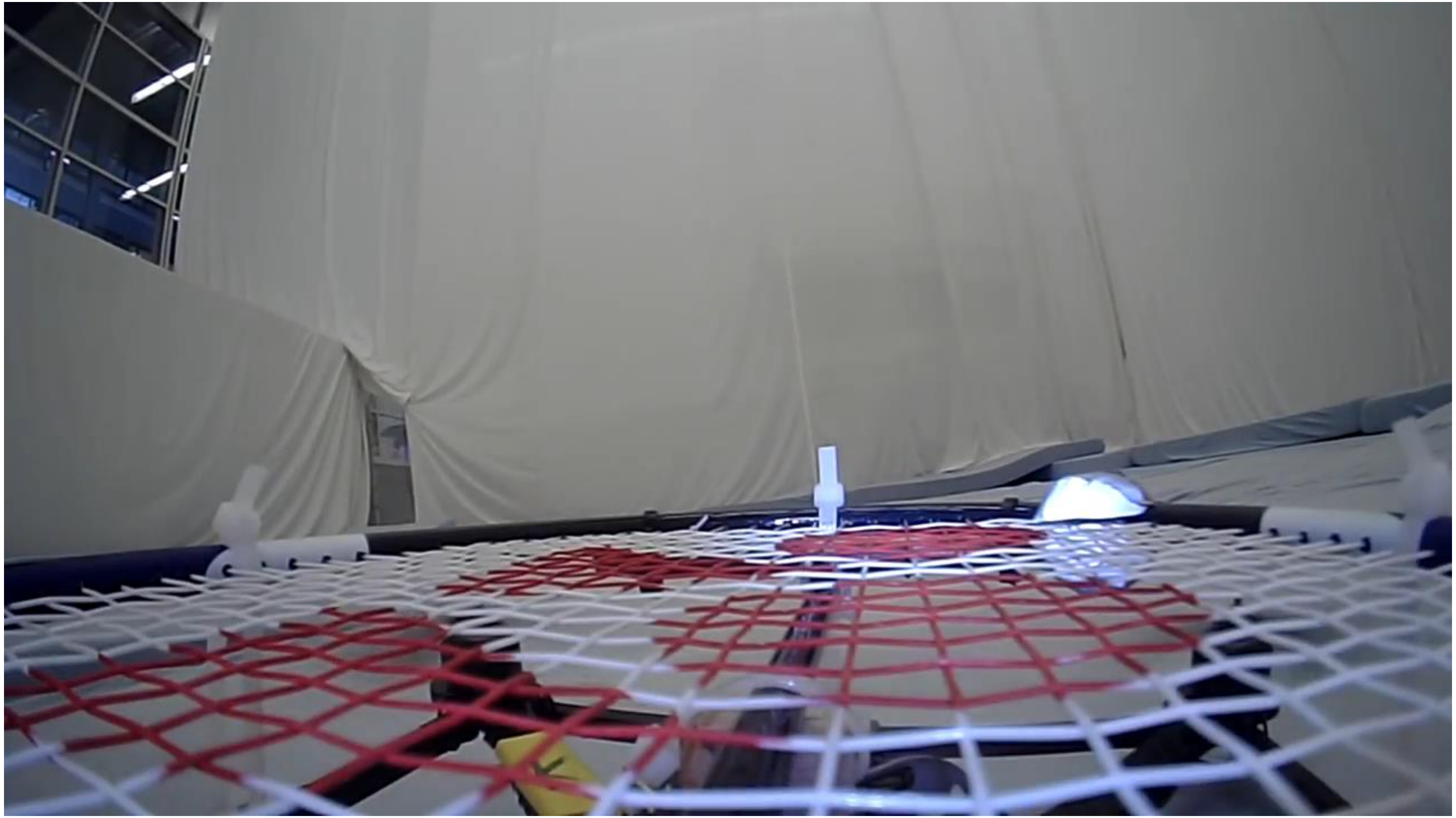
We have boundary condition for other costate:

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

Then we solve this problem again.

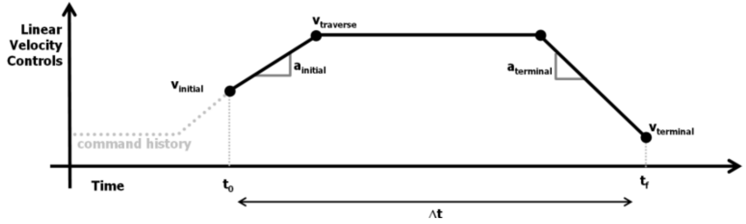
# Application

# Application



# Another example


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

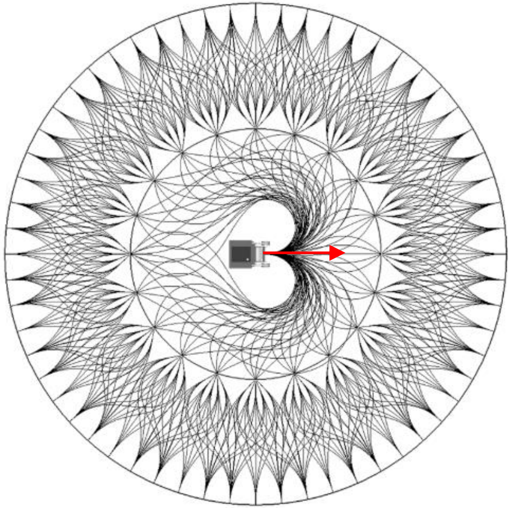


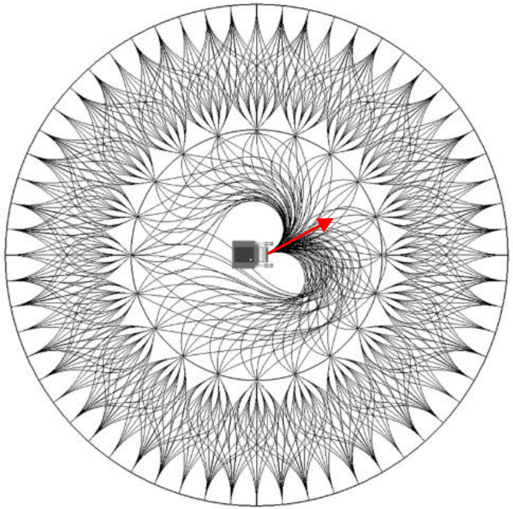

- Constrained Trajectory Generation
- Numerical difference evaluated Jacobian

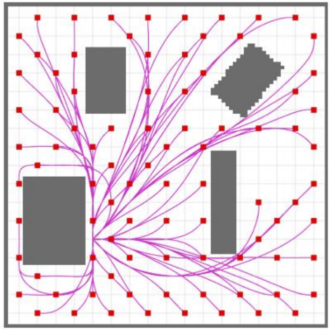
- Offline BVP, trajectory generation.
- Online search.

