MEN 491

# Lecture 2: Planning and Control

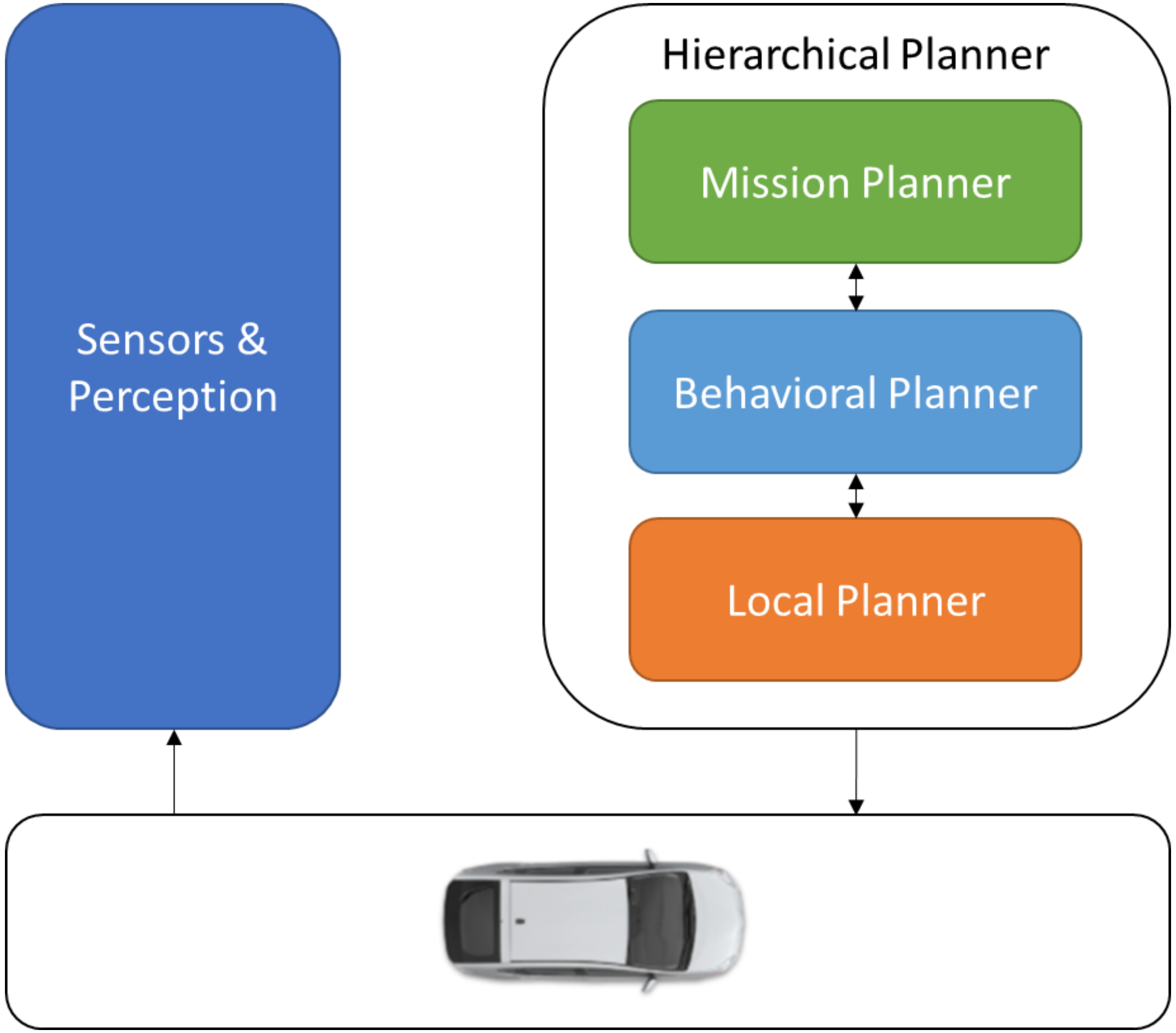

**Mechanical Engineering**

**Cheolhyeon Kwon**

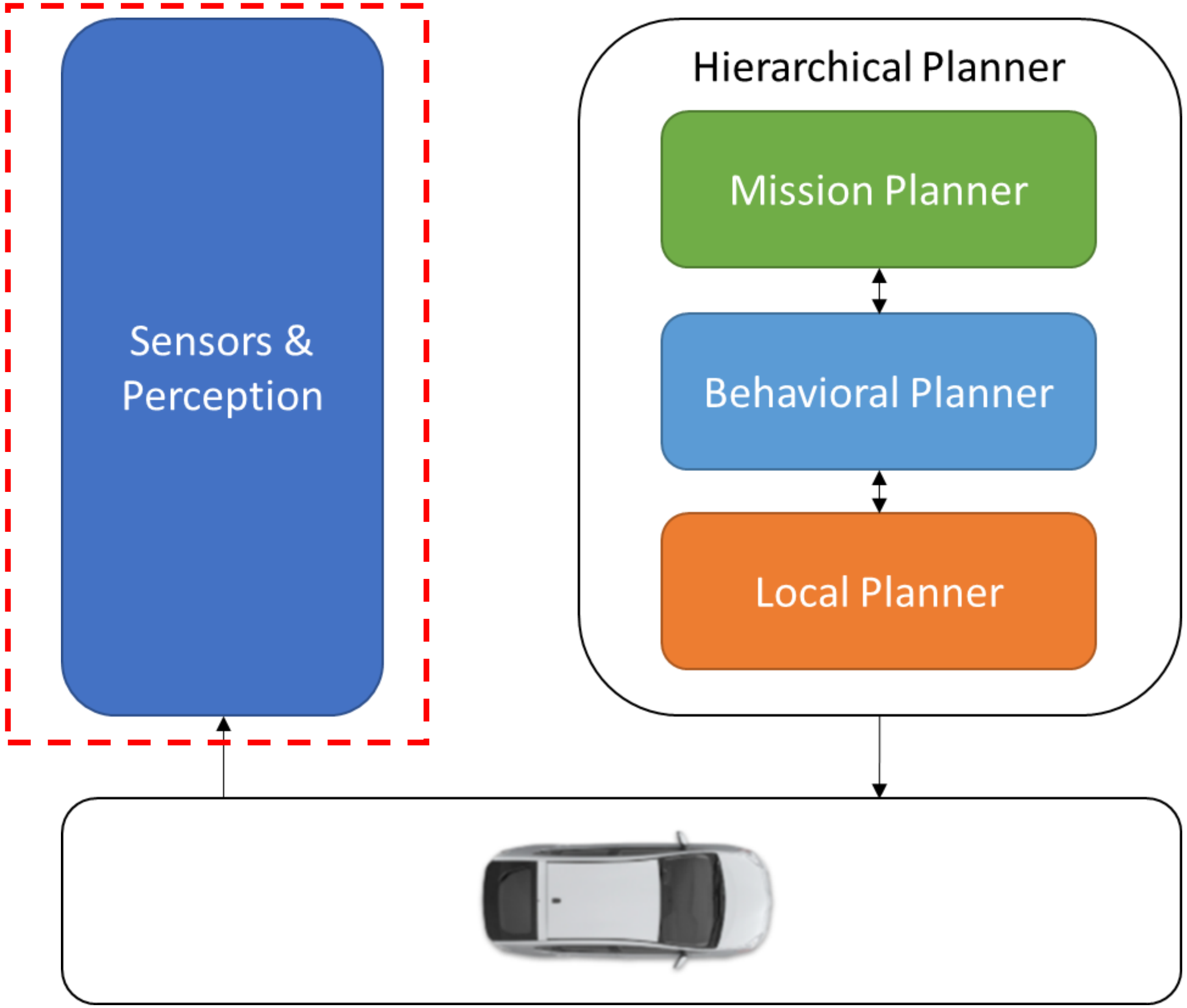

2024.09.26


ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

# Autonomous Vehicles Planning and Control Stack

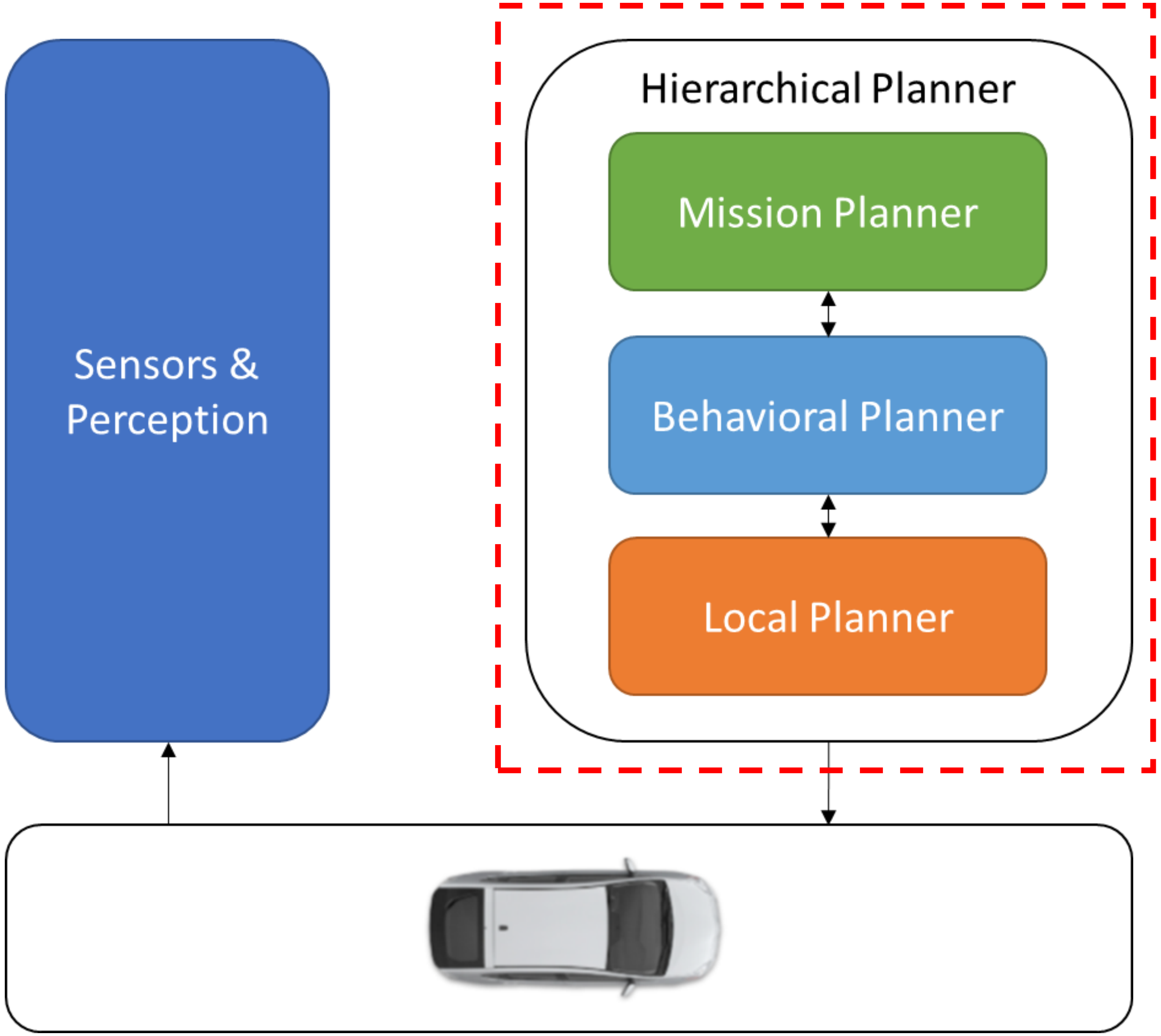


# Autonomous Vehicles Planning and Control Stack



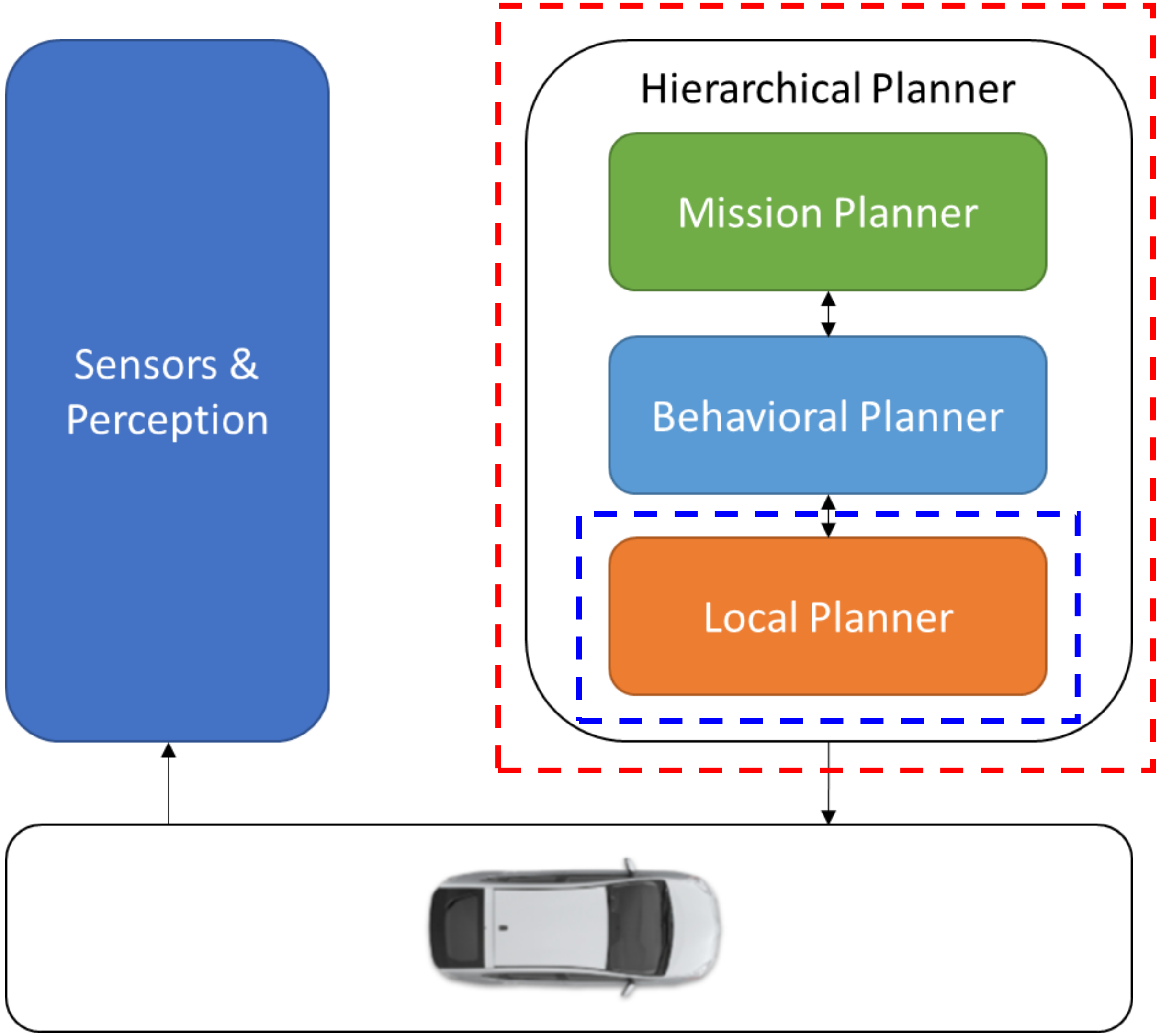
# Autonomous Vehicles Planning and Control Stack

- Mission Planner : what is the overall goal of the vehicle?
- Behavioral Planner : what rules should the vehicle follow in different situations?
- Local Planner : what is the optimal trajectory from position to a goal?



# Autonomous Vehicles Planning and Control Stack

- Mission Planner : what is the overall goal of the vehicle?
- Behavioral Planner : what rules should the vehicle follow in different situations?
- **Local Planner : what is the optimal trajectory from position to a goal?**

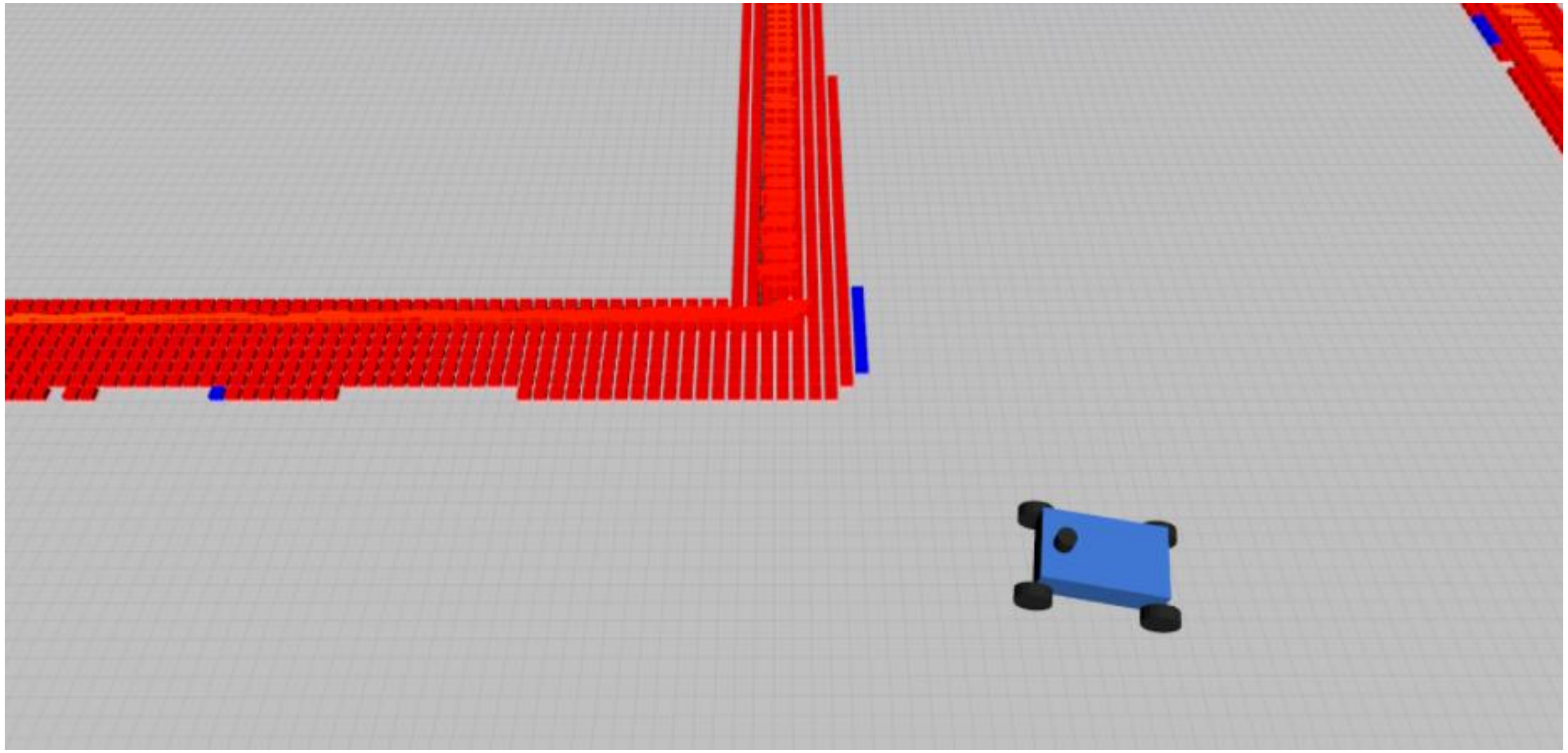


# The Local Planning Problem

- **Given**
  - A robot with configuration space: C
  - The set of obstacles: $C_{obs}$
  - An initial configuration: $q_{init}$
  - A goal configuration: $q_{goal}$
  - (possibly a cost function)
- **Find a path x:[0, 1] → C** (continuous function) such that the path
  - starts from the initial configuration x(0)= $q_{init}$
  - reaches the goal configuration x(1) = $q_{goal}$
  - avoids collision with obstacles x(s) $\notin C_{obs}$ for all s∈ $[0,1]$
  - (possibly is the minimum cost path)
- **We'll go over multiple ways to construct different representations: Grids, Graphs, and Trees and to search the path: Dijkstra, A*, and RRT**

# Discrete Approximations of Continuous Space

- How do we discretize free space into a grid?