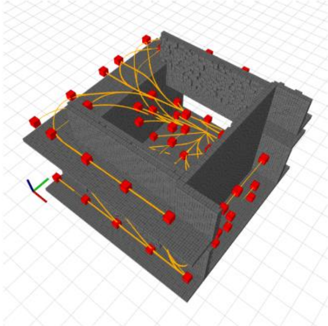
*Optimal Rough Terrain Trajectory Generation for Wheeled Mobile Robots,* Thomas M. Howard Alonzo Kelly

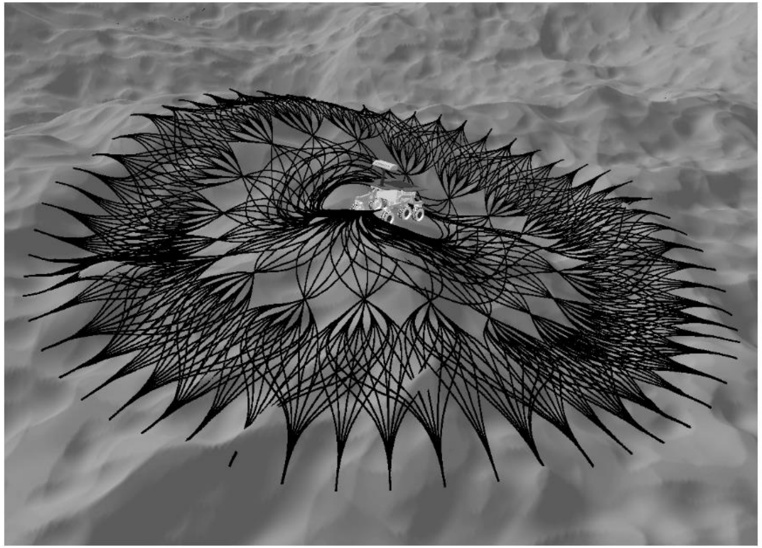
# Graph search problem









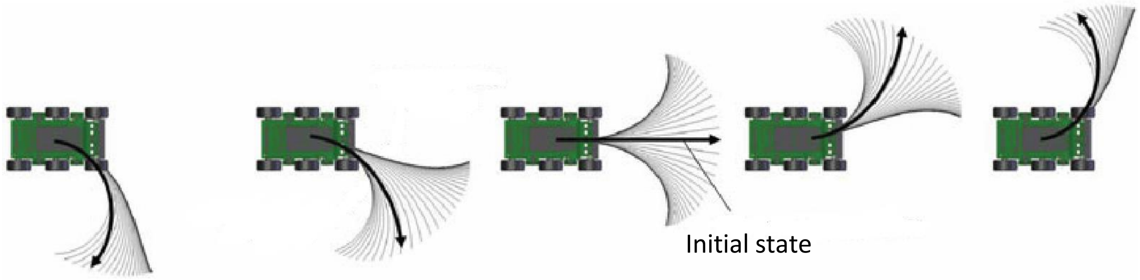


- Up to now, it ahs become a graph search problem.
- Every techniques learned in L2 can be applied here.

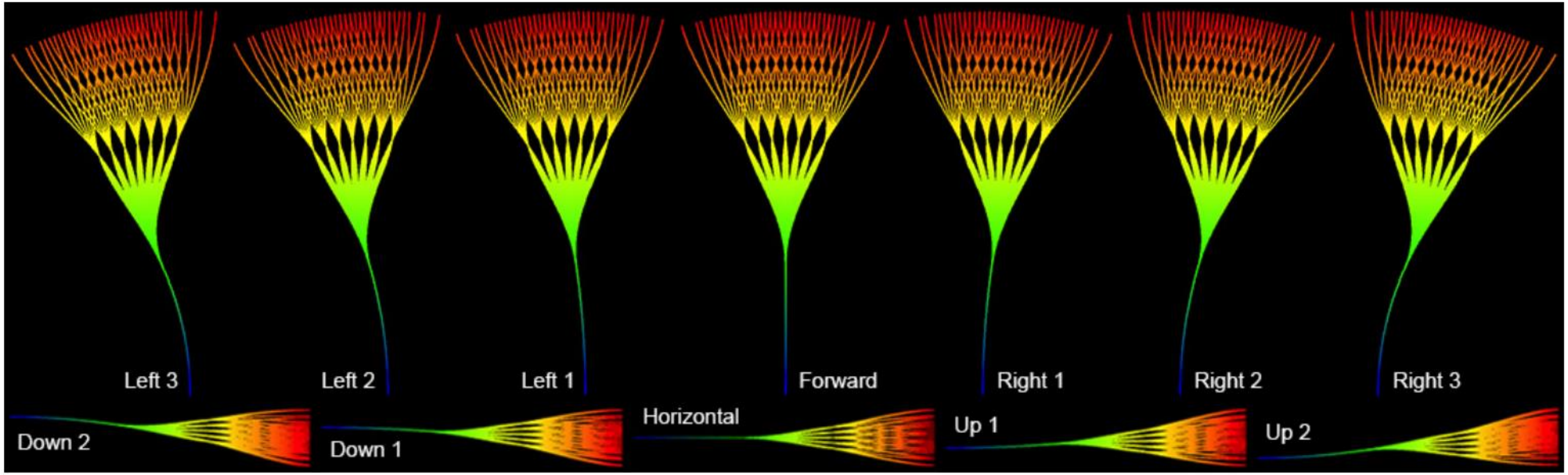
# Trajectory library

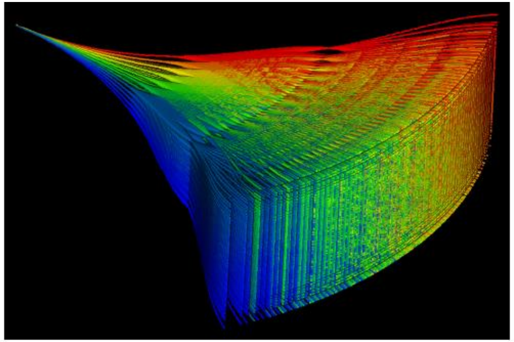
- Single layer lattice planning is a common option for local collision avoidance.
- No graph search, only trajectory selection.
- Rating each trajectory based on a multi-term cost function.

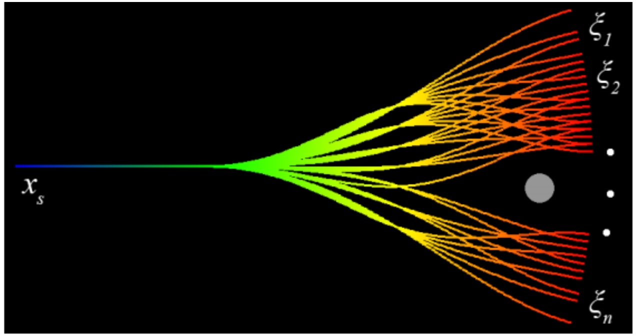
collision risk, information acquisition, comfort, energy, …



# Trajectory library





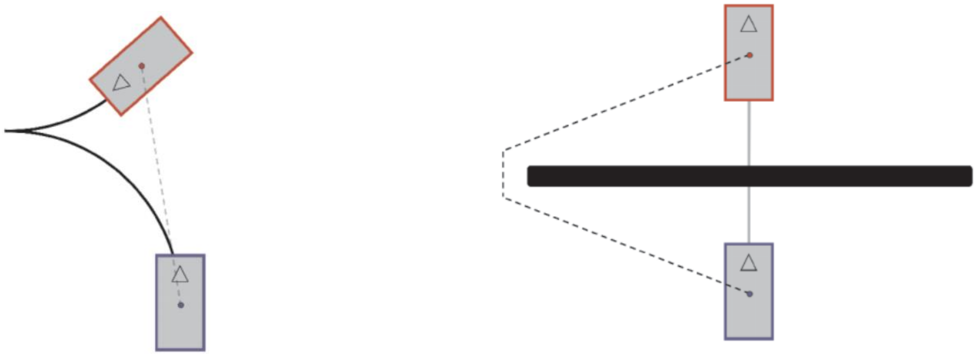

*Maximum Likelihood Path Planning for Fast Aerial Maneuvers and Collision Avoidance* Ji Zhang, Chen Hu, Rushat Gupta Chadha, and Sanjiv Singh

# Heuristic

# Heuristic design

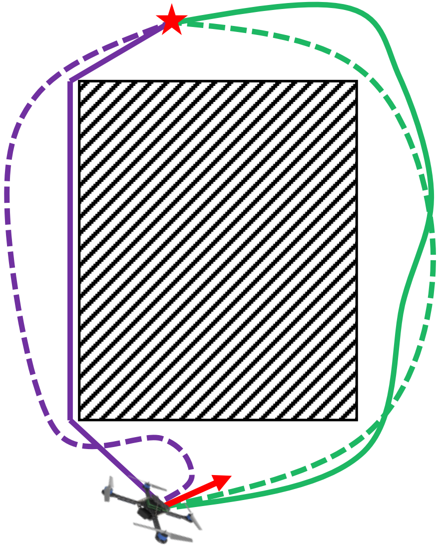
Principle: solve an easier problem

- 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)=infinite for all other nodes in the graph
- Loop
    - If the queue is empty, return FALSE; break;
    - Remove the node "n" with the lowest $f(n)=\boxed{g(n)}+\boxed{h(n)}$ 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) = infinite
            - g(m)= g(n) + Cnm
            - Push node "m" into the queue
        - If $g(m) > g(n) + C_{nm}$
            - g(m)= g(n) + Cnm
    - end
- End Loop

Accumulate cost

# Comparison

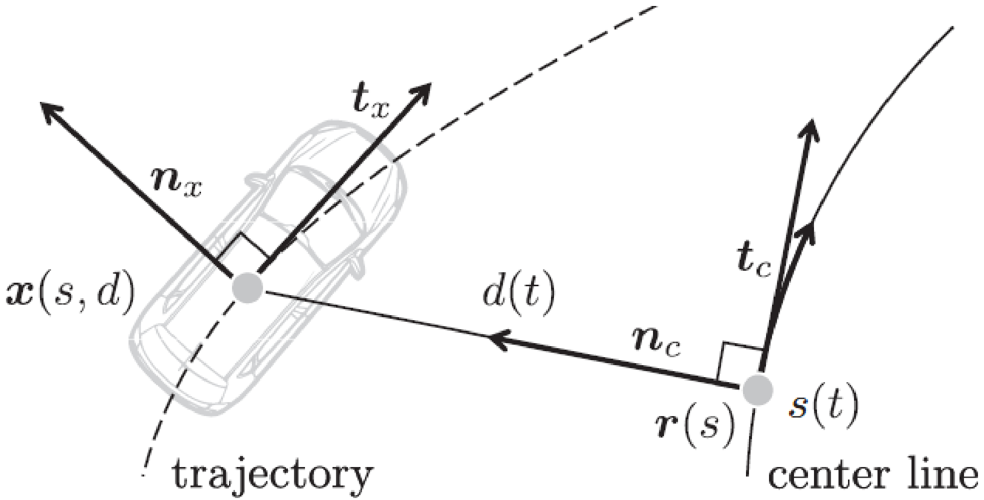
Euclidean 2D distance

non-holonomic-without-obstacles

*Path Planning for Autonomous Vehicles in Unknown Semi-structured Environments,* Dmitri Dolgov, Sebastian Thrun, Michael Montemerlo, James Diebel