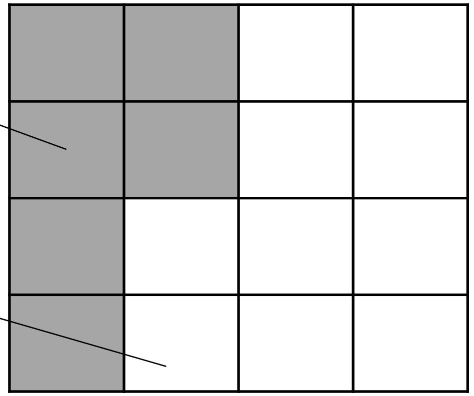
# Discrete Approximations of Continuous Space

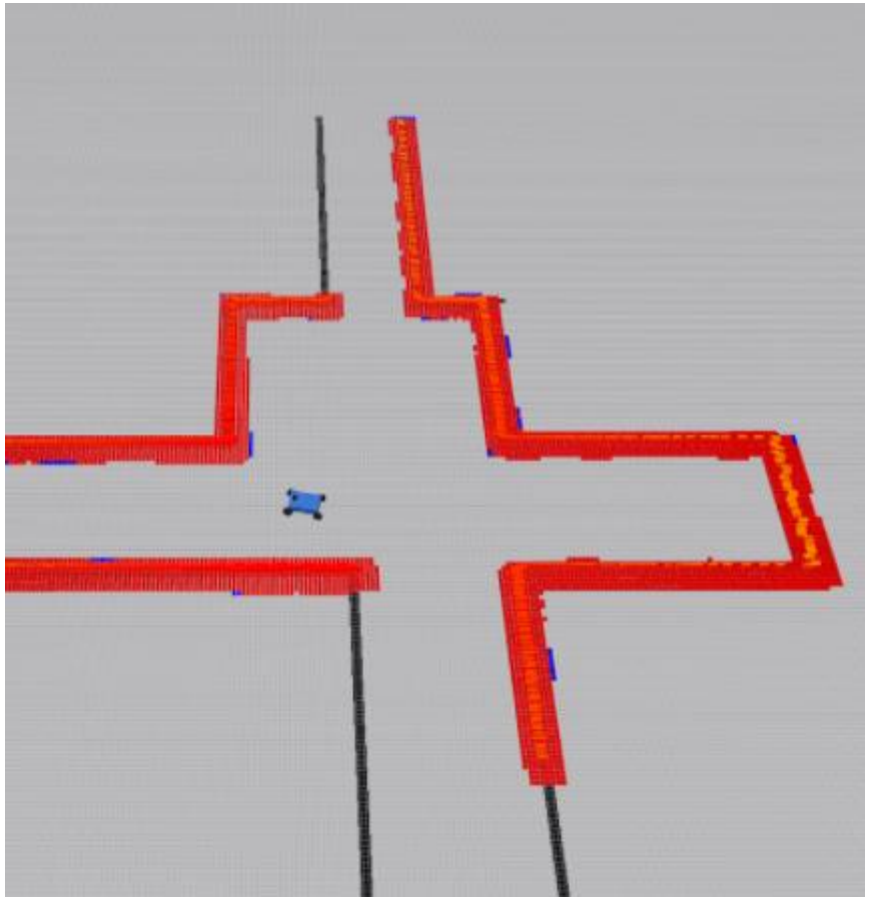
- How do we discretize free space into a grid?
  → Use an **Occupancy Grid**

- A grid cell represents **free space** if:
  - It doesn't overlap with any obstacle (value in grid cell is **binary**)

**occupied space**
$P(m_i) \to 1$

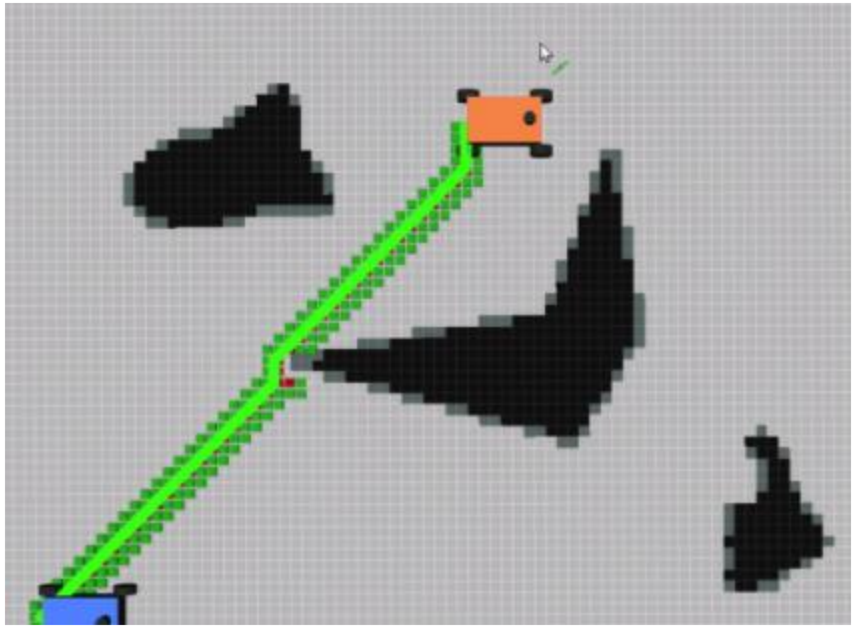
**free space**
$P(m_i) \to 0$





# Planning in Discrete State Space

- Input:
  - **State space**: discrete map with cells labelled as occupied or free
  - **Action space**: set of cells the robot can reach from a certain cell
  - **Cost function**: cost of each action
  - **Start state**
  - **Goal state**

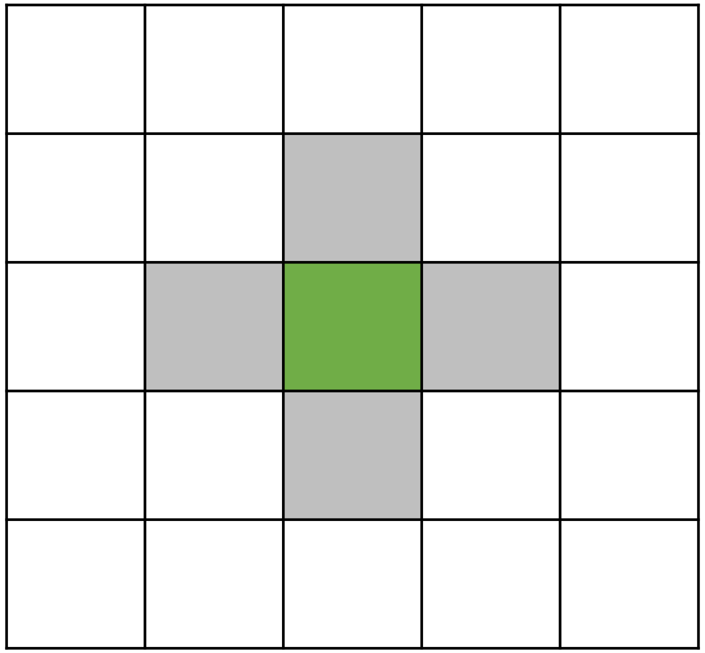


- Output
  - Series of cells in minimum cost path

**→ Now we have a discrete representation of the world, how do we search?**

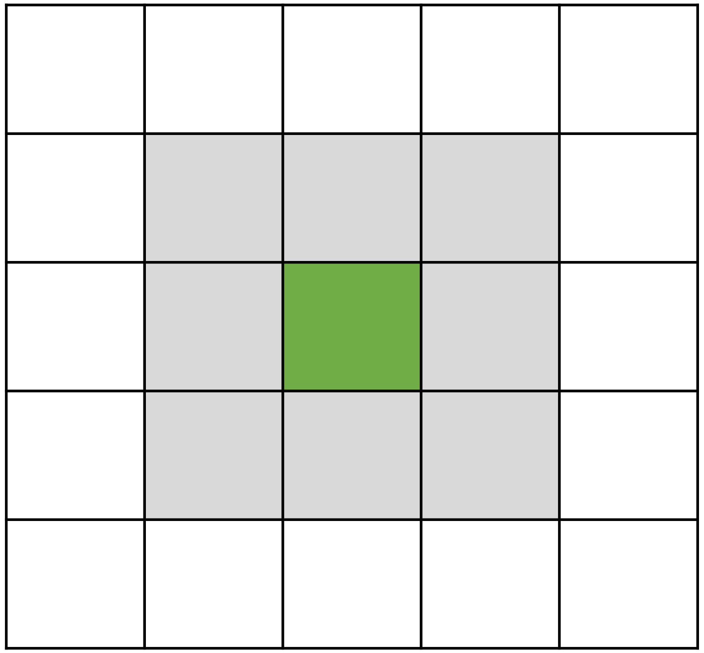
# Occupancy Grid as Graph

**4-Connected**



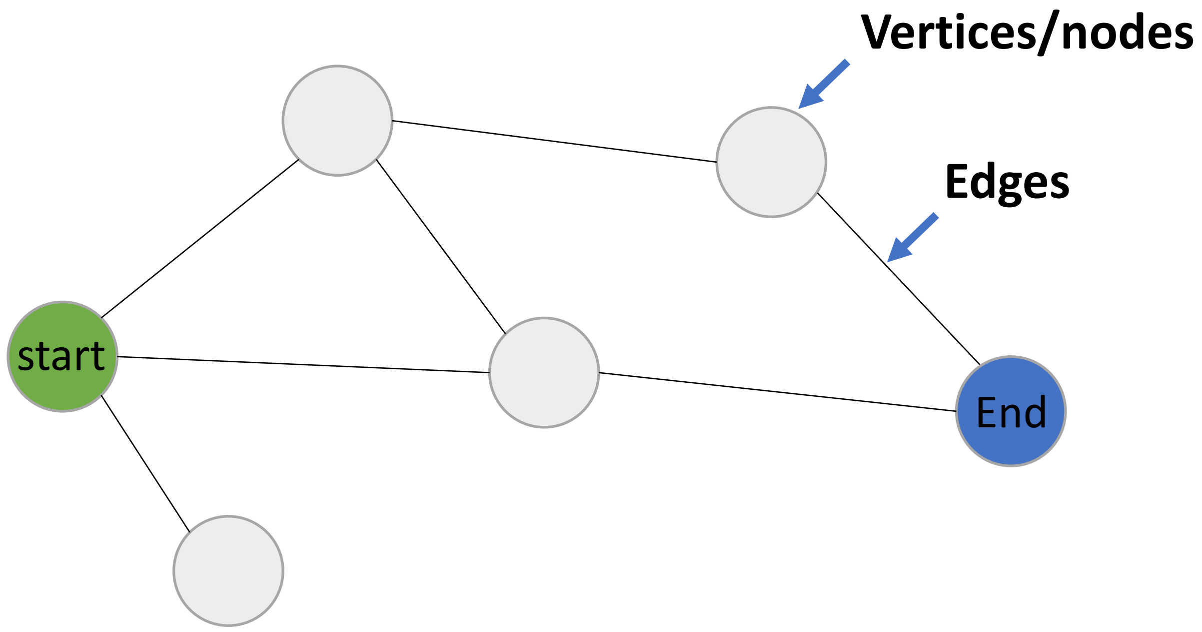
Green - Current Node
Gray - Neighbor Node

**8-Connected**



# Graphs

- Graph: ordered pair G = (V,E) where V is a set of vertices/nodes and E is a set of edges
- Edges: 2-tuple of vertices
  - Directed vs Undirected
  - Weighted vs Unweighted

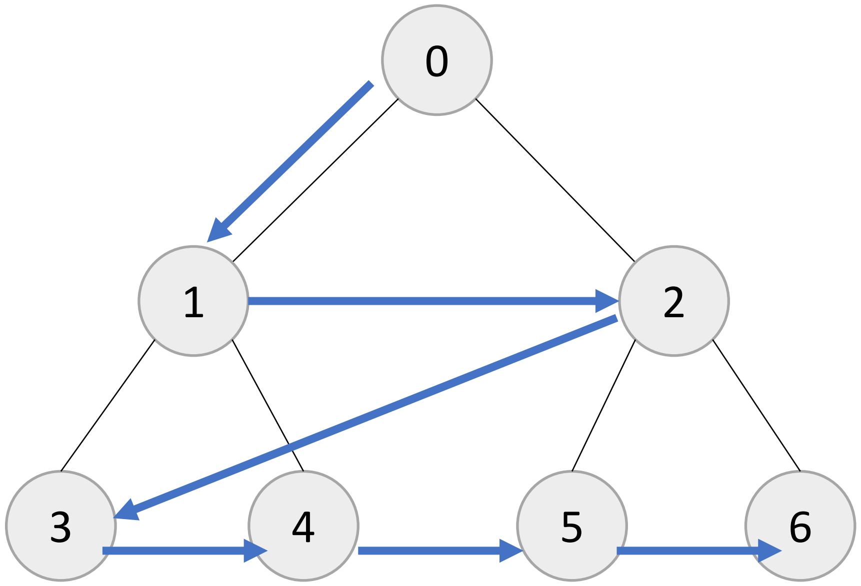


# Best-First Search

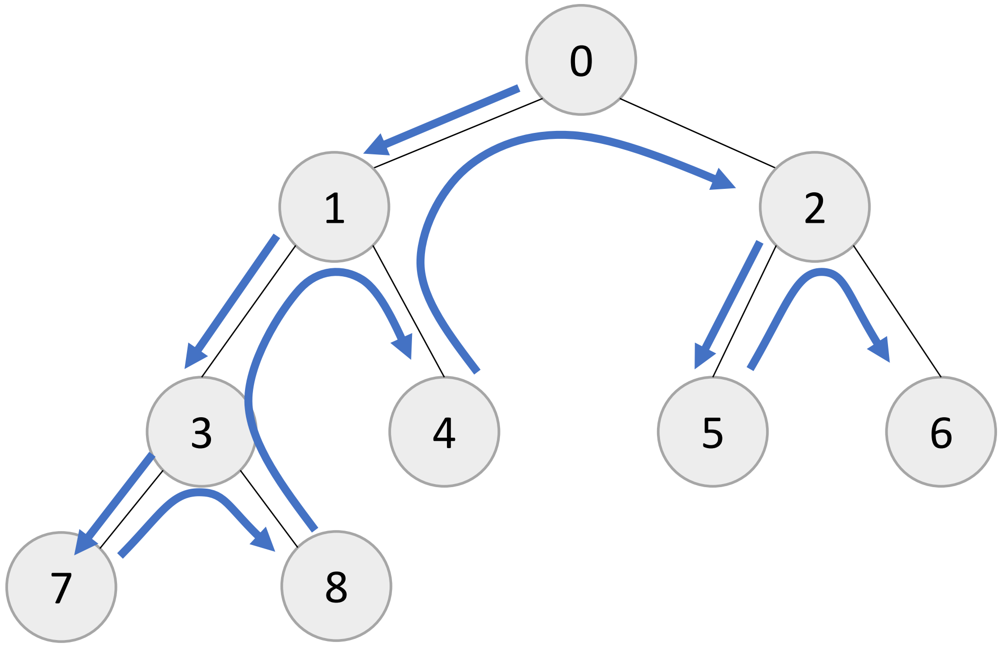
- Choose most promising node next according to some rule

# Best-First Search

- Choose most promising node next according to some rule
- Shallowest next:
  - Breadth-first search

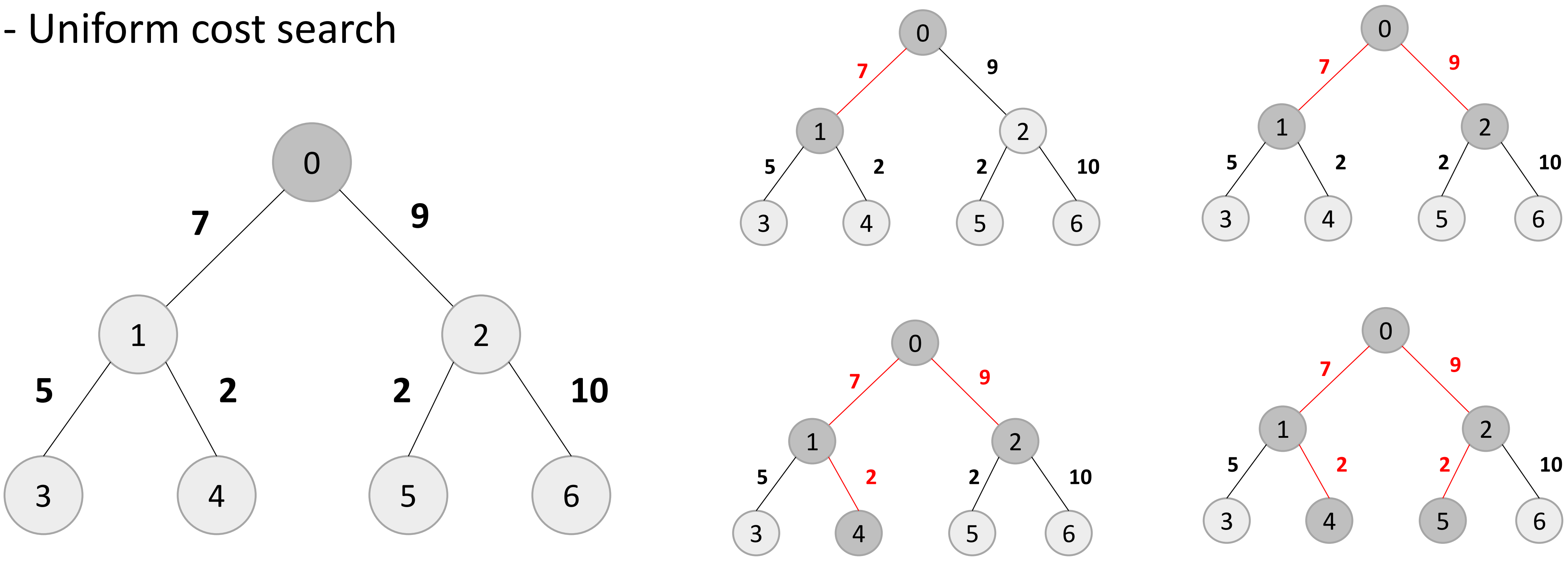



- Deepest next:
  - Depth-first search



# Best-First Search

- Choose most promising node next according to some rule
- Cheapest next:
  - Uniform cost search






HMCL High-assurance Mobility Control Lab

# Dijkstra's Algorithm

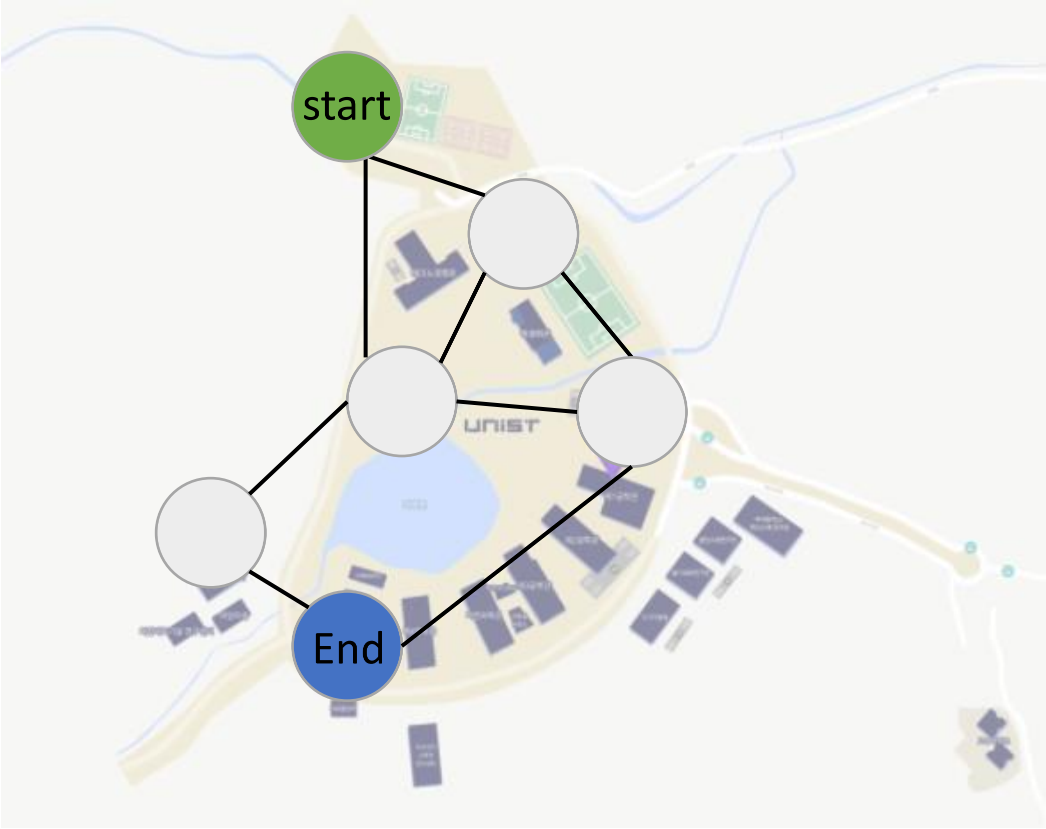
- For a node v, we want to:
  - Estimate the running cost g(v): the lowest cost to reach v from start

- Key Idea : ***Pick the frontier node that has the lowest running cost***

# Dijkstra’s Algorithm

```
function Dijkstra(Graph, source):
        create vertex set Q              // Unvisited set
        for each vertex v in Graph:
                g[v] ← INFINITY
                prev[v] ← UNDEFINED
                add v to Q
        g[source] ← 0

        while Q is not empty:
                u ← vertex in Q with min g[u]
                remove u from Q

                for each neighbor v of u:      // only v that are still in Q
                alt ← g[u] + cost(u, v)
                if alt < g[v]:                 // shorter path is found
                        g[v]  ← alt
                        prev[v] ← u
        return g[], prev[]
```