# 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.

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

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

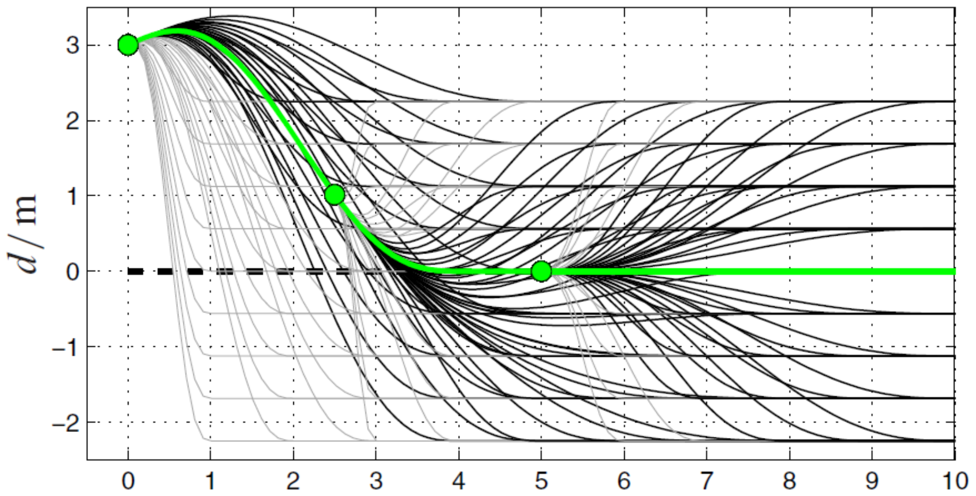
✓ Solve the optimal control problem.

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

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

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:

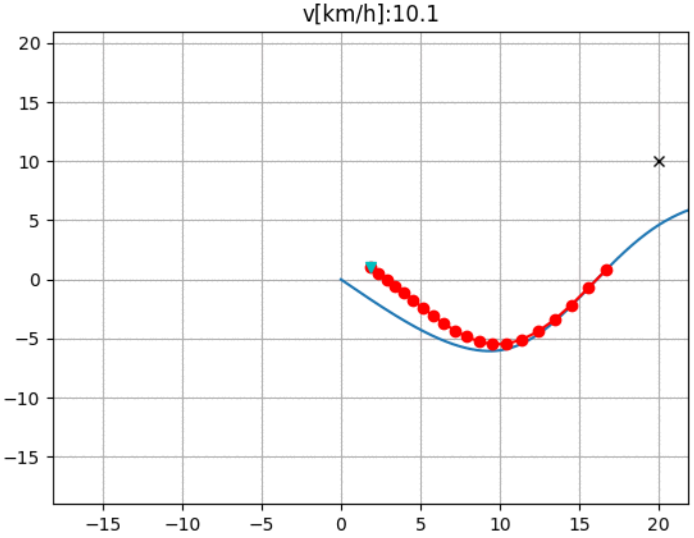
$$D(T) = \begin{pmatrix} d_f & 0 & 0 \end{pmatrix}$$

Recall what we learn previously:

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

# Planning in Frenet-serret frame

Example



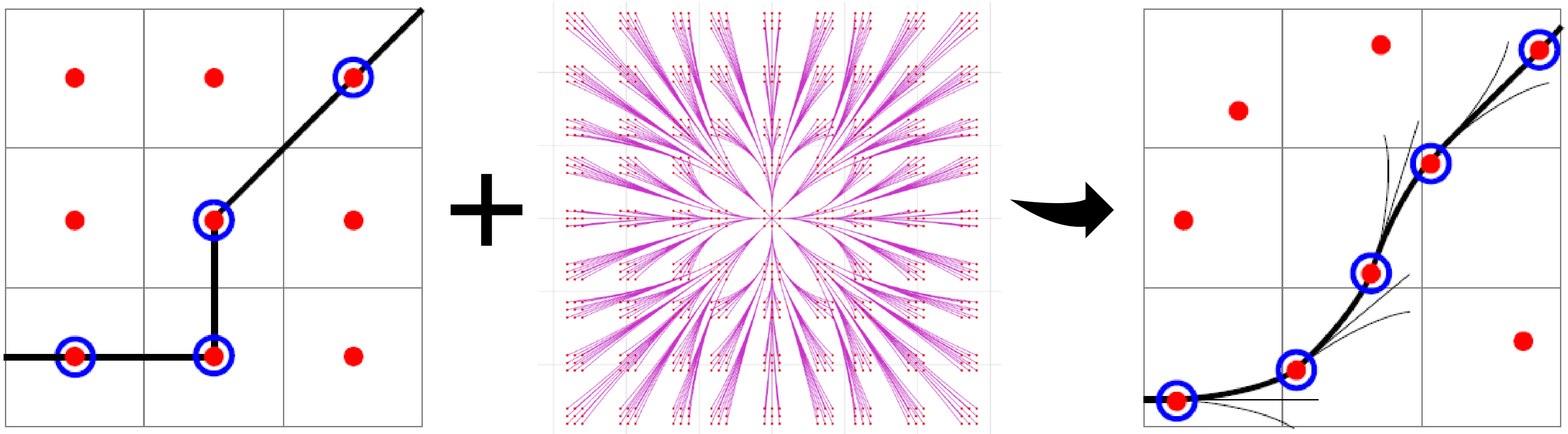

https://pythonrobotics.readthedocs.io/en/latest/modules/path_planning.html#optimal-trajectory-in-a-frenet-frame

# 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.



*Practical Search Techniques in Path Planning for Autonomous Driving*, Dmitri Dolgov, Sebastian Thrun
Path Planning for Autonomous Vehicles in Unknown Semi-structured Environments, Dmitri Dolgov, Sebastian Thrun

# 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)=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) = 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

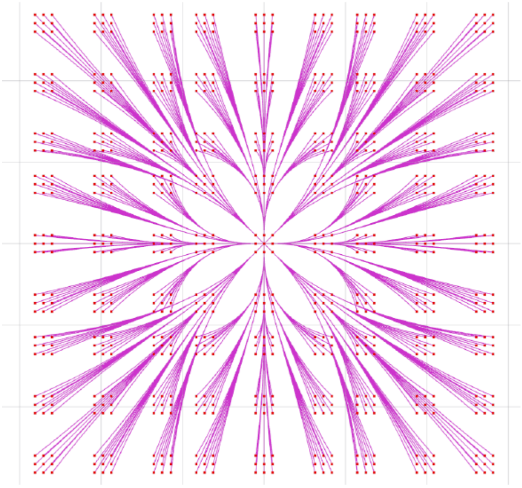
(b) non-holonomic-without-obstacles

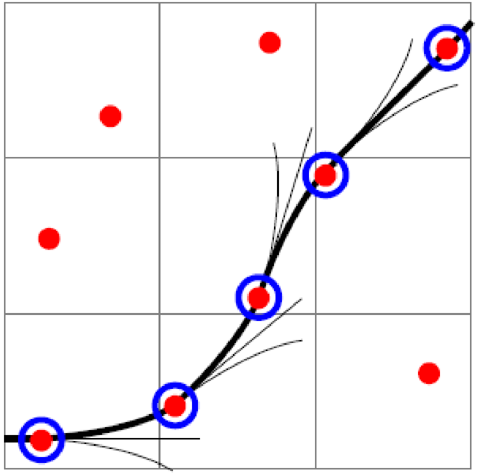
(c) non-holonomic-without-obstacles, bad performance in dead ends

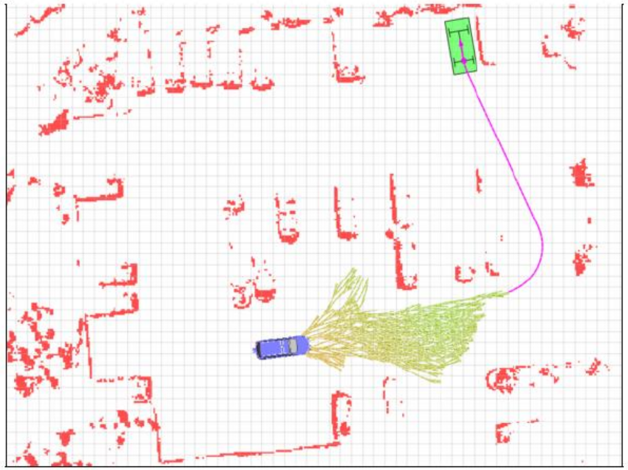
(d) non-holonomic-without-obstacles + holonomic-with-obstacles (2D shortest path)

# Other tricks

**Analytic Expansions**: One shot heuristic
Add a state-driven bias towards the searching process





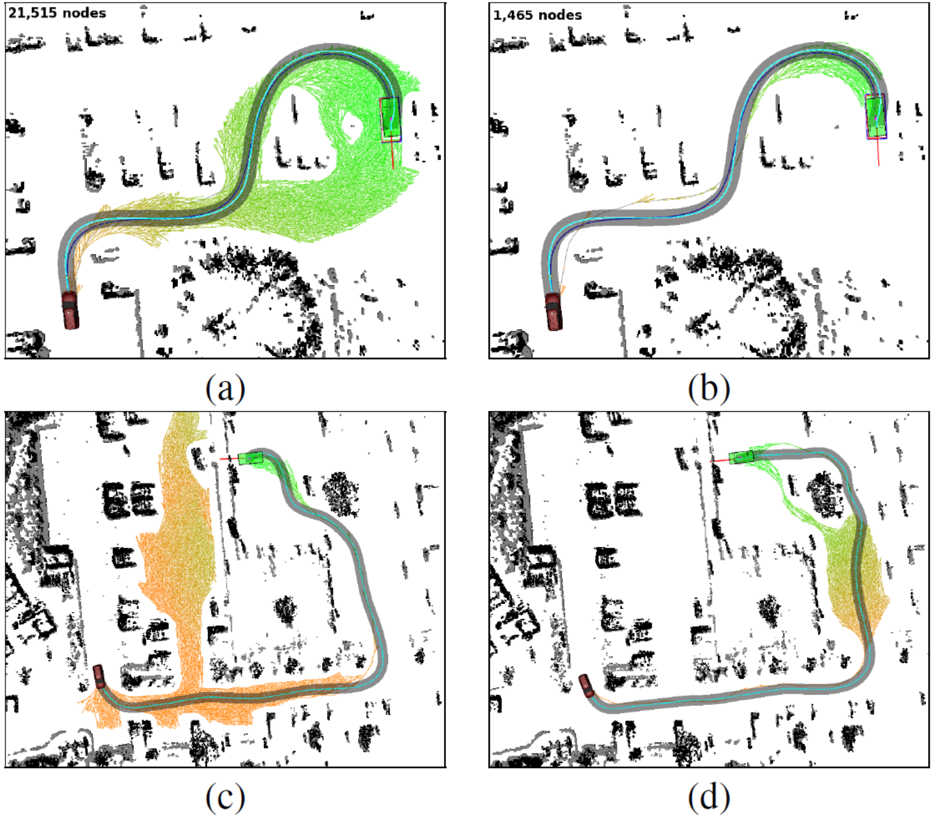


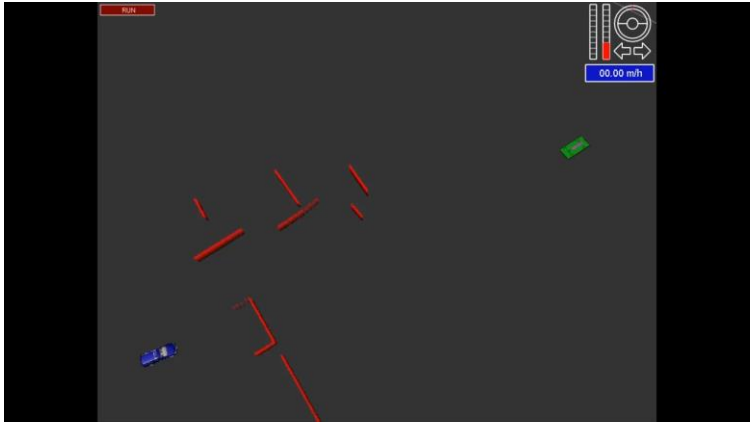
Control space sample (discretization) is kind of low-efficient, since no target biasing is encoded