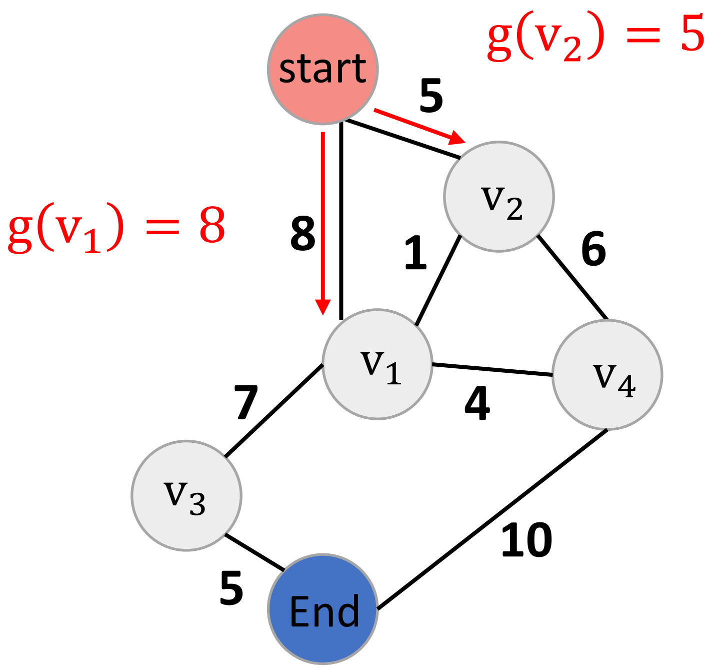
# Dijkstra's Algorithm

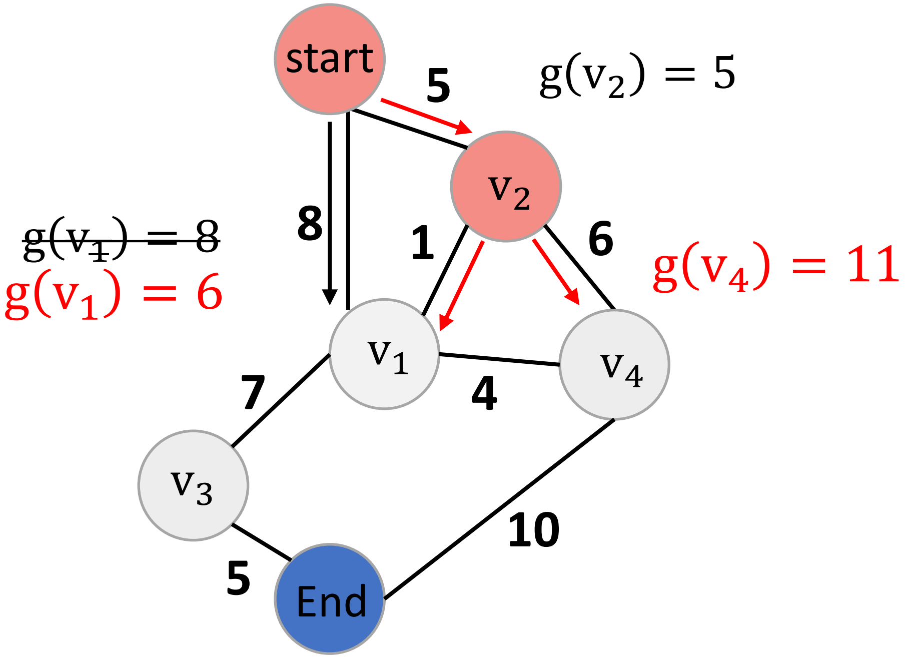



Goal: Find  the shortest path between two nodes on the graph

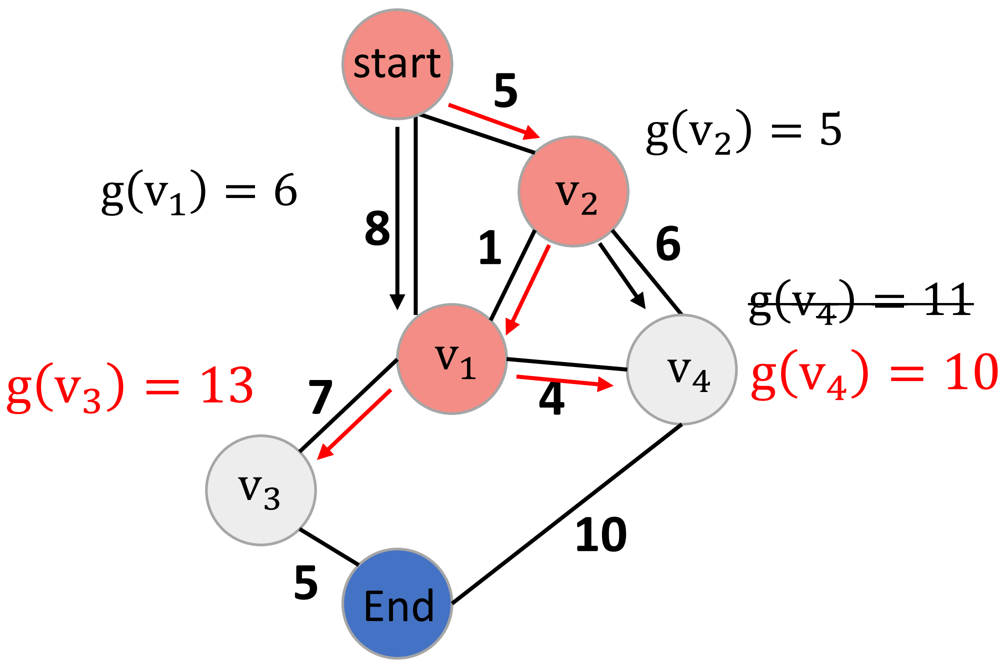
(e.g., find the shortest path from dormitory to bldg.112)

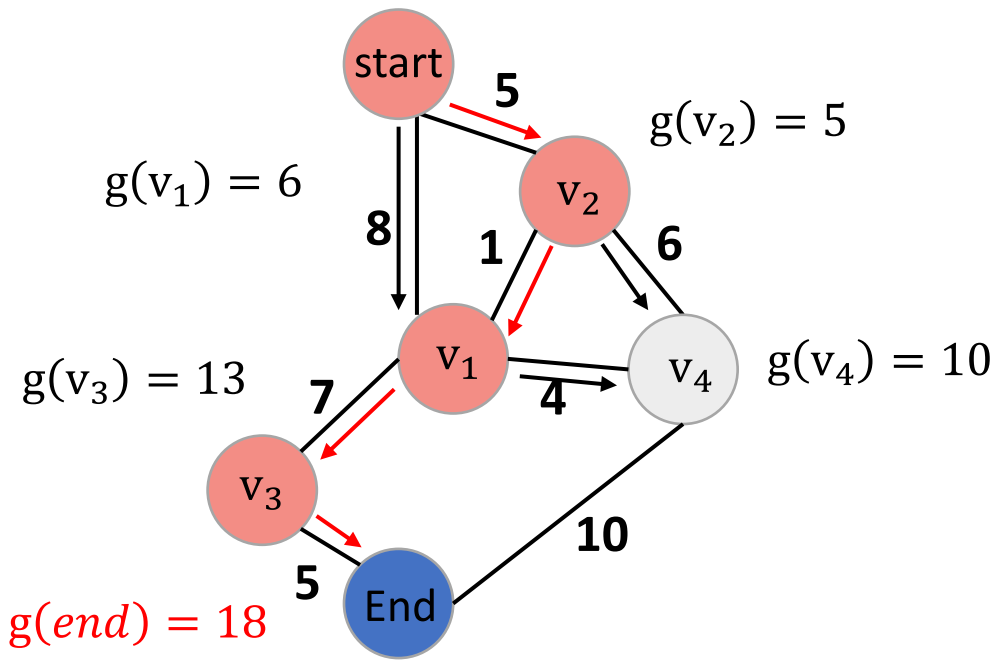
# Dijkstra's Algorithm



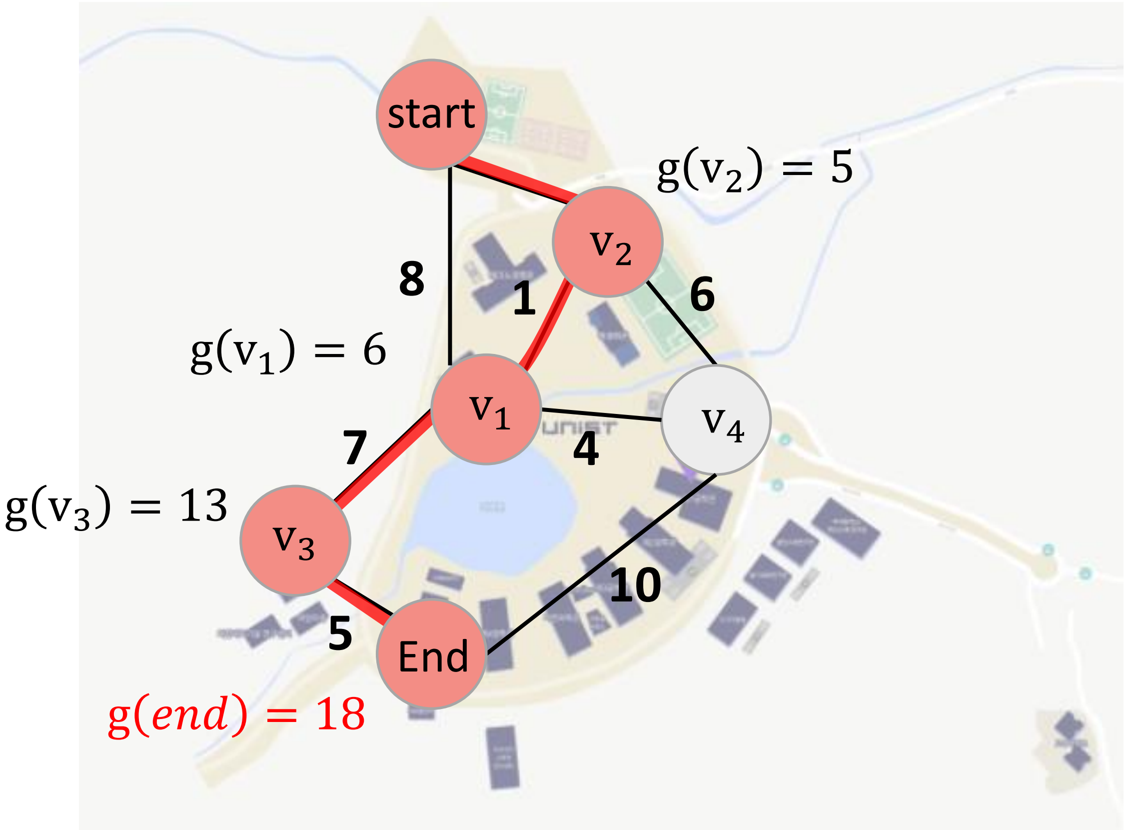



# Dijkstra's Algorithm



# Dijkstra's Algorithm



# A* Algorithm

- For a node v, we want to:
  - Estimate the running cost g(*v*): the lowest cost to reach *v* from start
  - (Under)estimate the cost to reach the goal

- For cost of a node v:
  - f(v) = g(v) + h(v)
  - h is a **heuristic** function that underestimates in cost to reach the goal from v

# Heuristic

- Heuristic functions:
  - Problem specific

  - Admissible: never overestimate the actual cost to get the goal

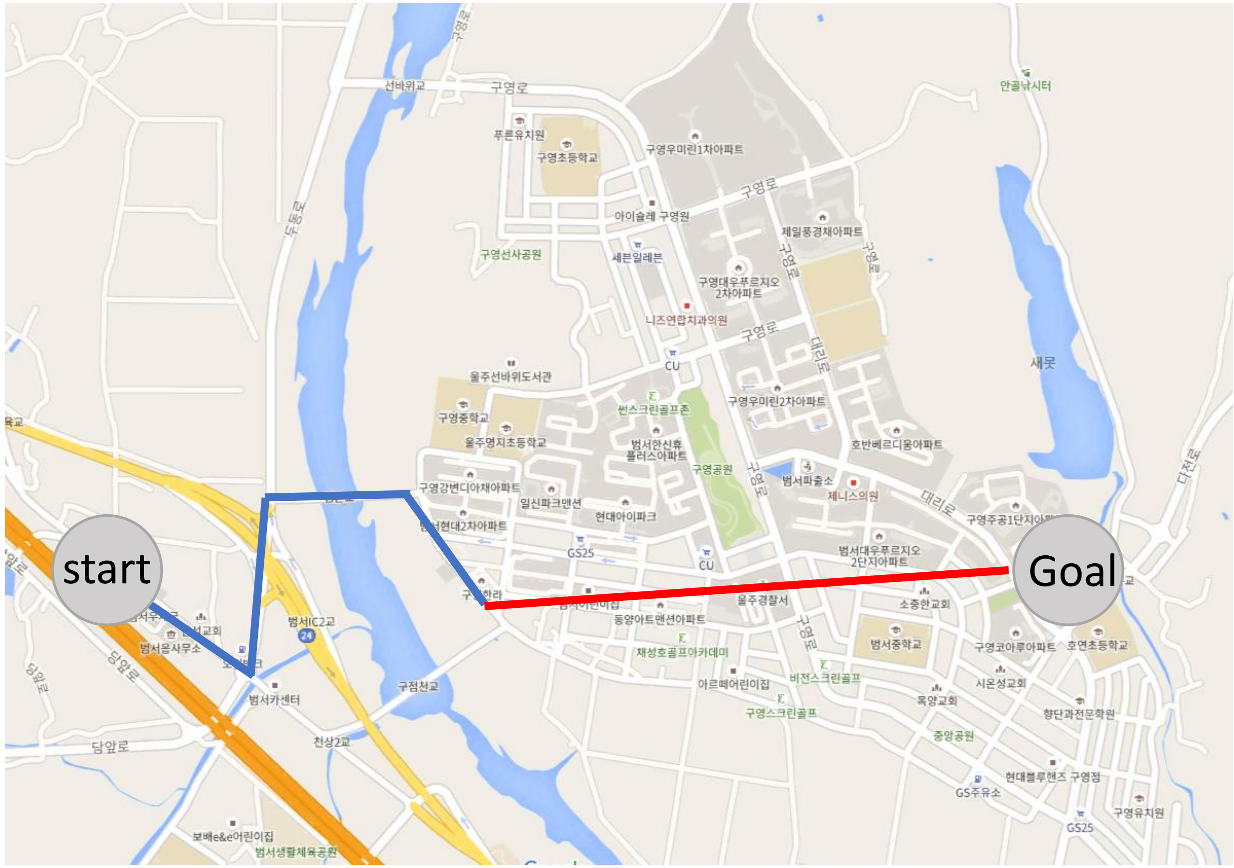
  - Consistent: h($\mathrm{v}_g$) = 0, and for every $\mathrm{v} \neq \mathrm{v}_g$,

$$\mathrm{h}(\mathrm{v}) \leq \mathrm{cost}\big(\mathrm{v}, \mathrm{successor}(\mathrm{v})\big) + \mathrm{h}\big(\mathrm{successor}(\mathrm{v})\big)$$

  - Consistency implies admissibility, not necessarily the other way around

# A* Algorithm

- In the example here, the Euclidean distance to the goal is used as heuristic



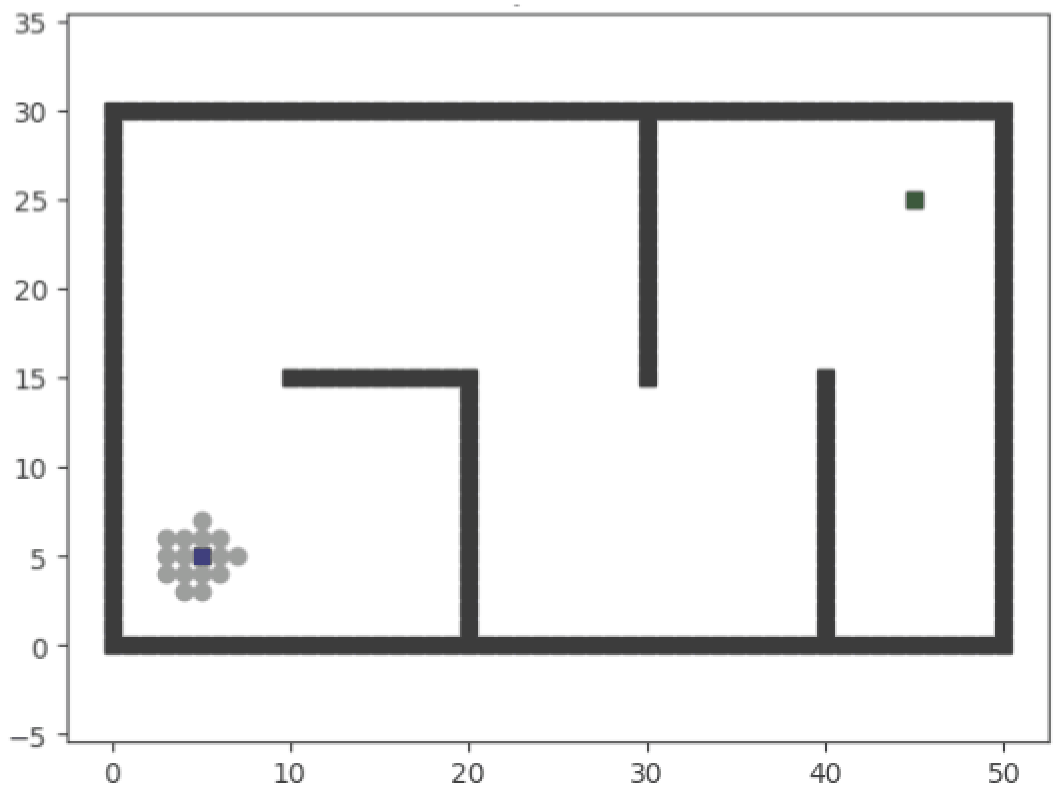

$f(v) = g(v) + h(v)$

# A* Algorithm

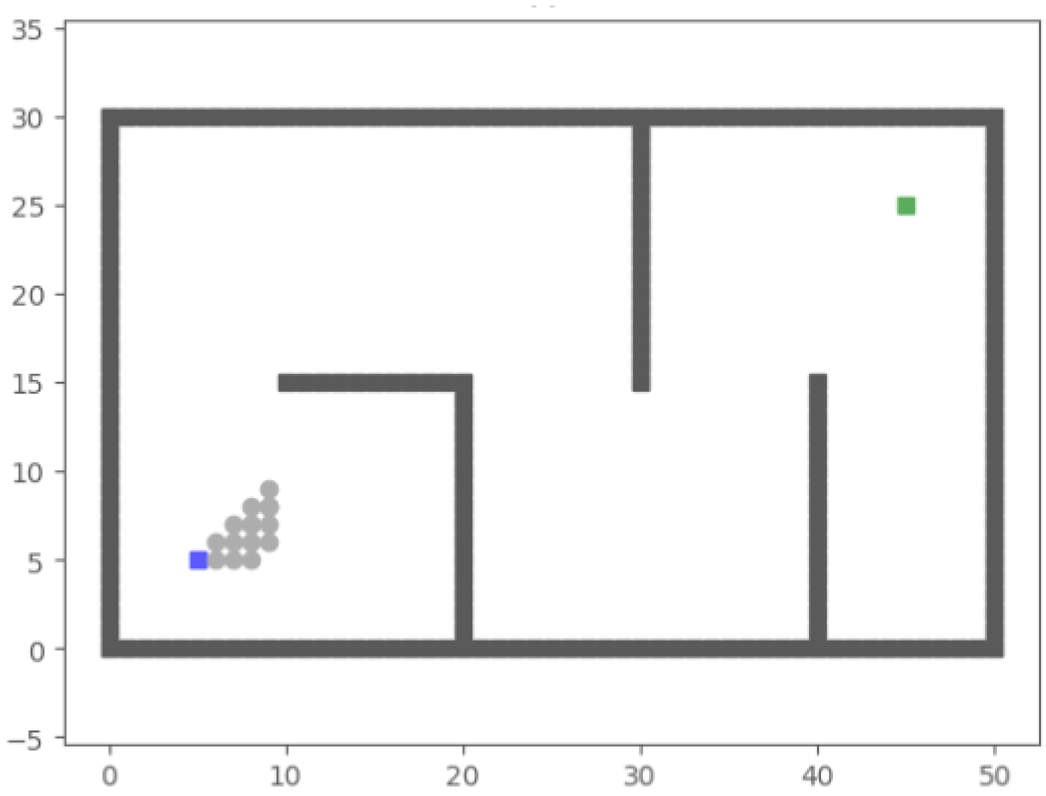
```
function A*(Graph, source):
        create vertex set Q                    // Unvisited set
        for each vertex v in Graph:
                g[v] ← INFINITY
                prev[v] ← UNDEFINED
                add v to Q
        g[source] ← 0

        while Q is not empty:
                u ← vertex in Q with min f[u]          //replace g[u] with f[u] = g[u] + h[u]
                remove u from Q

                for each neighbor v of u:      // only v that are still in Q
                alt ← g[u] + cost(u, v)
                if alt < g[v]:                 // shorter path is found
                        g[v]  ← alt
                        prev[v] ← u
        return g[], prev[]
```

# Dijkstra's vs A*

**Dijkstra's**




**A***



# Planning Challenges for Race Car

- **Dimensionality:** as the state of the robot we define encodes more information, the higher the dimensionality of the planning space
- **Responsiveness:** the planning algorithm needs to react fast when the car is travelling at high speeds
- **Obstacles:** potentially dynamic and moving obstacles(head-to-head racing)
- **Motion constraints:** kinematic or dynamic

# Sampling based Algorithm

Goal: Plan collision-free trajectories through cluttered space

# RRT

**function** rrt($x_{init}$, max _iter):

V ← $\{x_{init}\}$ *// vertices*

E ← Ø *// edges*

**for** i = 1:max_iter: *// maximum number of expansions*

$\mathrm{x_{rand}}$ ← SampleFree () *// collision free random configuration*

$\mathrm{x_{nearest}}$ ← Nearest (G = (V,E), $\mathrm{x_{rand}}$) *// closest neighbor in the tree*

$\mathrm{x_{new}}$ ←Steer($x_{nearest}$, $x_{rand}$) *// expanding the tree*

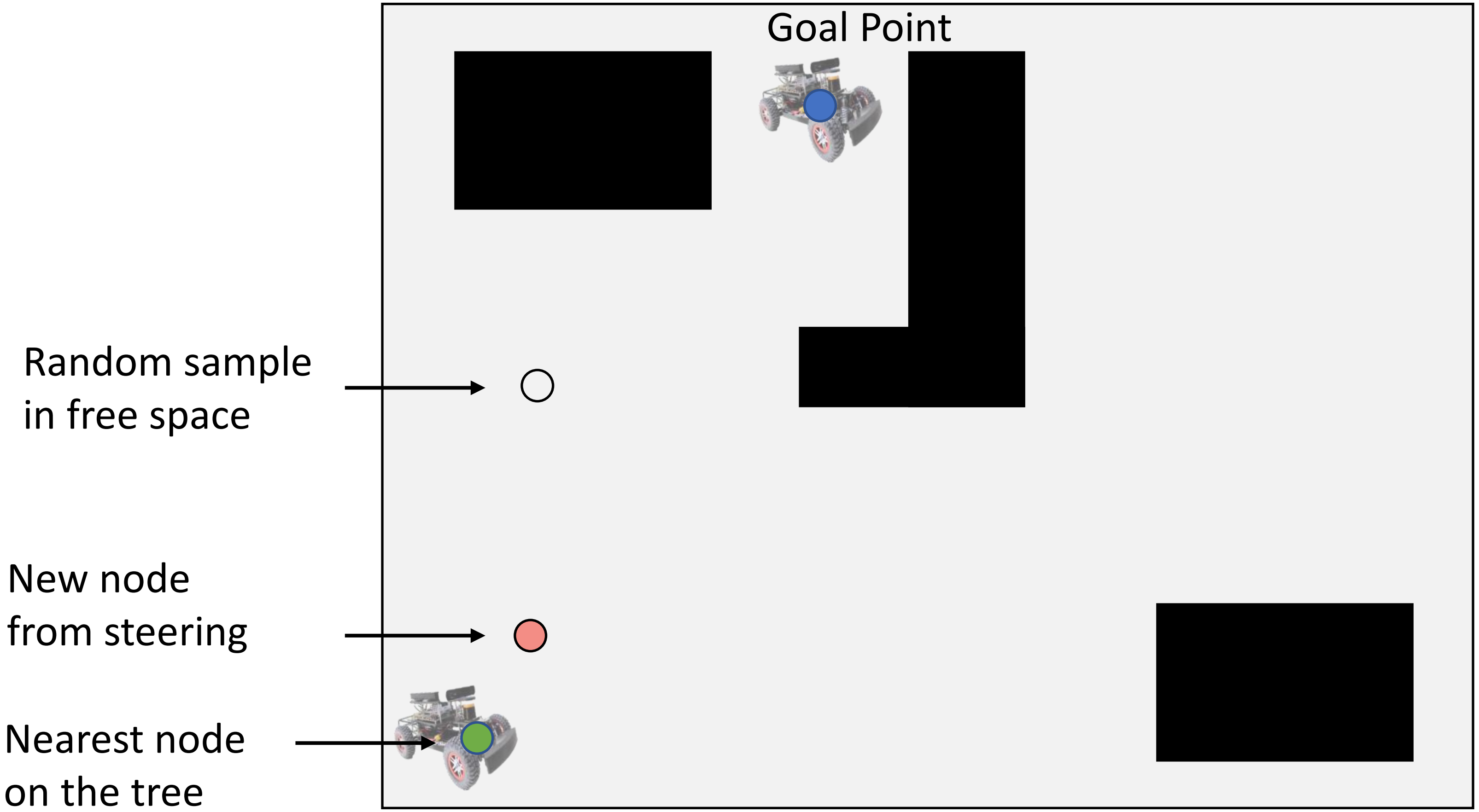
**if** ObstacleFree ($x_{nearest}$, $x_{new}$) *// the edge is collison-free*

V ← V ∪ $\{x_{new}\}$

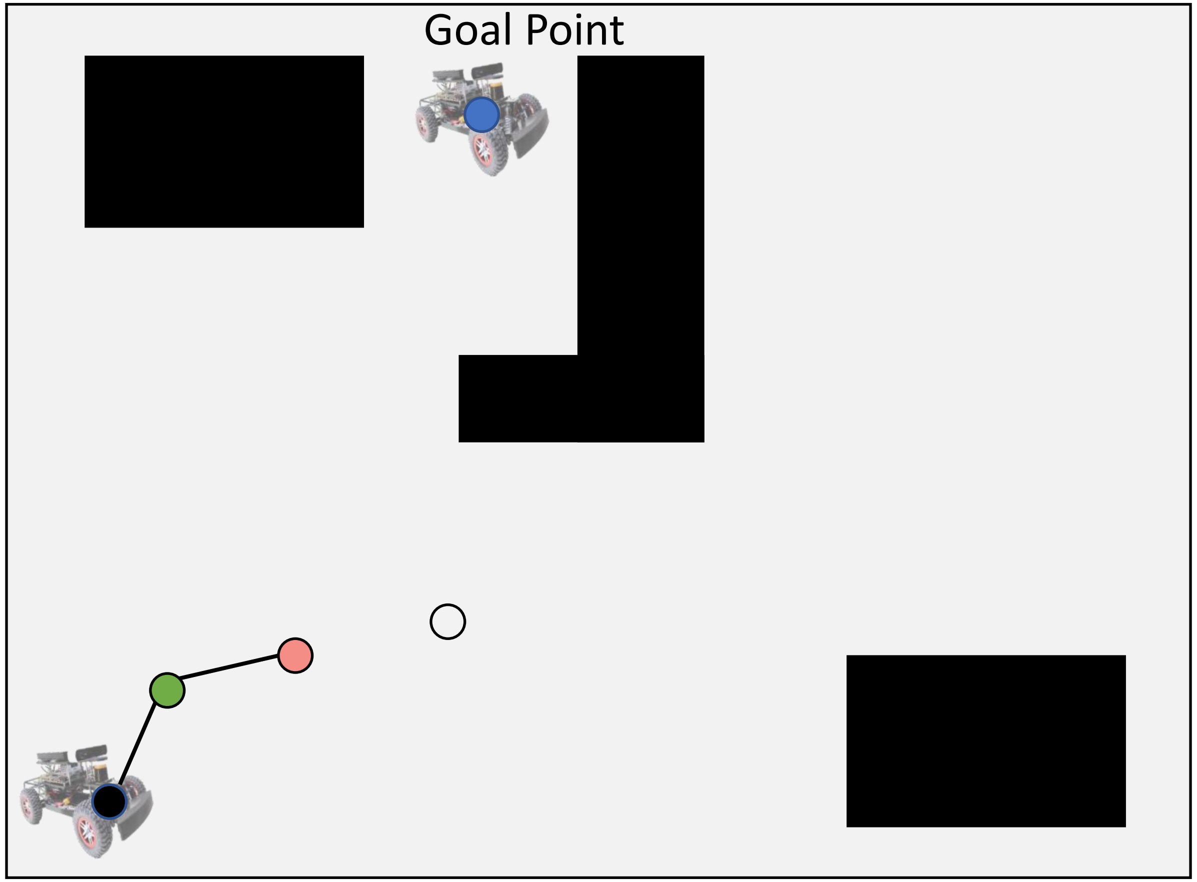
V ← E ∪ $\{(\mathrm{x}_{nearest}, x_{new})\}$

**return** G = (V,E)

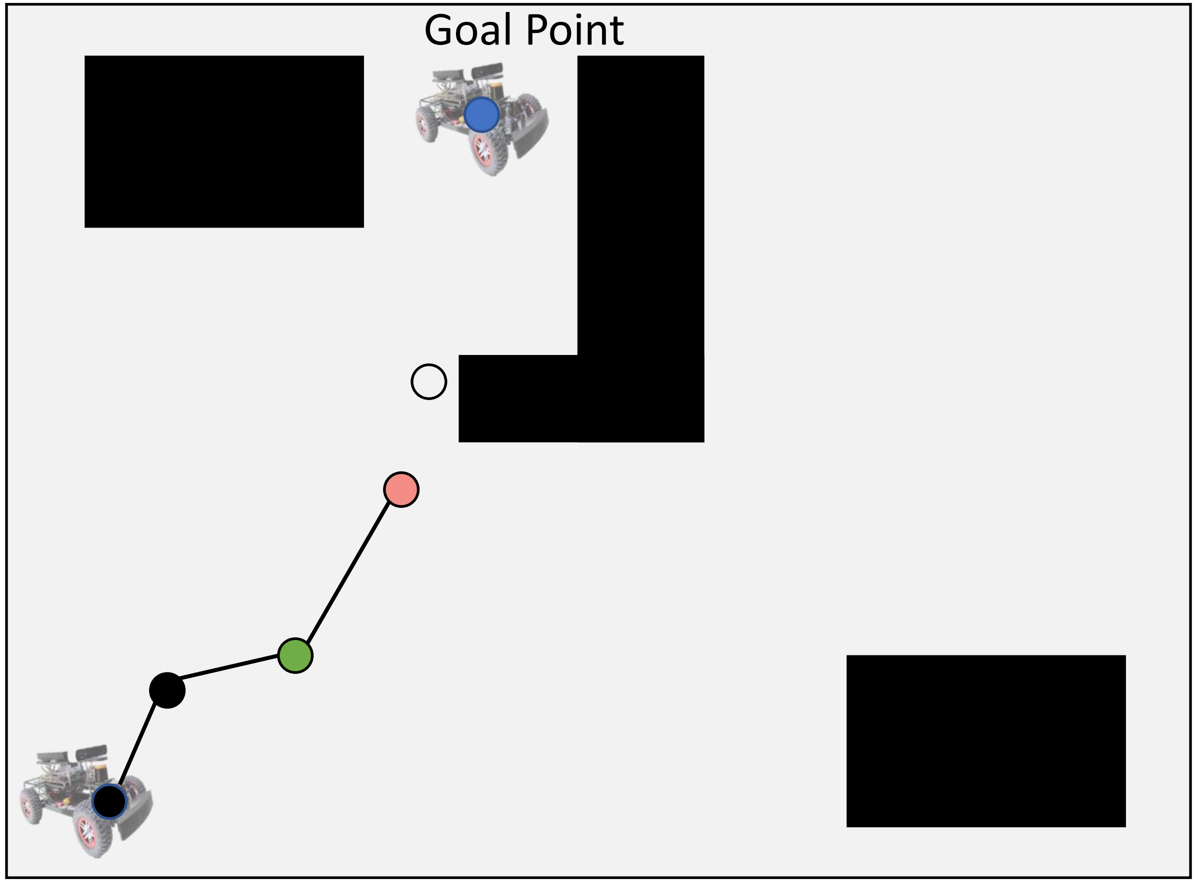
# RRT(Rapidly-exploring Random Trees)



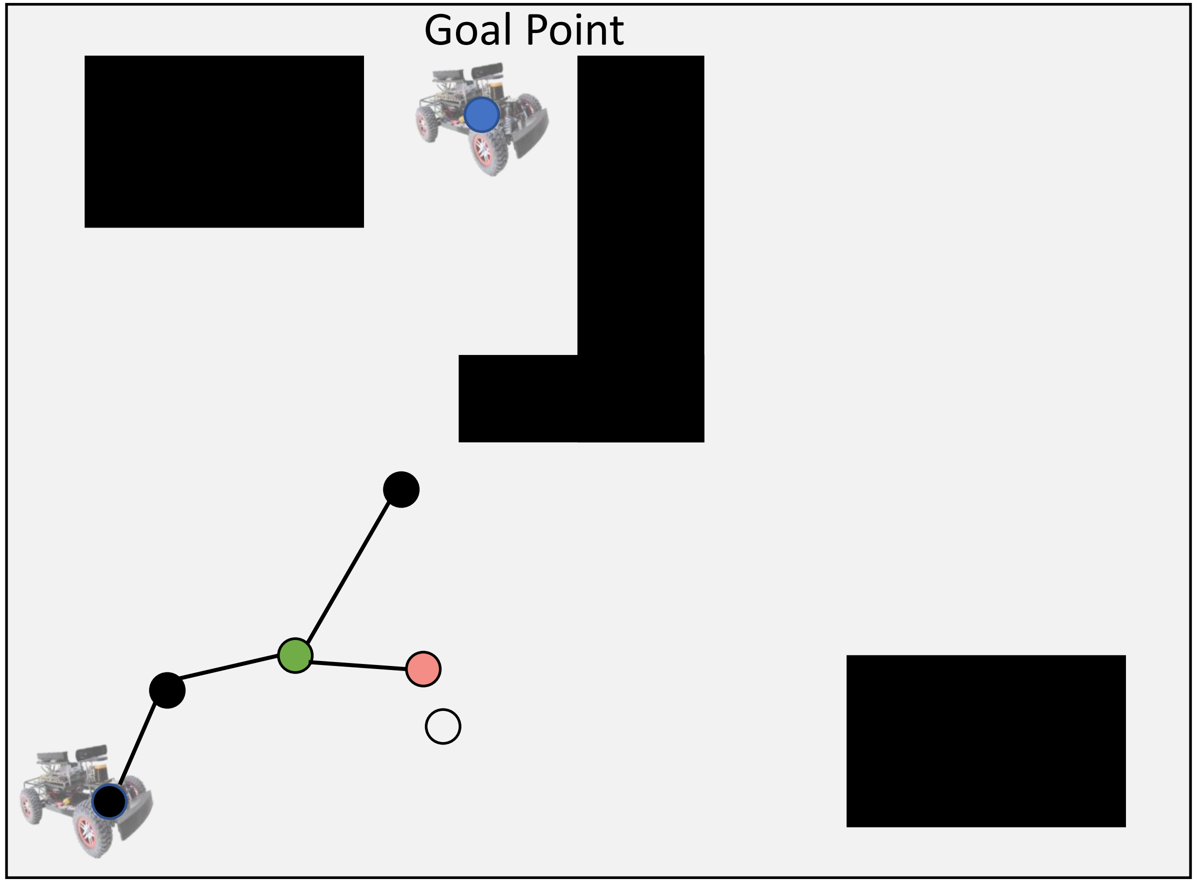
# RRT(Rapidly-exploring Random Trees)



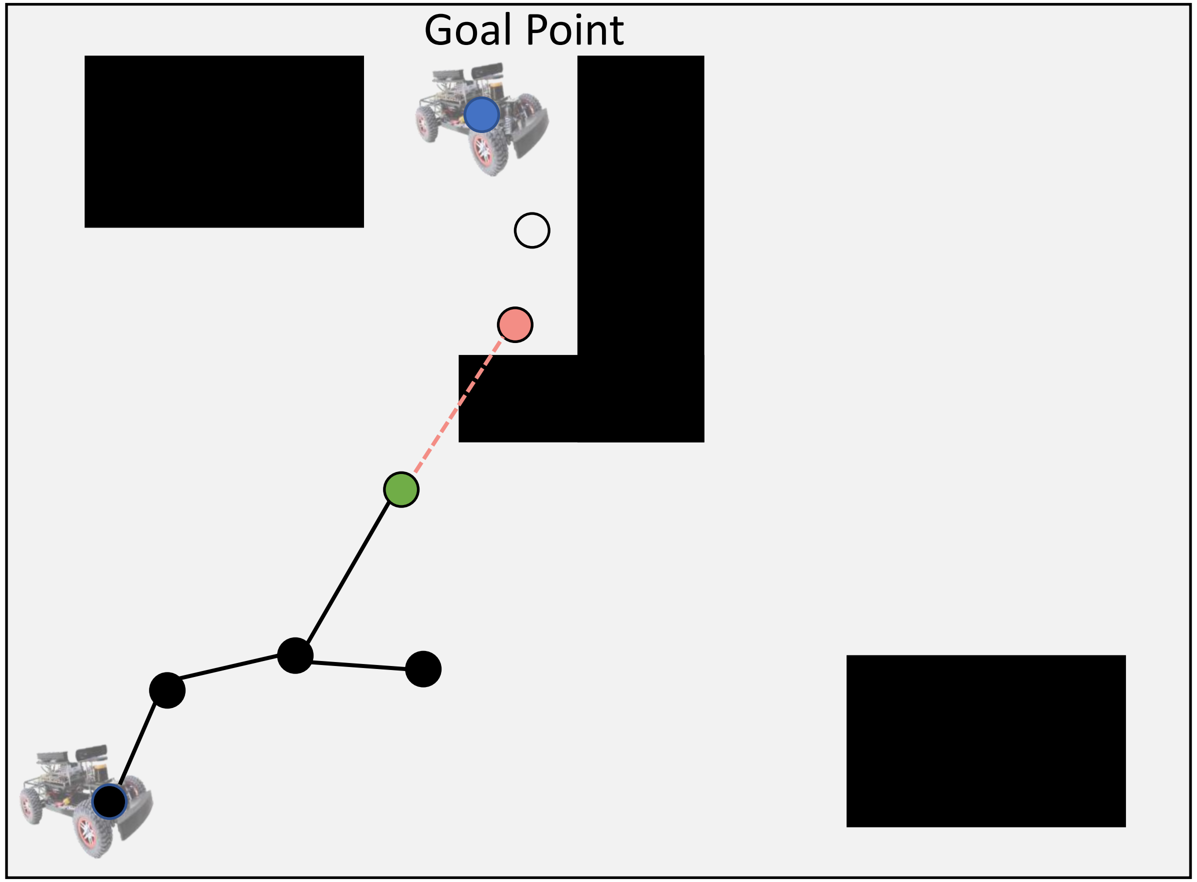
# RRT(Rapidly-exploring Random Trees)



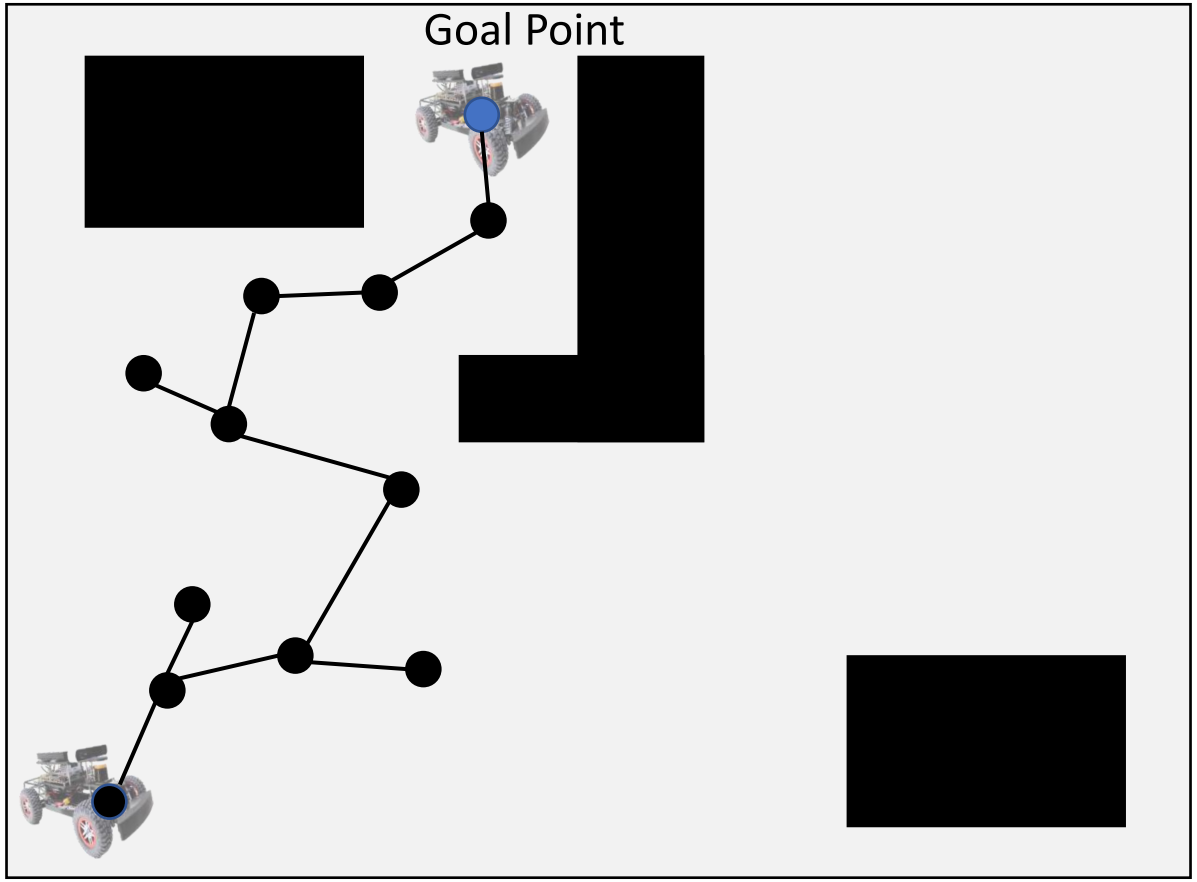
# RRT(Rapidly-exploring Random Trees)



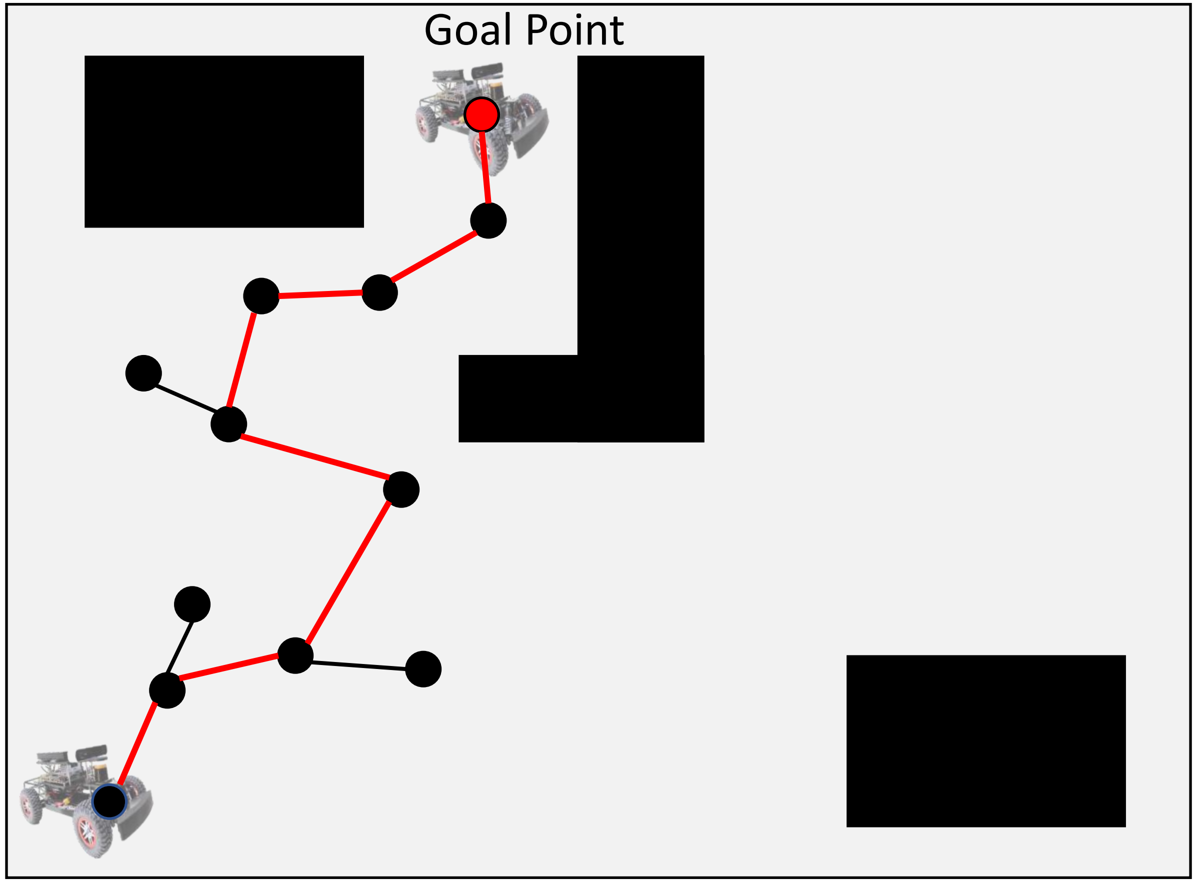
# RRT(Rapidly-exploring Random Trees)