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

# Application

# Autonomous car

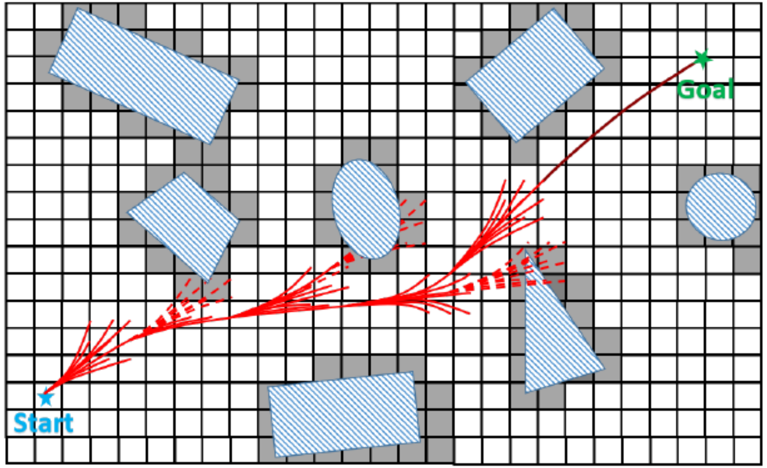


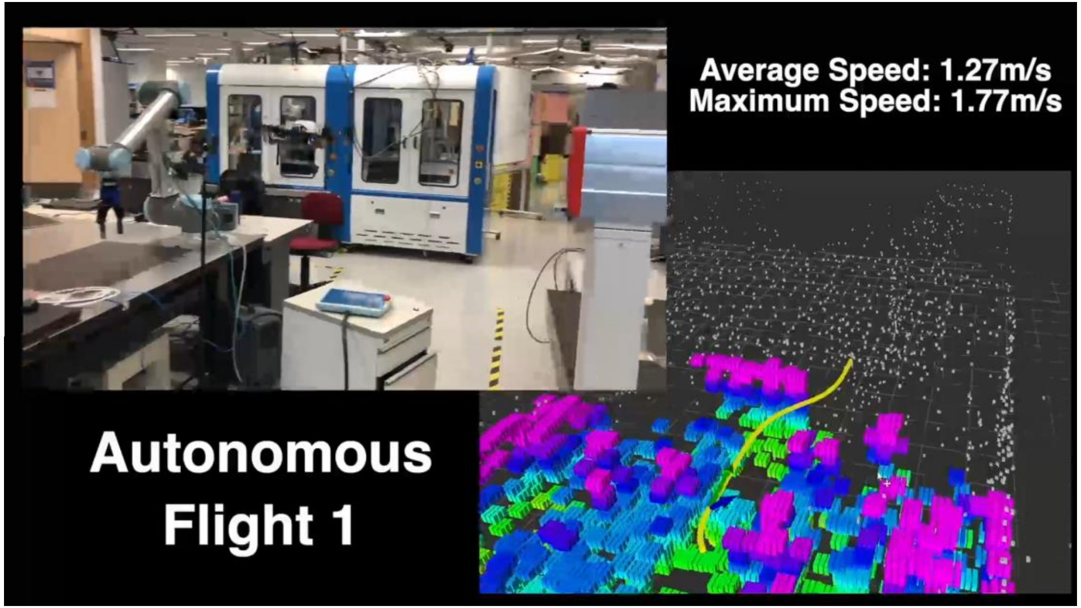





*Practical Search Techniques in Path Planning for Autonomous Driving*, Dmitri Dolgov, Sebastian Thrun

# Autonomous UAV



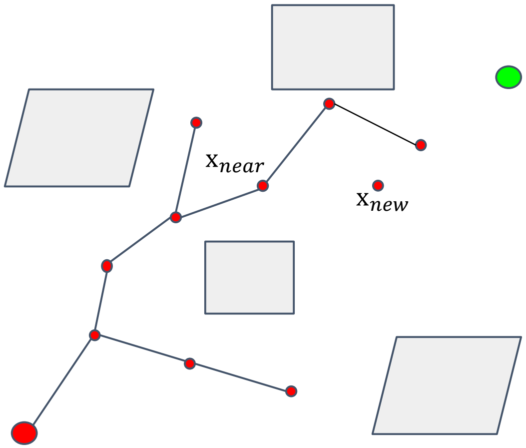

- As a promising front-end
- Careful engineering considerations
- Linear UAV model: nilpotent!
- Sophisticated C++ implementation