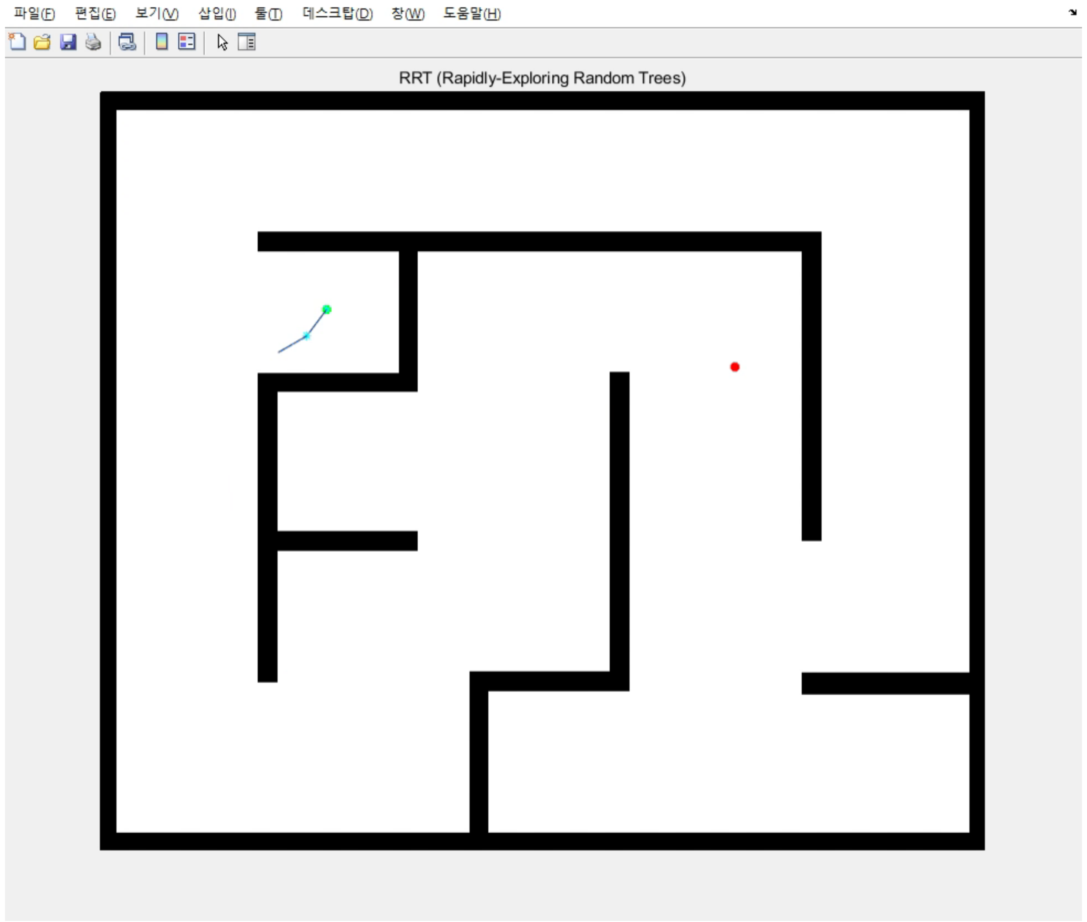
# RRT(Rapidly-exploring Random Trees)



# RRT(Rapidly-exploring Random Trees)

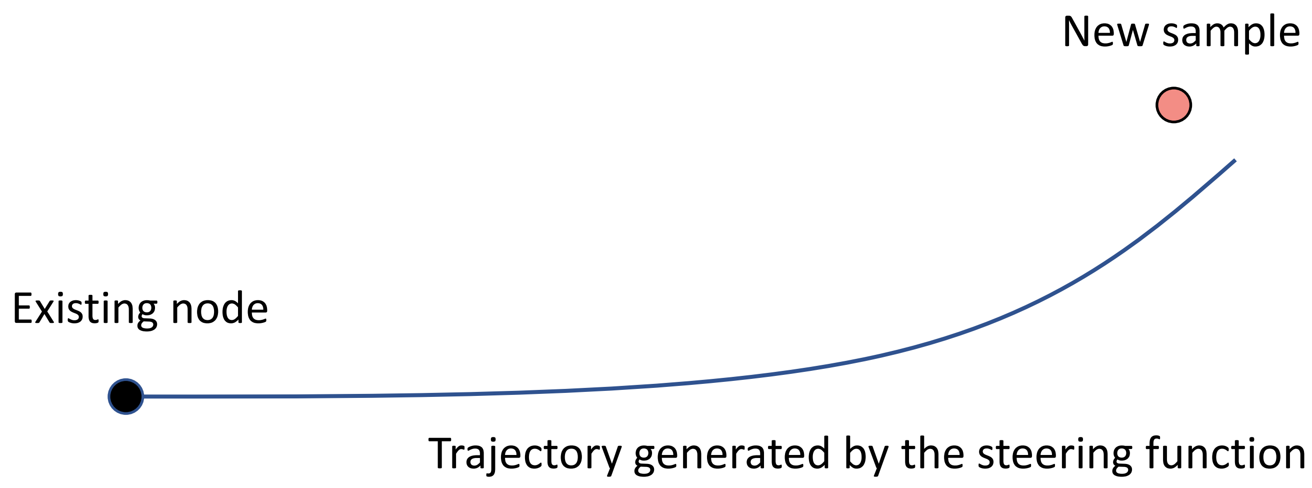


# RRT(Rapidly-exploring Random Trees)



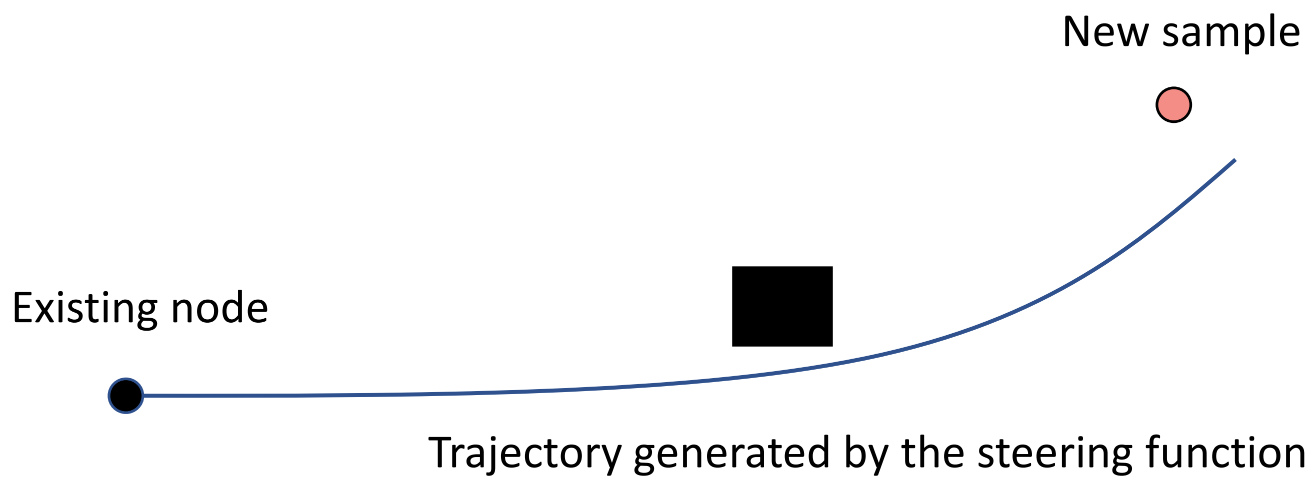
# RRT(Rapidly-exploring Random Trees)

- Need to design a “steering function “ for your system based on its constraints



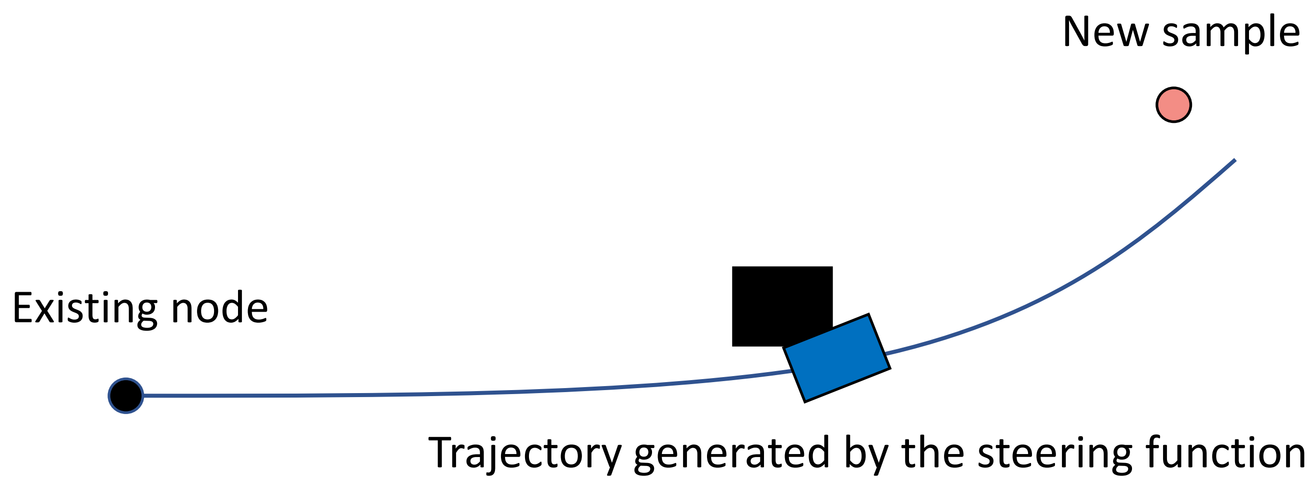
# RRT(Rapidly-exploring Random Trees)

- Need to design a collision checking function for your system



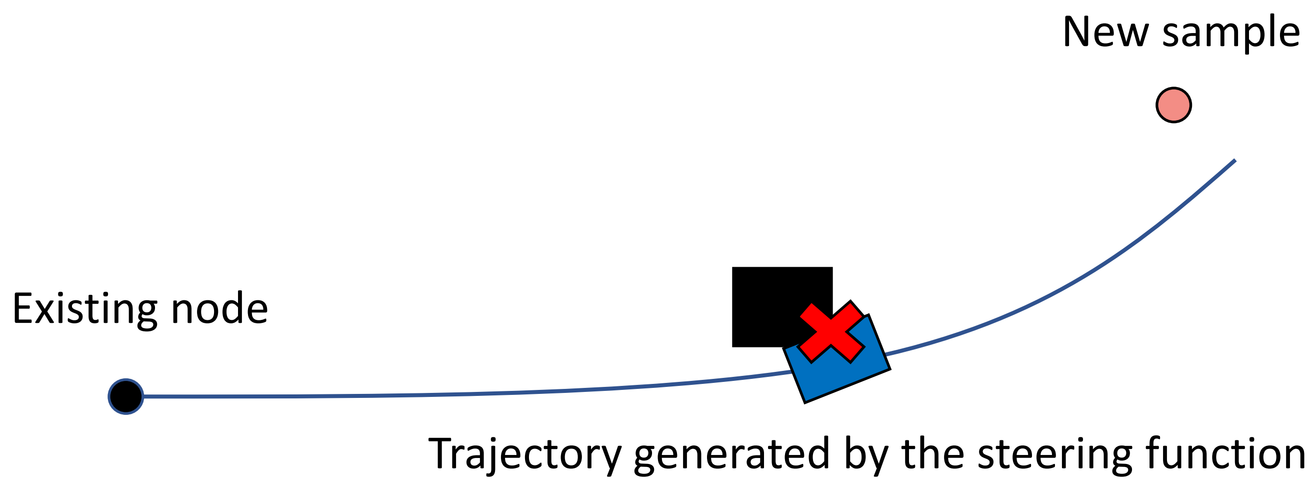
# RRT(Rapidly-exploring Random Trees)

- Need to design a collision checking function for your system



# RRT(Rapidly-exploring Random Trees)

- Need to design a collision checking function for your system



# Path with a velocity profile

- The top speed of the car at certain points on the path is constraint by the curvature of the path at those points
- We'll denote a path, along with the velocity profile on the path as **trajectories**

# Optimizing for speed

- A naïve approach:
- Curvature limits the maximum speed the car can achieve on the curve
- Find maximum curvature locations on the curve
- Define speed limit on these maximum curvature location
- Come up with a full speed profile by interpolating velocities in between locations on the curve

- An efficient way to do this is to use **(Convex) Optimization**

# Convex Optimization

- Mathematical optimization that studies the problem for minimizing convex functions over convex sets
- A standard form of a convex optimization problem:

$$\underset{x}{\text{minimize}}\ f(x)$$

$$\begin{aligned} \text{subject to}\quad & g_i(\mathrm{x}) \le 0, \qquad i = 1, \dots, m \\ & h_i(\mathrm{x}) = 0, \qquad i = 1, \dots, p \end{aligned}$$

- Where $x$ is the optimization variable, $f$ is the objective function, functions $g_i$ and $h_i$ are constraint functions. The functions $f$ and $g_i$ are convex, and the functions $h_i$ are affine

# What are the constraints?

- The dynamics of the vehicle:

Vehicle State Control Input

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

**Kinematic Bicycle Model**

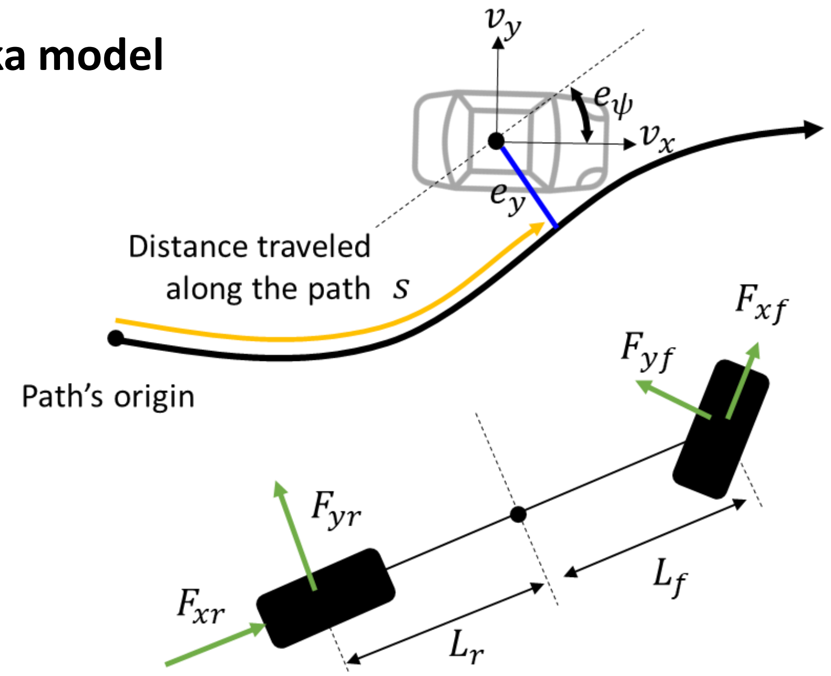
$$\dot{x} = \begin{bmatrix} v\cos\varphi \\ v\sin\varphi \\ \frac{v}{L}\tan(\delta) \\ a \end{bmatrix}$$

$$x = [x, y, \varphi, v]^T,\ u = [a, \delta]^T$$

**Dynamic bicycle model** with **simplified Pacejka model**

$$\dot{x} = \begin{bmatrix} \frac{v_x\cos(e_\psi) - v_y\sin(e_\psi)}{1 - k(s)e_y} \\ v_x\sin(e_\psi) + v_y\cos(e_\psi) \\ w - \frac{v_x\cos(e_\psi) - v_y\sin(e_\psi)}{1 - k(s)e_y}k(s) \\ a_x - \frac{1}{m}(F_{yf}\sin(\delta) + mv_y w) \\ \frac{1}{m}(F_{yf}\cos(\delta) + F_{yr}) - \dot{\psi}v_y \\ \frac{1}{I_z}(L_f F_{yf}\cos(\delta) - F_{yr}L_r) \end{bmatrix}$$

$$x = [s, e_y, e_\psi, v_x, v_y, w]^T,\ \ u = [a, \delta]^T$$



# What are the constraints?

- Constraints on state and input:

$$x \in X, \qquad u \in U$$

- Ensure the vehicle remains within the track, adhering to the physical limits of the vehicle.

# Convex Optimization

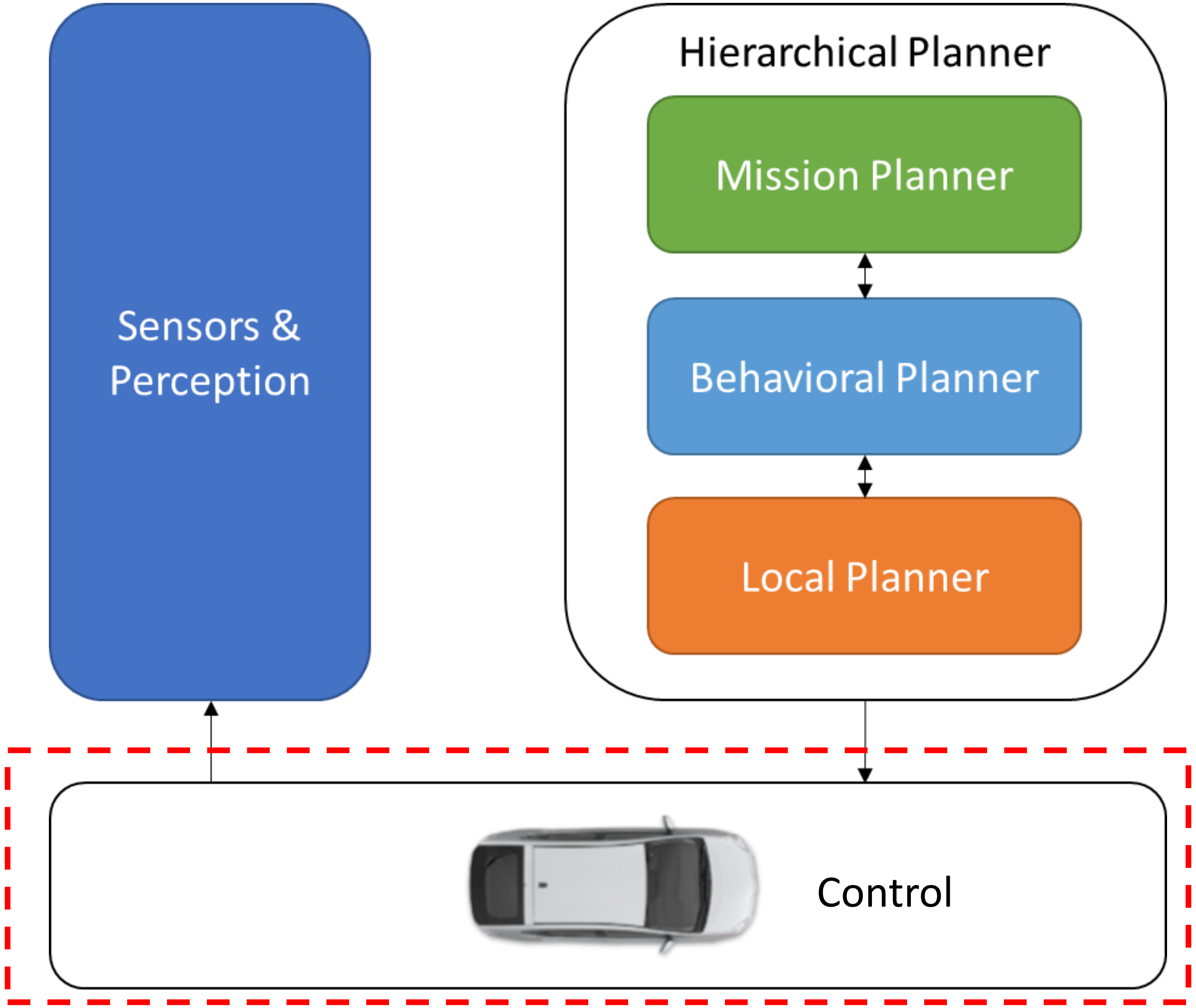
- Putting it all together our goal is to minimize T subject to the constraints defined in the previous slides

$$\begin{aligned} &\text{minimize} \;\; \mathrm{T} \\ &\text{subject to} \quad \dot{x} = f(x, u), \\ &\qquad\qquad\quad\; x \in X, \quad u \in U \end{aligned}$$

- How to solve? Use different optimization tools, e.g., CVX, etc.

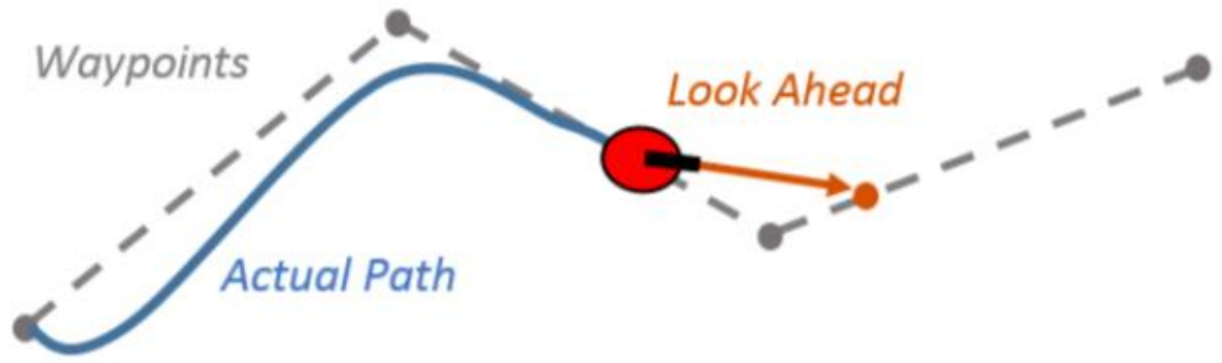
# Autonomous Vehicles Planning and Control Stack

- How do we track a given trajectory?
- How do we correct for actuation errors?



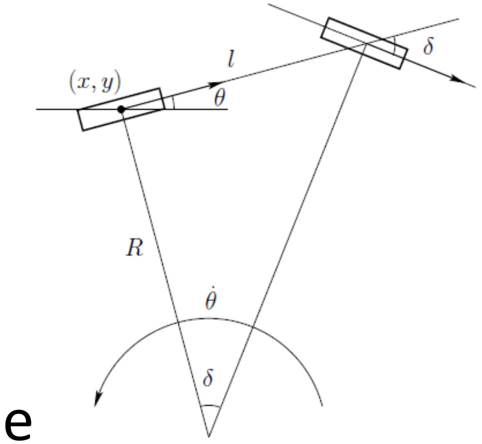
# Assumptions

- Vehicle is given a sequence of 2D position, i.e. waypoints, to follow
- Vehicle knows where the given waypoints are in the vehicle's frame of reference
  - **Underlying assumptions that the vehicle can localize itself**
- Goal is to follow these waypoints



# Revisit to Vehicle Kinematics: Bicycle Model

- The bicycle model (simplified form of the four-wheel Ackerman steering kinematics) is used to derive a steering angle for the pure pursuit method
  - $\dot{x}_{p_v} = v\cos(\theta_{car})$
  - $\dot{y}_{p_v} = v\sin(\theta_{car})$
  - $\dot{\theta}_{car} = \frac{v\tan(\delta)}{L}$

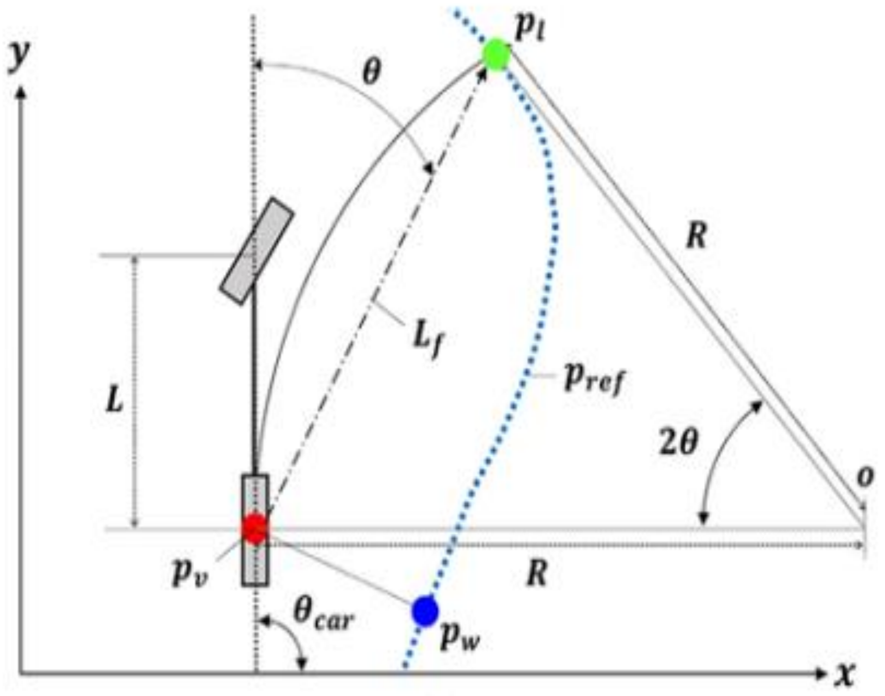



- $p_v$ is the center of the rear wheel of the vehicle position
- $x_{p_v}$ and $y_{p_v}$ are the Cartesian coordinates of the vehicle in the global frame
- $\theta_{car}$ is the vehicle's orientation in the global frame
- $\delta$ is the steering angle in the vehicle frame
- $v$ is the longitudinal velocity
- $L$ is the wheelbase

# Pure Pursuit Method Geometric Interpretation

- The pure pursuit method tracks the path $p_{ref}$ by calculating the look-ahead point $p_l$ on the $p_{ref}$
- $L_f : Look - ahead\ distance$
- $R\ : radius\ of\ arc$
- $\theta\ : Look - ahead\ heading$
- $p_l\ : Look - ahead\ point$
- $p_w\ : closest\ point$

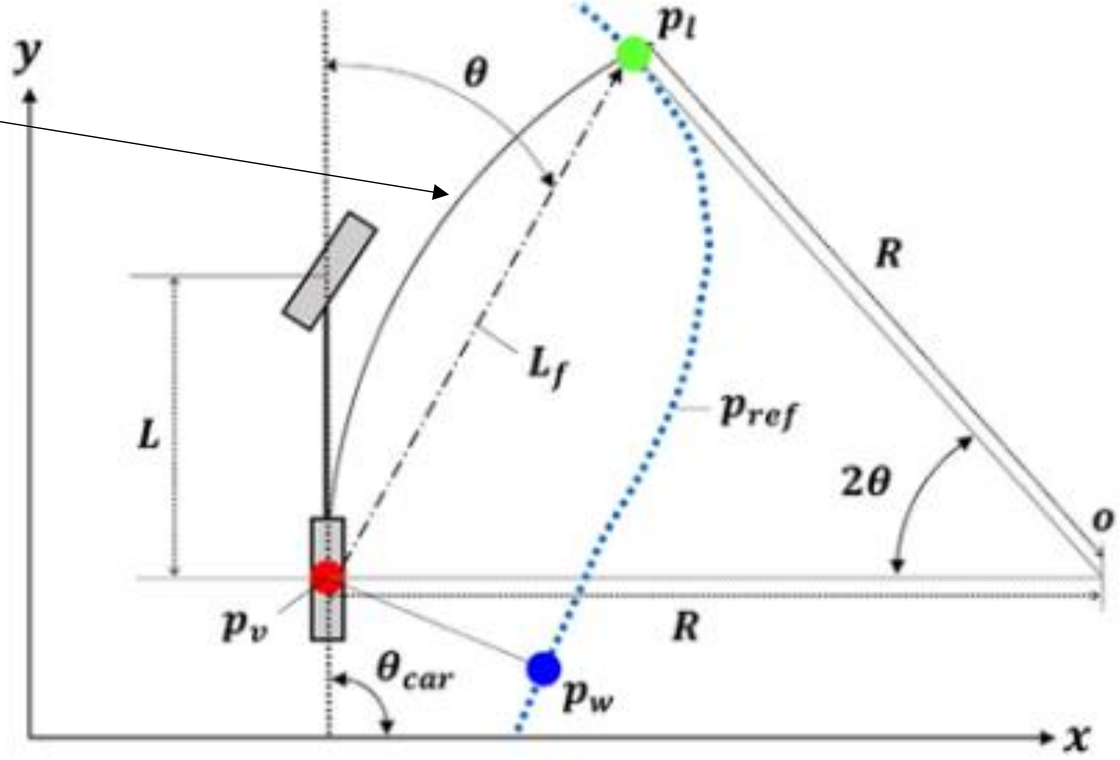
- $v$ is assumed to be constant



# Pure Pursuit Method Geometric Interpretation

Follow the arc of a circle
to reach the goal point

What else we need to set?



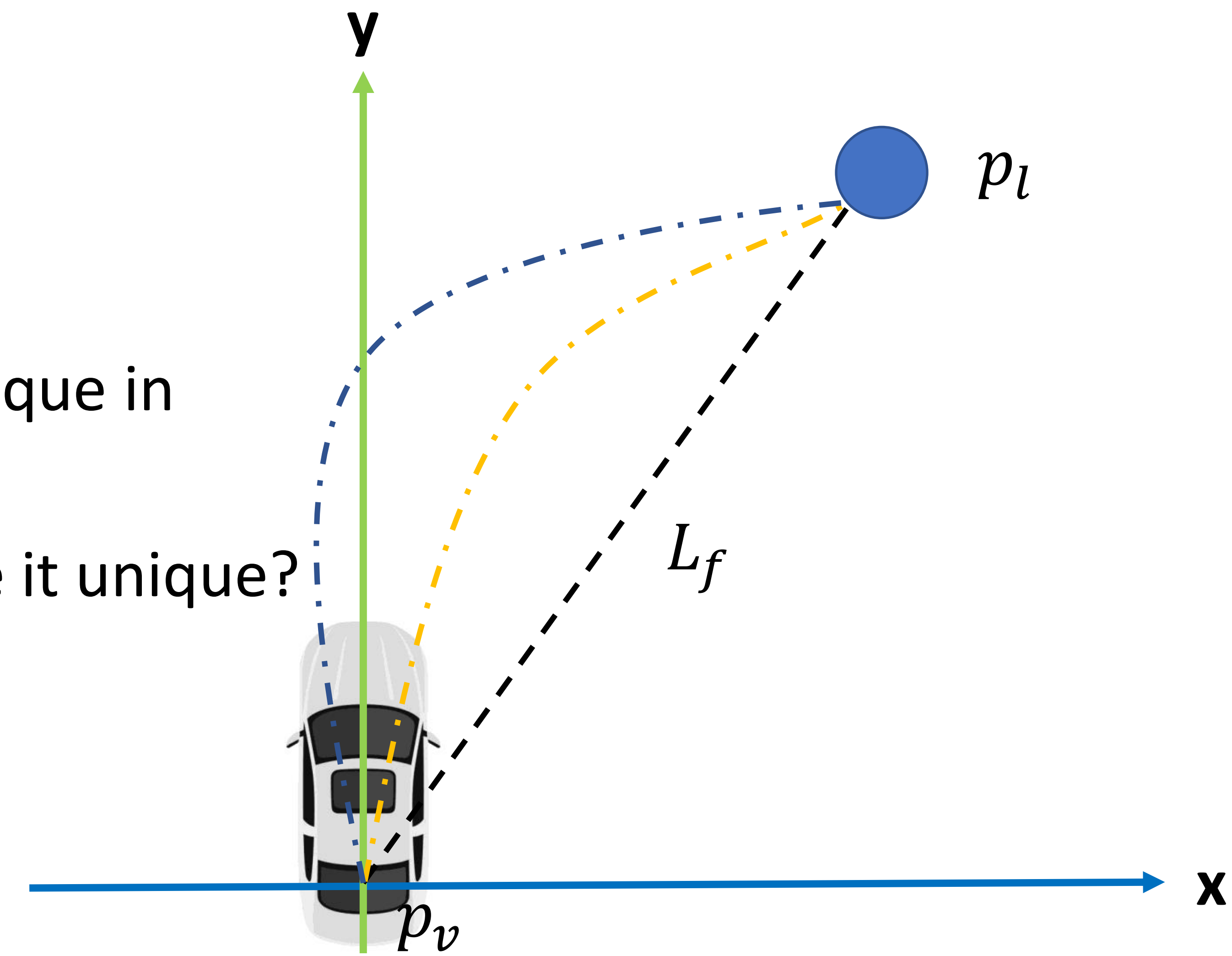



HMCL
High-assurance Mobility Control Lab

# Geometric Interpretation

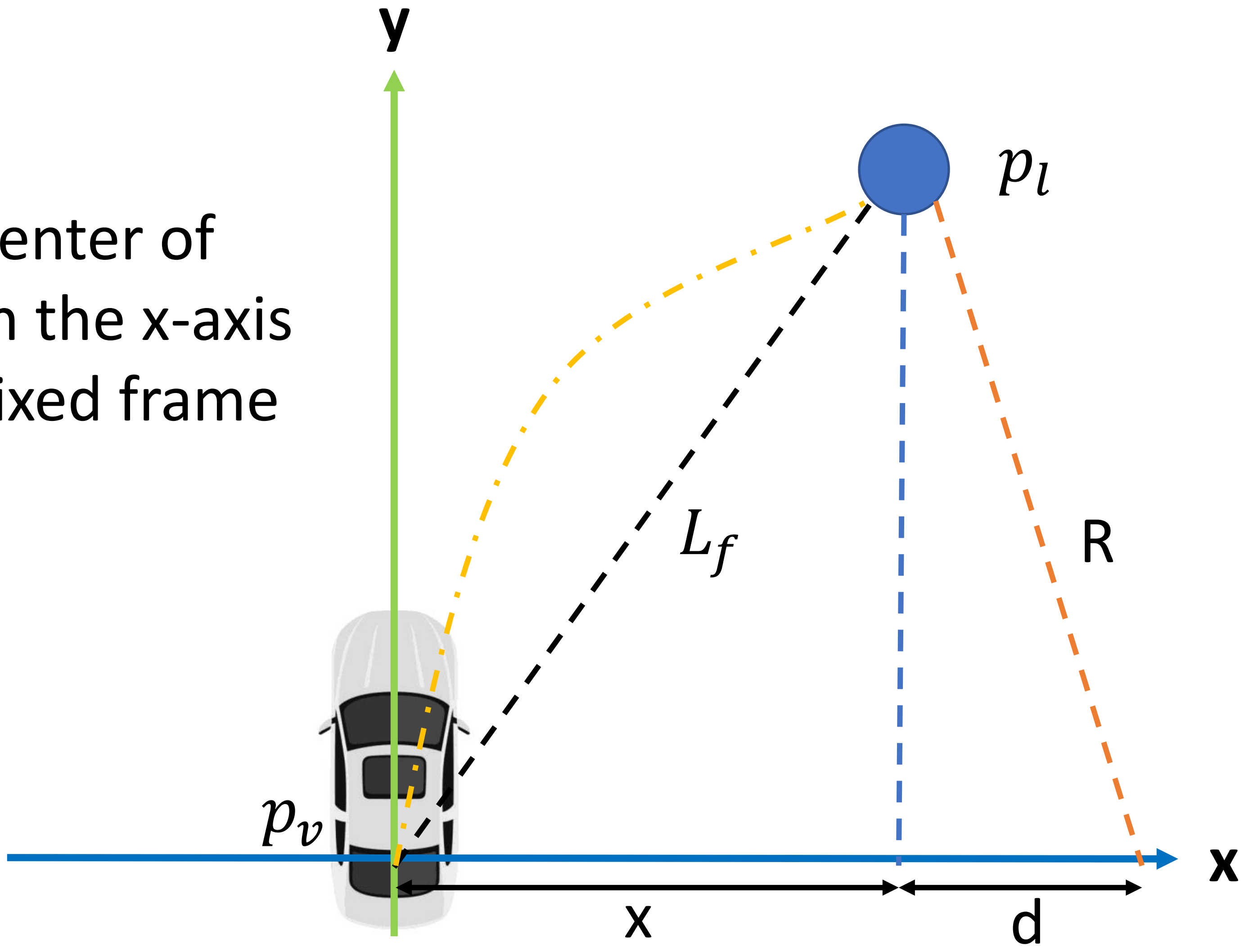
The arc is not unique in general

How do we make it unique?



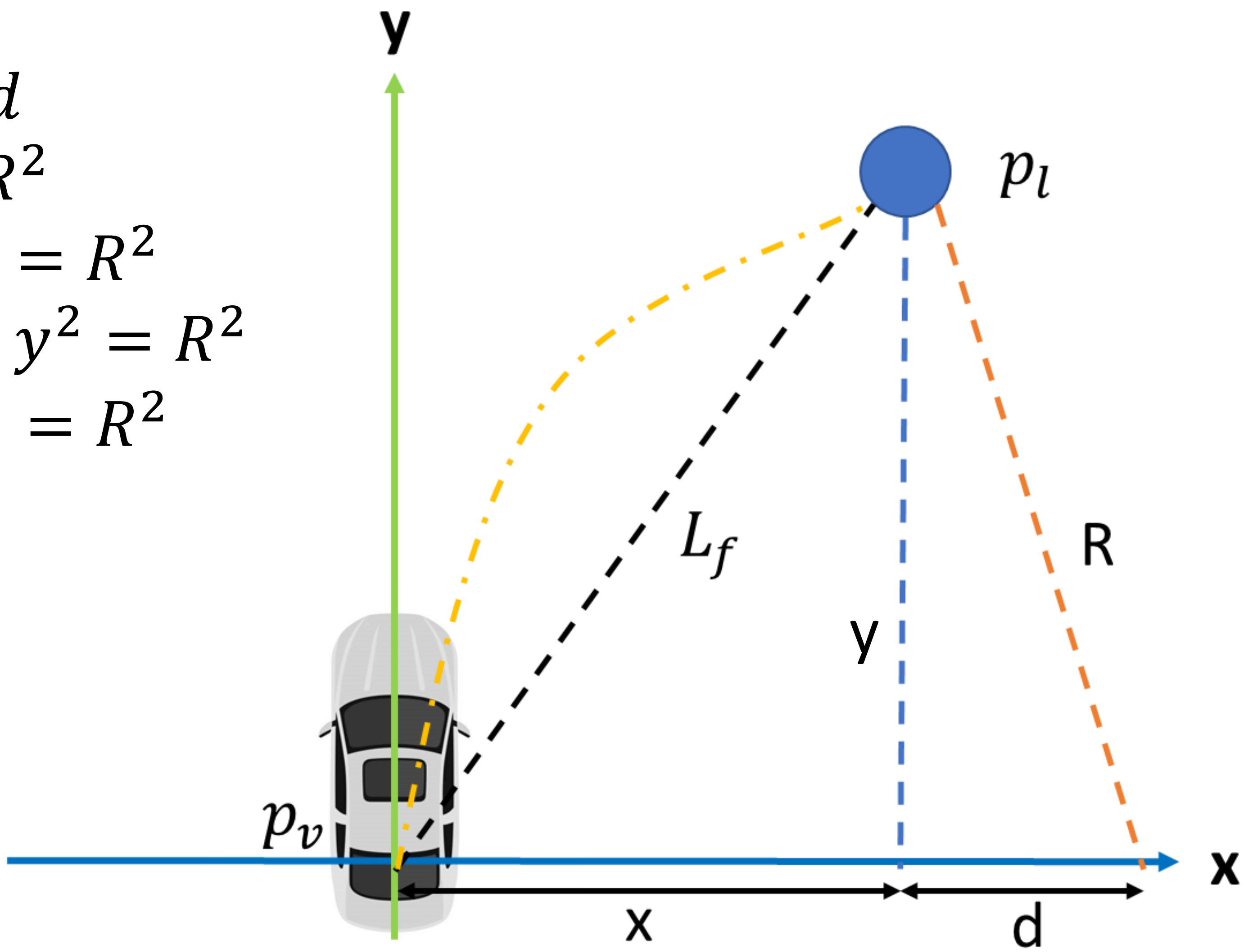
# Geometric Interpretation

Constrain the center of the arc to be on the x-axis of rear-wheel fixed frame



# Some simple math

$$R = |x| + d$$
$$d^2 + y^2 = R^2$$
$$(R - |x|)^2 + y^2 = R^2$$
$$R^2 + x^2 - 2R|x| + y^2 = R^2$$
$$R^2 + L_f^2 - 2R|x| = R^2$$
$$R = \frac{L_f^2}{2|x|}$$