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

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

# Kinodynamic RRT*

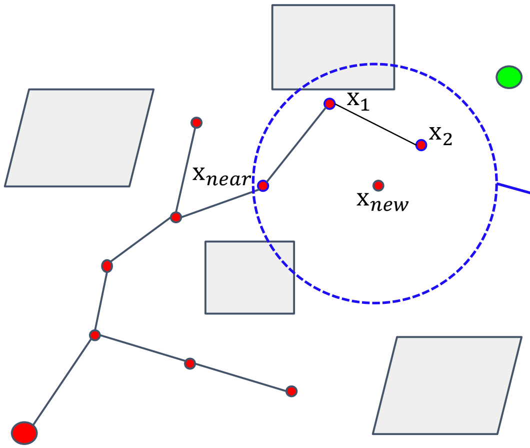
# Review of RRT*




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

# Review of RRT*




**Algorithm 2:** RRT Algorithm

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

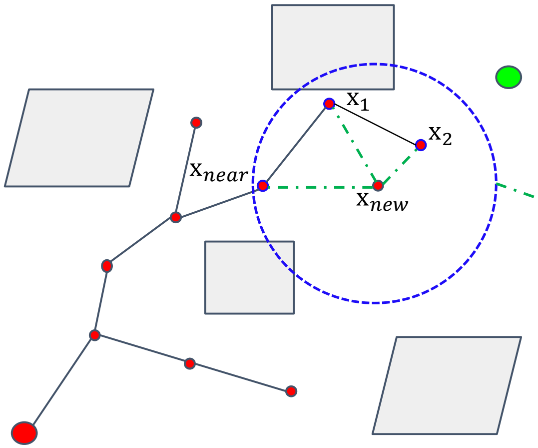
**Result:** A path $\Gamma$ from $x_{init}$ to $x_{goal}$

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

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

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

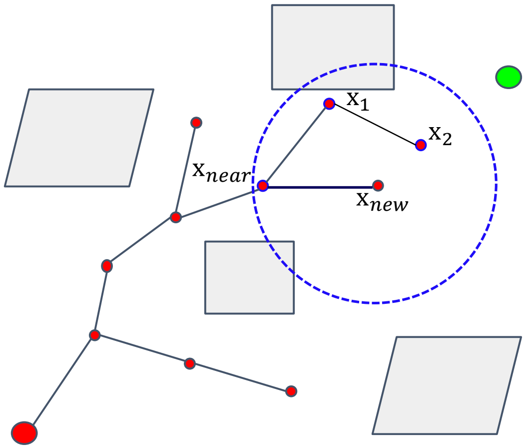
# Review of RRT*




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

# Review of RRT*




---

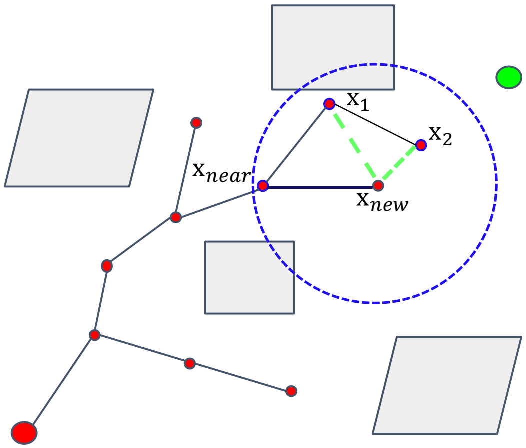
**Algorithm 2:** RRT Algorithm

---

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

---

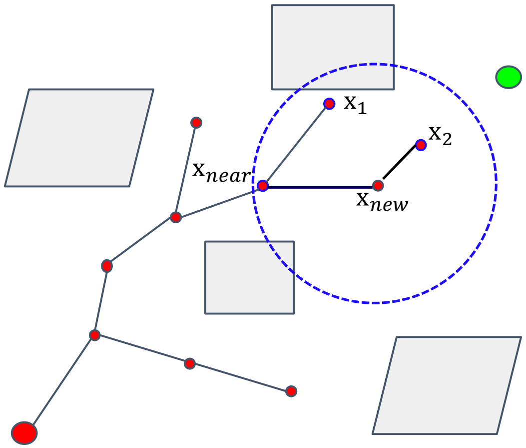
# Review of RRT*




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

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

# 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**
        $X_{near} \leftarrow NearC(\mathcal{T}, x_{new})$;
        $x_{min} \leftarrow ChooseParent(X_{near}, x_{near}, x_{new})$ ;
        $\mathcal{T}.addNodEdge(x_{min}, x_{new})$;
        $\mathcal{T}.rewire()$;

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 ← Sample(E);
    X_near ← Near(T, x_rand);
    x_min ← 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<br>**Output:** A trajectory T from x_init to x_goal<br>T.init();<br>**for** i = 1 to n **do**<br>    x_rand ← Sample(E);<br>    X_near ← Near(T, x_rand);<br>    x_min ← ChooseParent(X_near, x_rand);<br>    T.addNode(x_rand);<br>    T.rewire(); |

LTI system state-space equation:

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

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

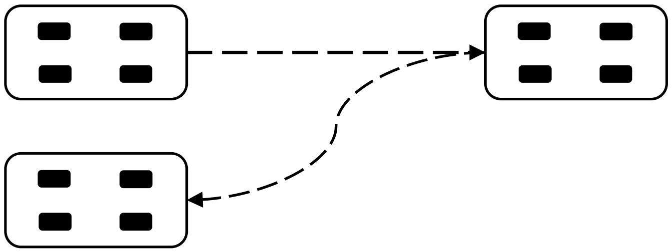
## 2. How to define "Near"