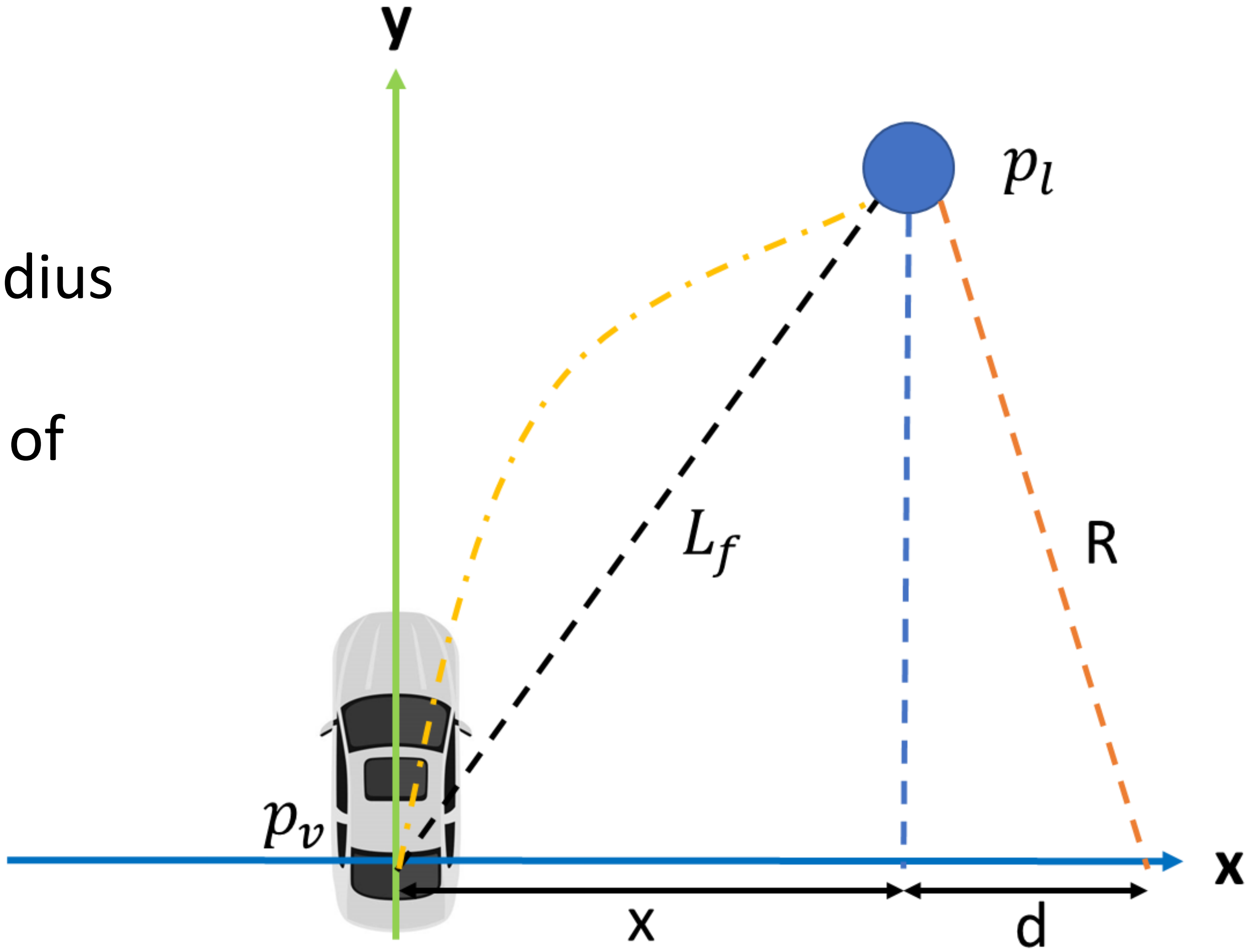
# Some simple math

$$R = \frac{L_f^2}{2|x|}$$

- Curvature is the inverse of radius
- Steering angle should be **proportional** to the curvature of the arc

$$\gamma = \frac{1}{R} = \frac{2|x|}{L_f^2}$$

Looks familiar? P control



# Some simple math

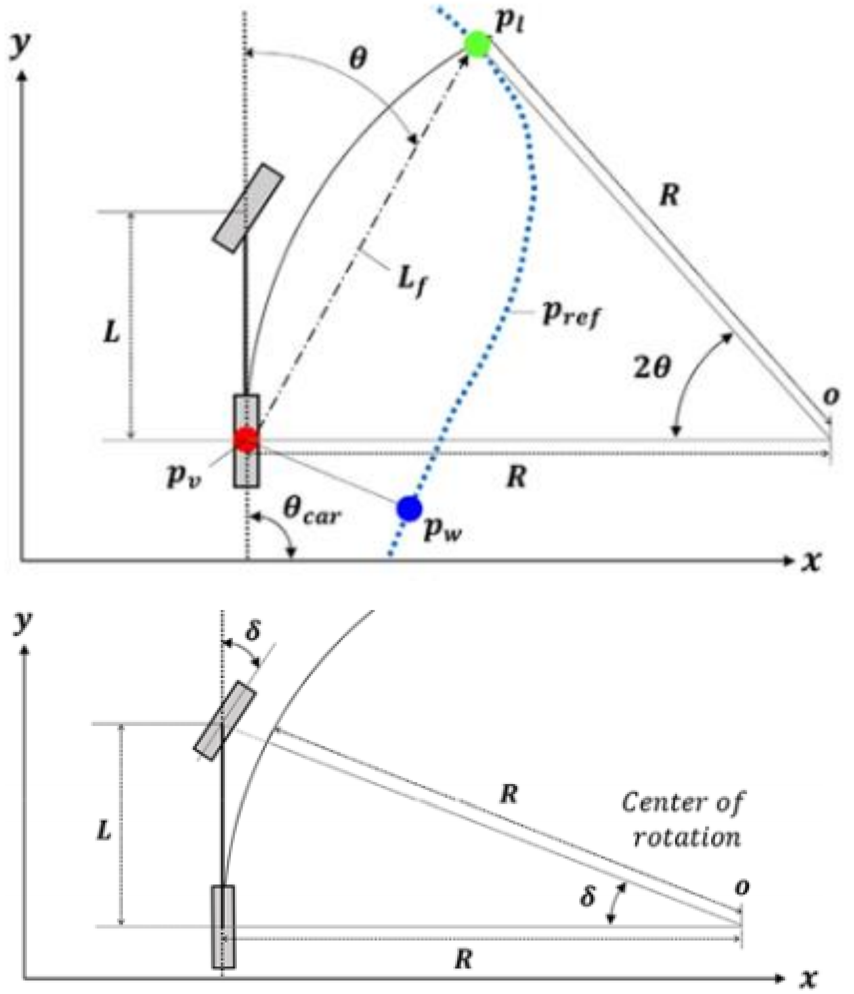
$$x = L_f \sin(\theta)$$

$$\gamma = \frac{1}{R} = \frac{2|x|}{L_f^2} = \frac{2\sin(\theta)}{L_f}$$

- There exists a geometric relationship between the vehicle steering angle ($\delta$) and R :

Bicycle steering geometry

$$\tan(\delta) = \frac{L}{R}$$




Bicycle steering geometry

# Some simple math

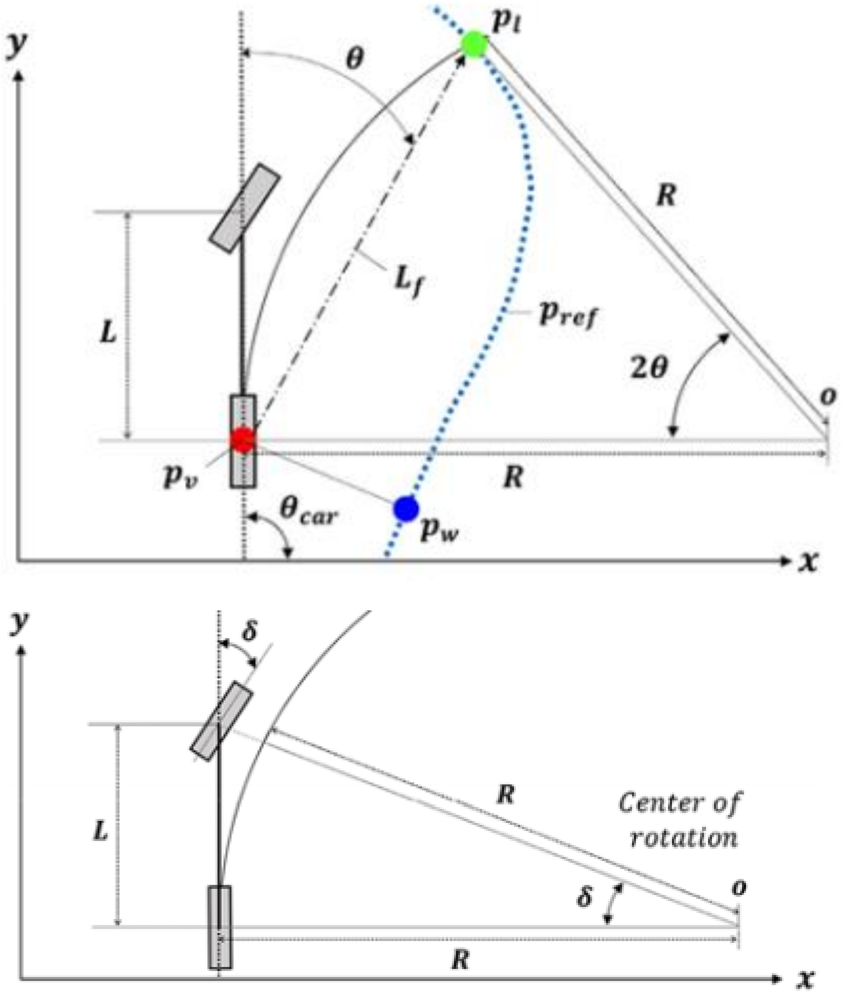
$$\tan(\delta) = \frac{L}{R}$$

$$\delta = \tan^{-1}(\gamma L)$$

$$\delta = \tan^{-1}\left(\frac{2\sin(\theta)}{L_f}\right)$$

Steering angle command is calculated!

And this is unrelated to $v$ value

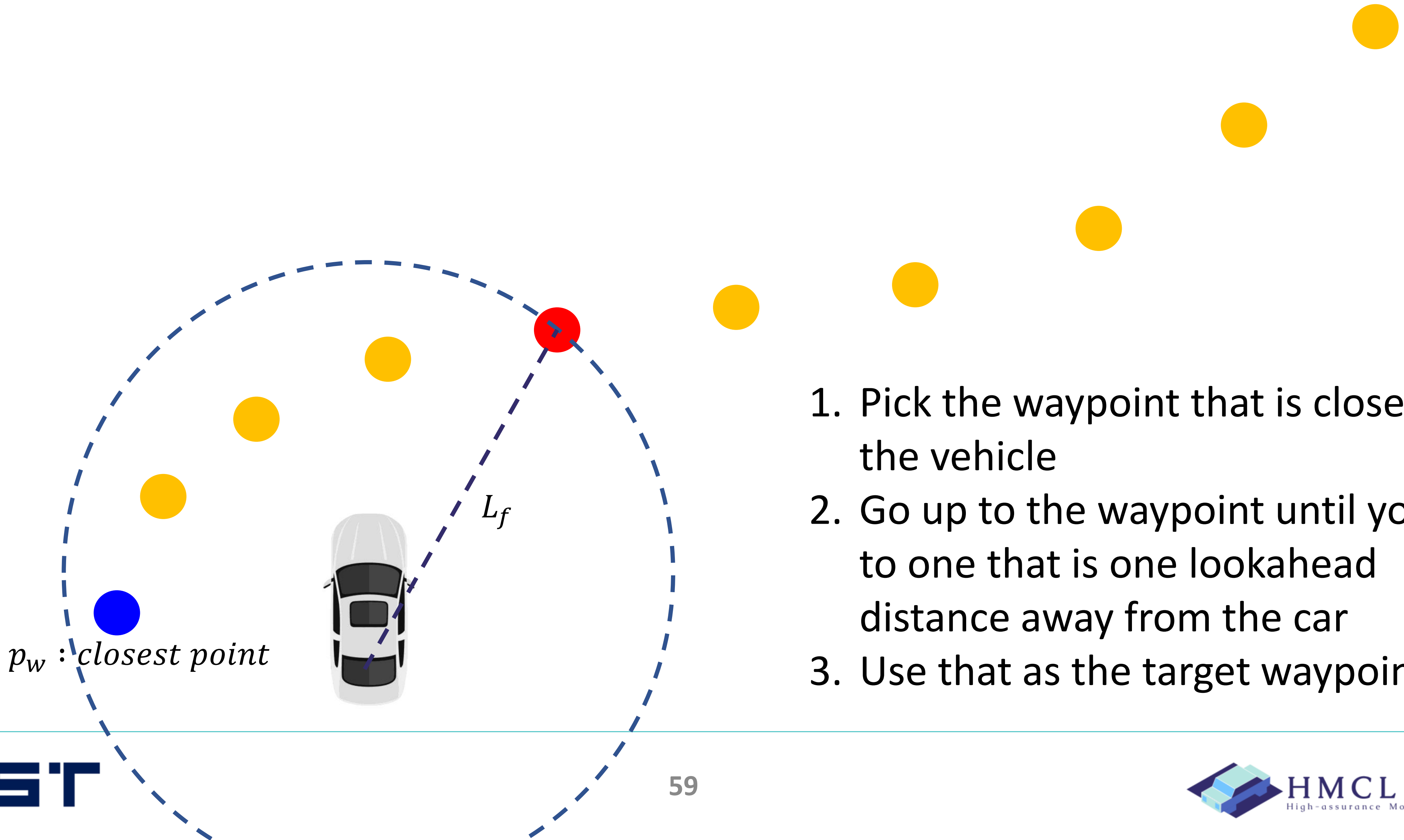



Bicycle steering geometry

# Picking a goal point

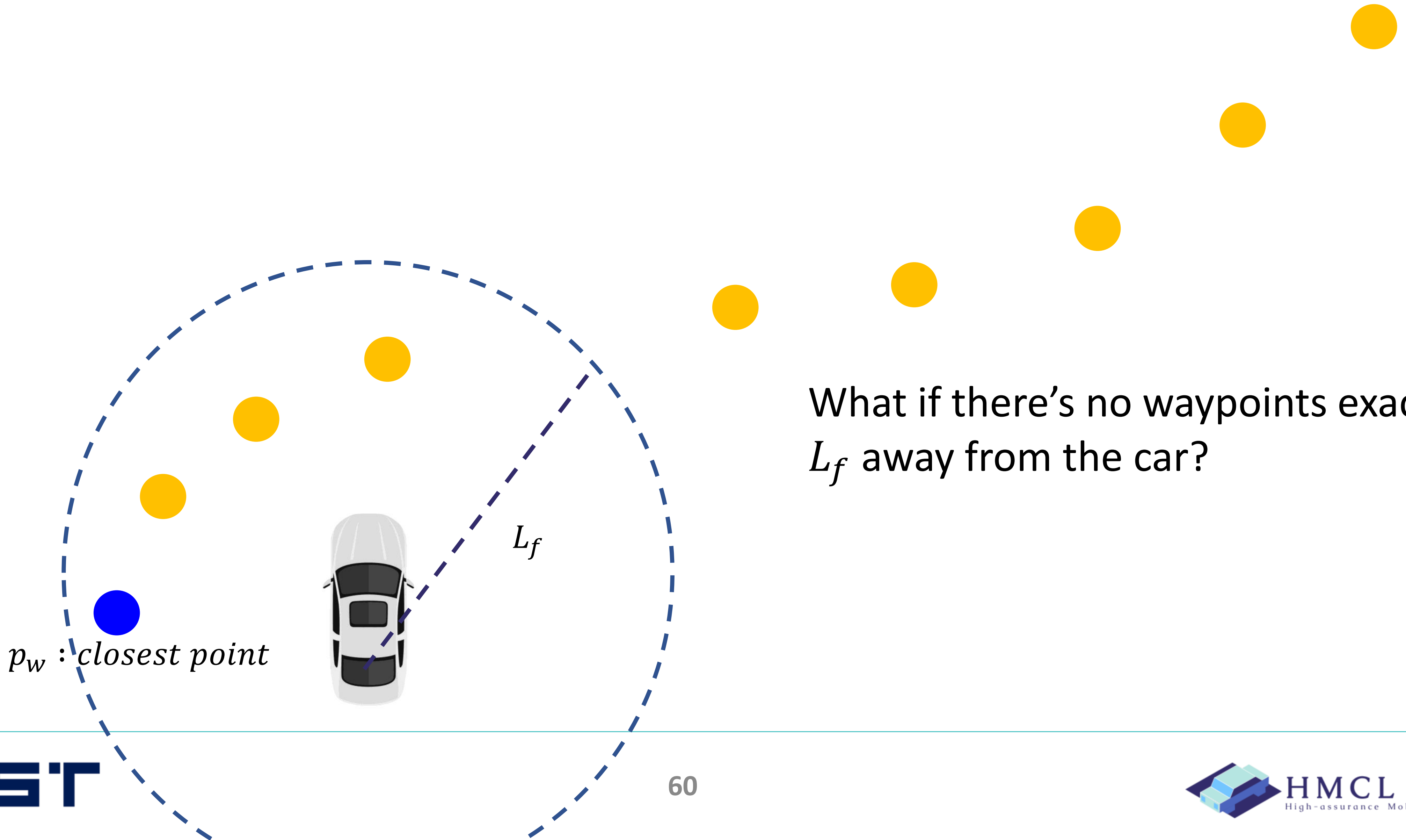
- Now that we know how to calculate steering angle command to a given waypoint, how do we pick a current waypoint from a list of waypoints?

# Picking a goal point



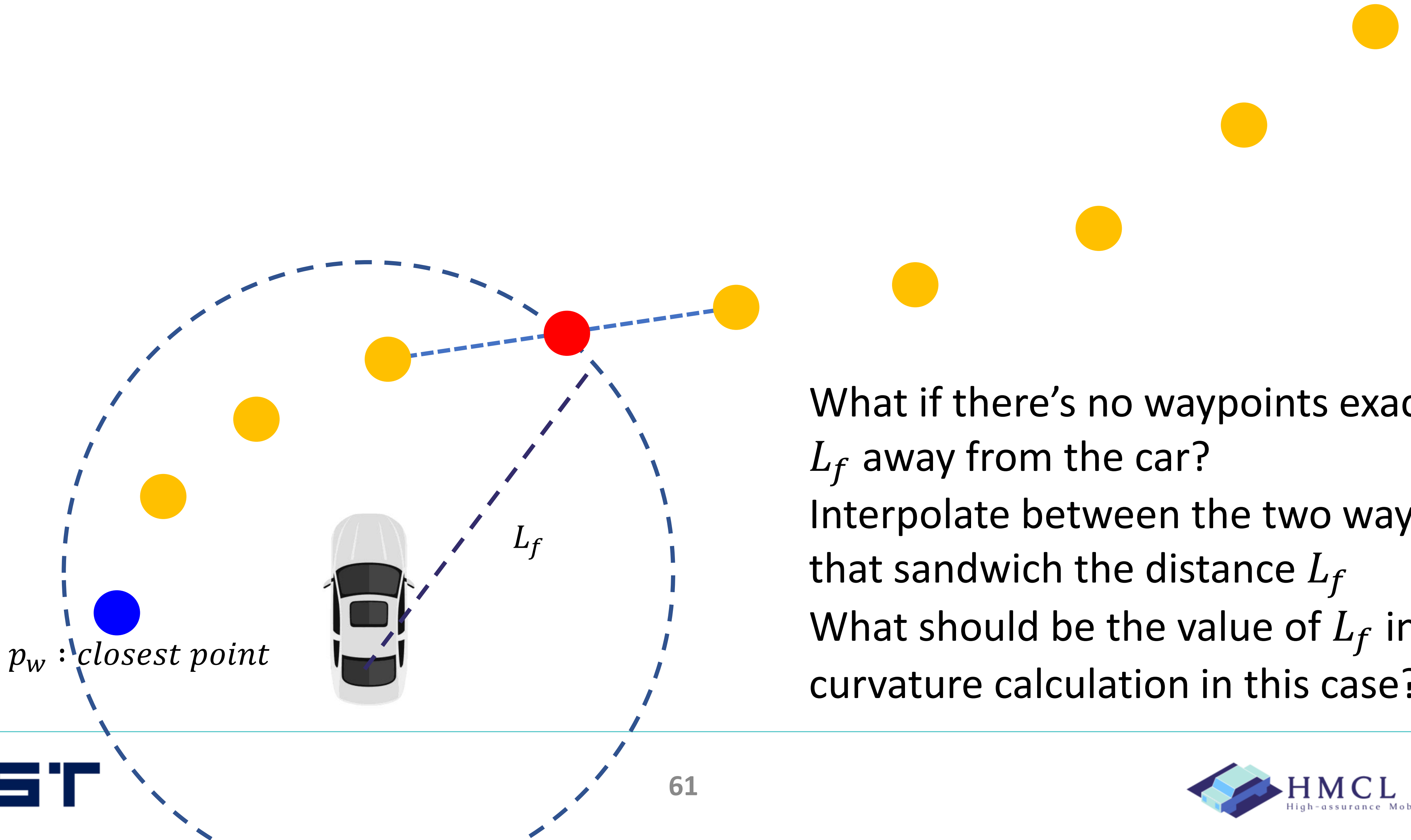

1. Pick the waypoint that is closest to the vehicle
2. Go up to the waypoint until you get to one that is one lookahead distance away from the car
3. Use that as the target waypoint

# Picking a goal point




What if there's no waypoints exactly $L_f$ away from the car?