Kinodynamic RRT*

**Input:** E, x_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();
```



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.

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

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"

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

If we know the arriving time **τ** 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 x1, fixed final time τ

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:

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

F. Lewis, V. Syrmos. Optimal Control. John Wiley & Sons

# Unbounded Optimal control solutions

## 2.1 Fixed final state x1, fixed final time τ

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

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:

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

F. Lewis, V. Syrmos. Optimal Control. John Wiley & Sons

# Unbounded Optimal control solutions

## 2.2 Fixed final state x1, free final time τ

If we want to find the optimal arrival time τ, 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 τ is found by taking the derivative of $c[\tau]$ with respect to τ:

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

Where

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

# Unbounded Optimal control solutions

## 2.2 Fixed final state x1, free final time τ

Solve $\dot{c}[\tau] = 0$ for $\tau^*$.
Noted that the function $\mathrm{c}[\tau]$ may have multiple local minima.
And for a double integrator system, it's a $4^{\text{th}}$ 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_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);
    T.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 ← Sample(E);
    X_near ← Near(T, x_rand);
    x_min ← 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 ← Sample(E);
    X_near ← Near(T, x_rand);
    x_min ← 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 $\boldsymbol{x_0}$ to state $\boldsymbol{x_1}$ changes with arrival time $\boldsymbol{\tau}$.
We can see that given initial state $\boldsymbol{x_0}$, cost tolerance $\boldsymbol{r}$ and arrival time $\boldsymbol{\tau}$, the forward-reachable set of $\boldsymbol{x_0}$ is:

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

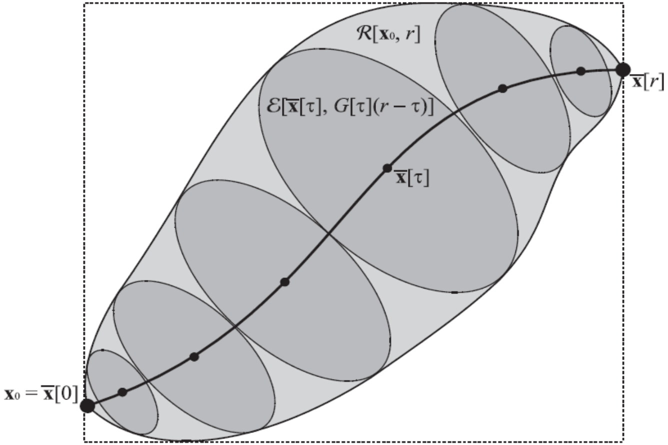
# Problems when it comes to motion constraints

## 4. How to find near nodes efficiently

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

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

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



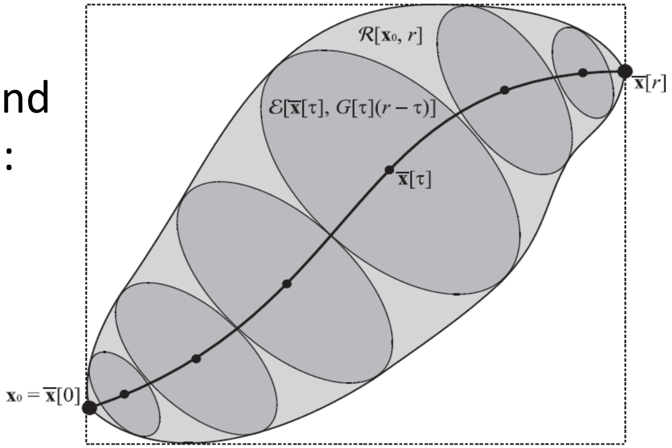

Hence, the forward-reachable set is the union of high dimensional ellipsoids for all possible arrival times $\tau$.

# 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:

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




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_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);
    T.rewire();
```

When do "Near" query and "ChooseParent", $X\_near$ can be found from the **backward-reachable set** of $x\_rand$.

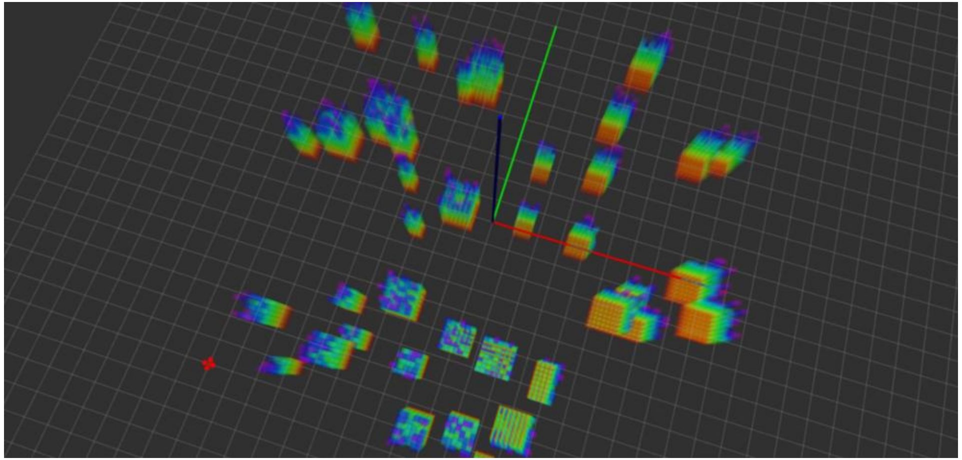
# Problems when it comes to motion constraints

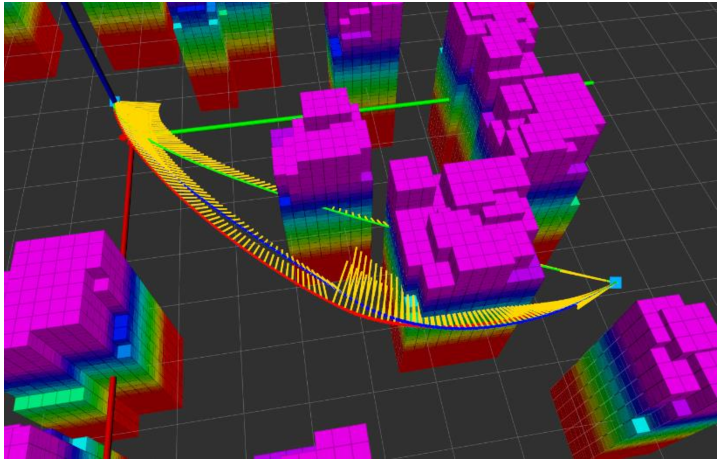
## 5. How to “Rewire”

| Kinodynamic RRT* |
|---|
| **Input:** E, x_init, x_goal<br><br>**Output:** A trajectory T from x_init to x_goal<br><br>T.init();<br>**for** i = 1 to n **do**<br>    x_rand ← Sample(E);<br>    X_near ← Near(T, x_rand);<br>    x_min ← ChooseParent(X_near, x_rand);<br>    T.addNode(x);<br>    T.rewire(); |

When “Rewire”, we calculate the **forward-reachable set** of $x\_rand$, 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!