# Picking a goal point

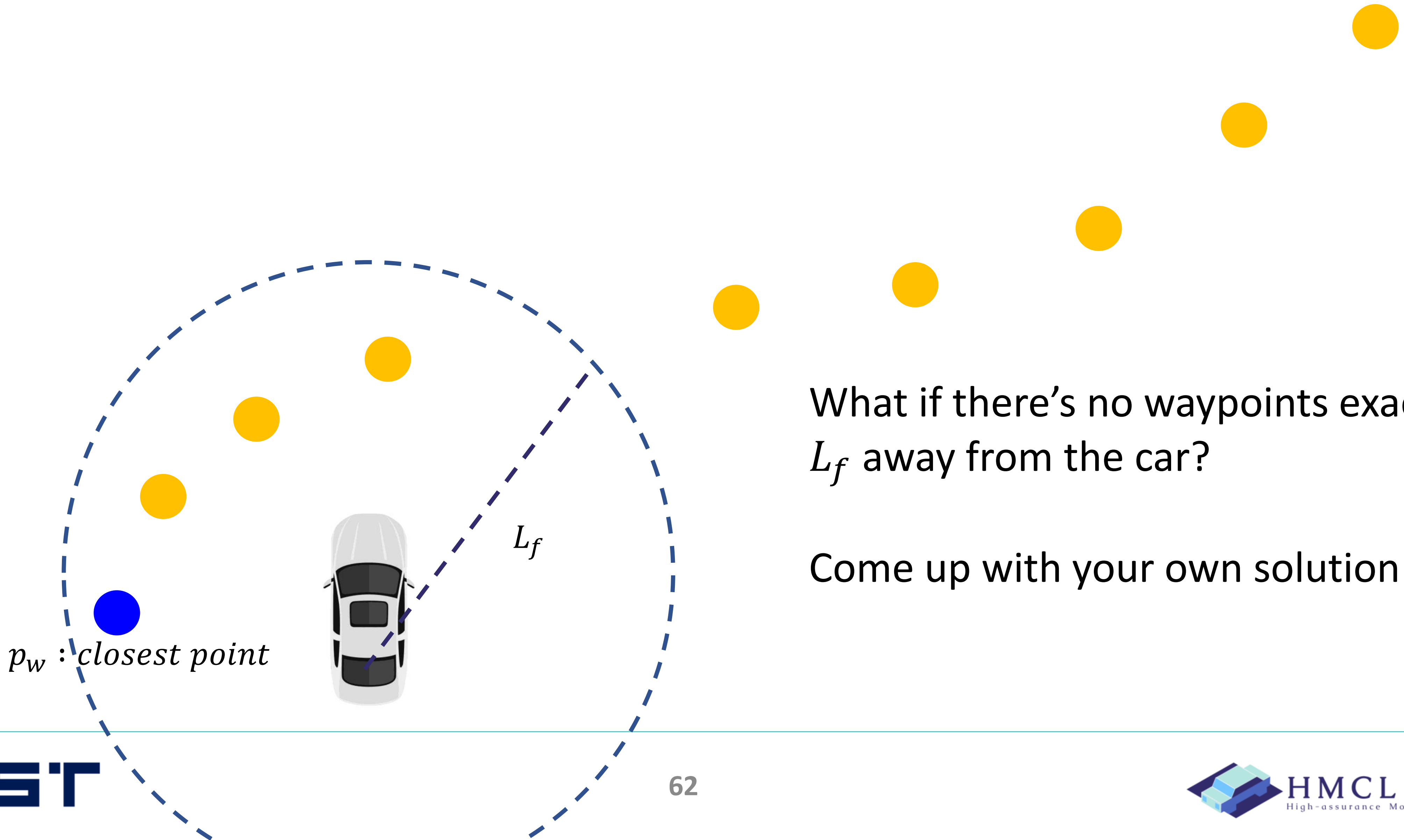



What if there's no waypoints exactly $L_f$ away from the car?

Interpolate between the two waypoints that sandwich the distance $L_f$

What should be the value of $L_f$ in your curvature calculation in this case?

# Picking a goal point



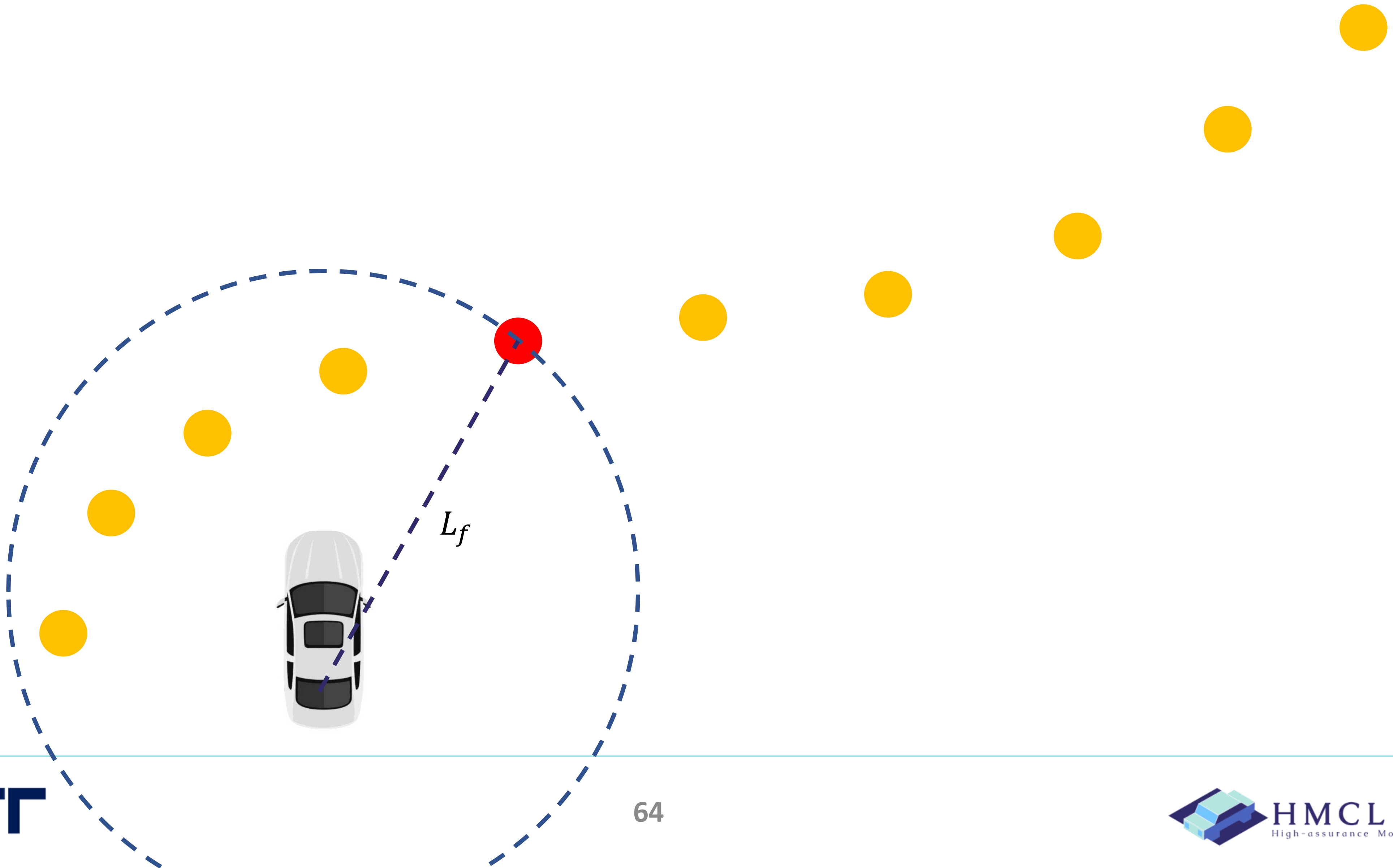

What if there's no waypoints exactly $L_f$ away from the car?

Come up with your own solution!

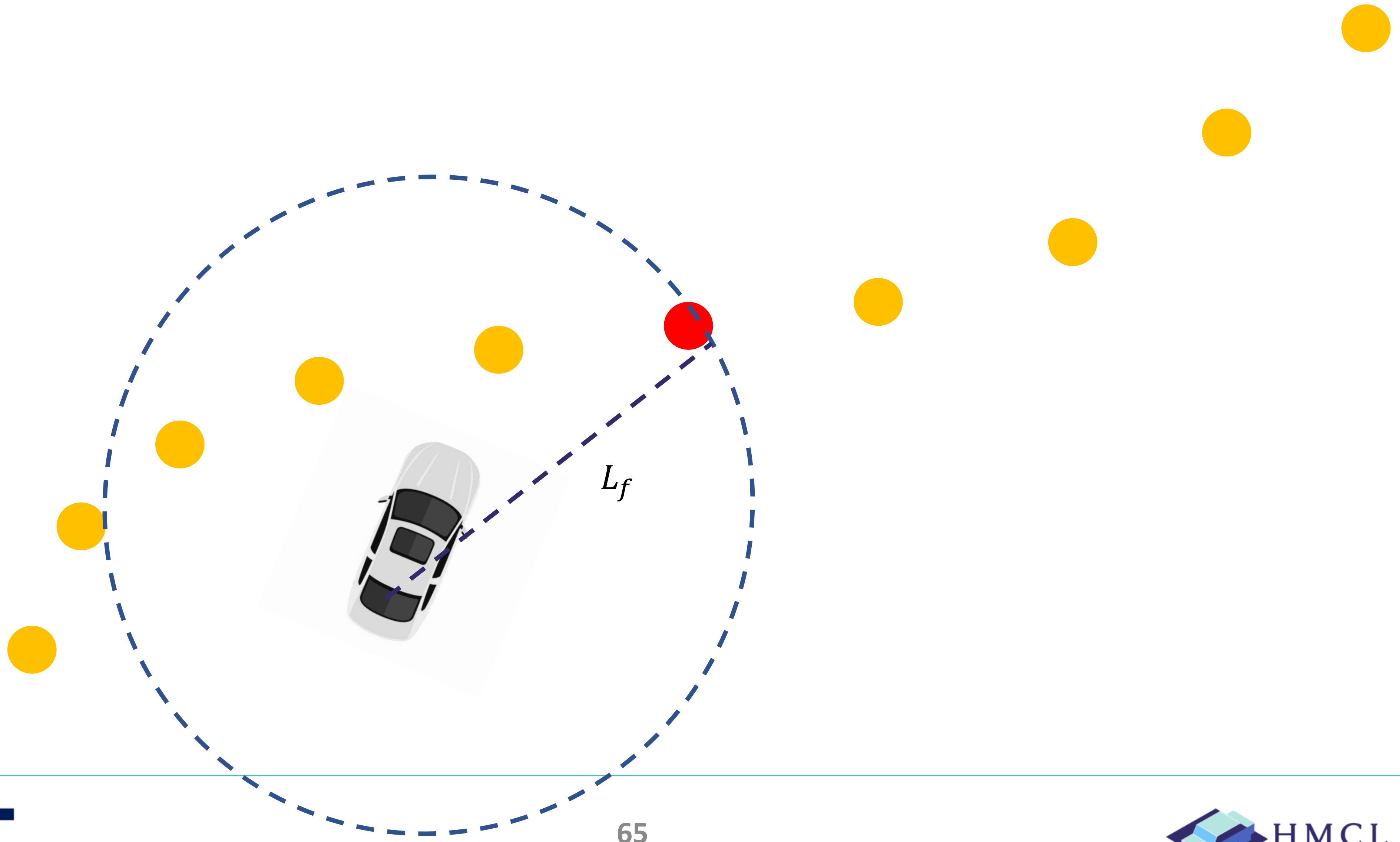
# Updating the goal point (one way to do it)

- Each time we have a new pose of the car, we could :
  - Find the target waypoint
  - Actuate towards that waypoint with calculated steering angle
  - Localize to find the new pose, repeat

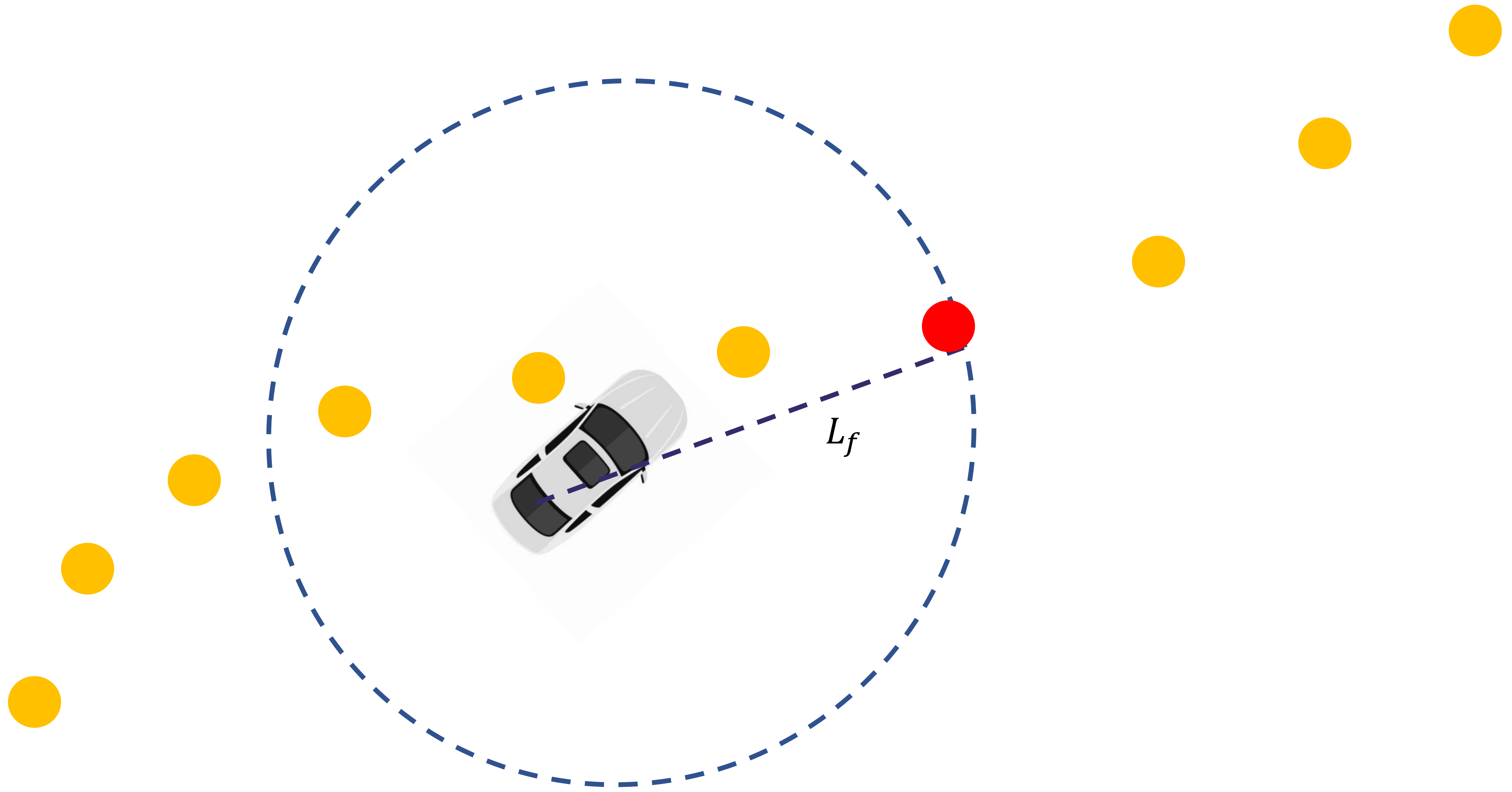
# Updating the goal point



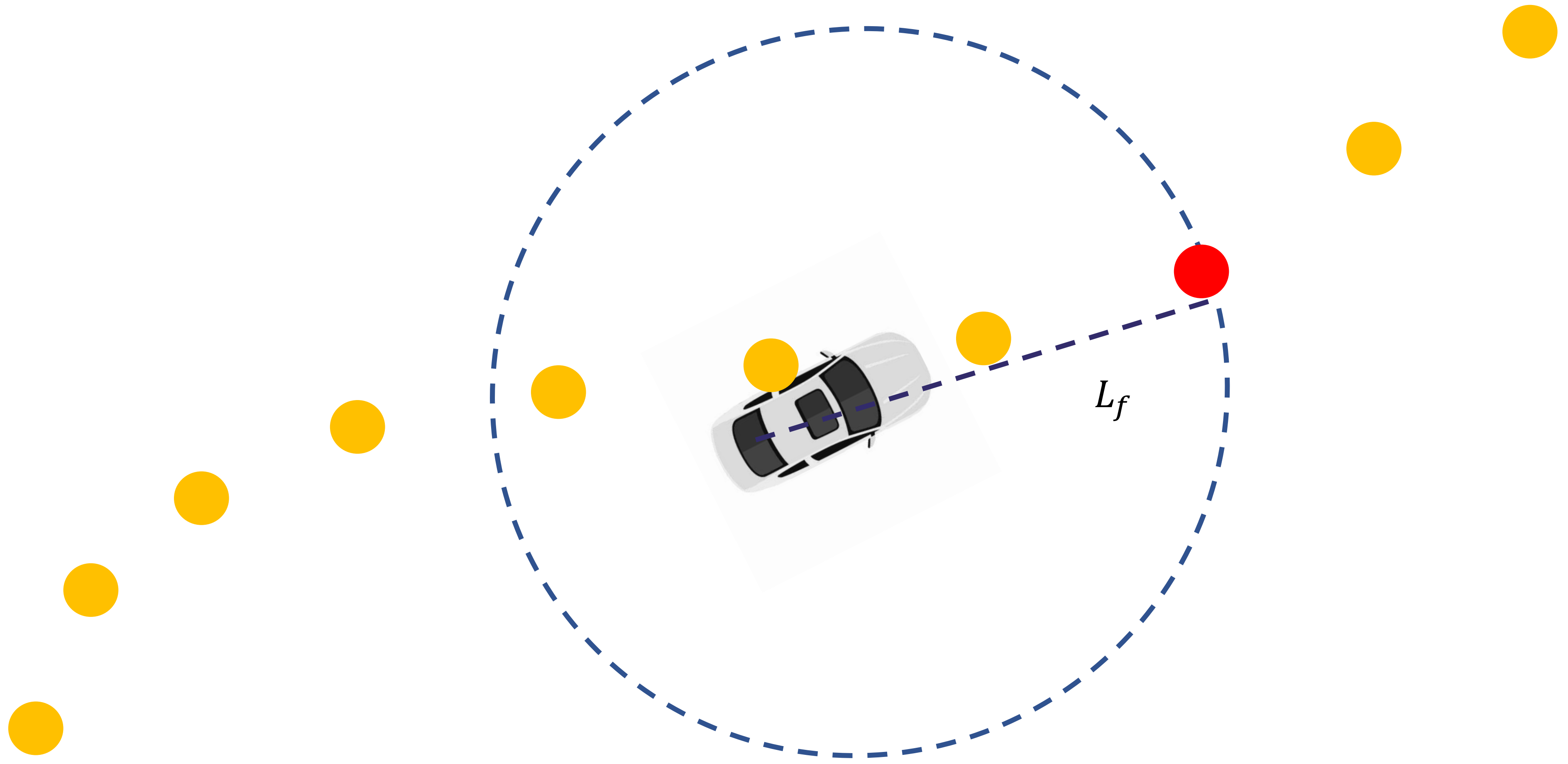
# Updating the goal point



# Updating the goal point

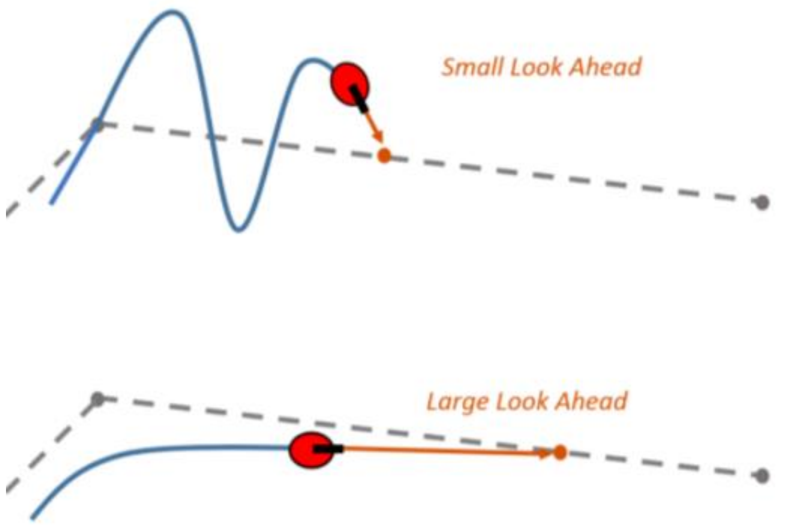


# Updating the goal point



# Tuning

- The parameter $L_f$ (lookahead distance) is a parameter of pure pursuit
- Smaller $L_f$ leads to more aggressive maneuvering to track tighter arc, and and the tighter arcs might be against dynamical limits of the car
- Larger $L_f$ leads to smoother trajectory but larger tracking errors, might lead to close calls with obstacles



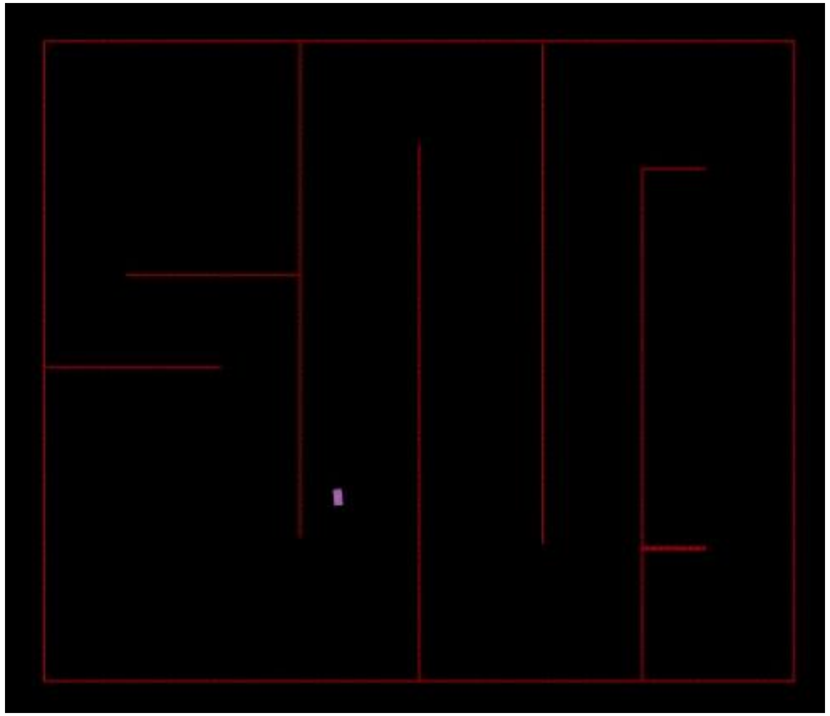
# Notes

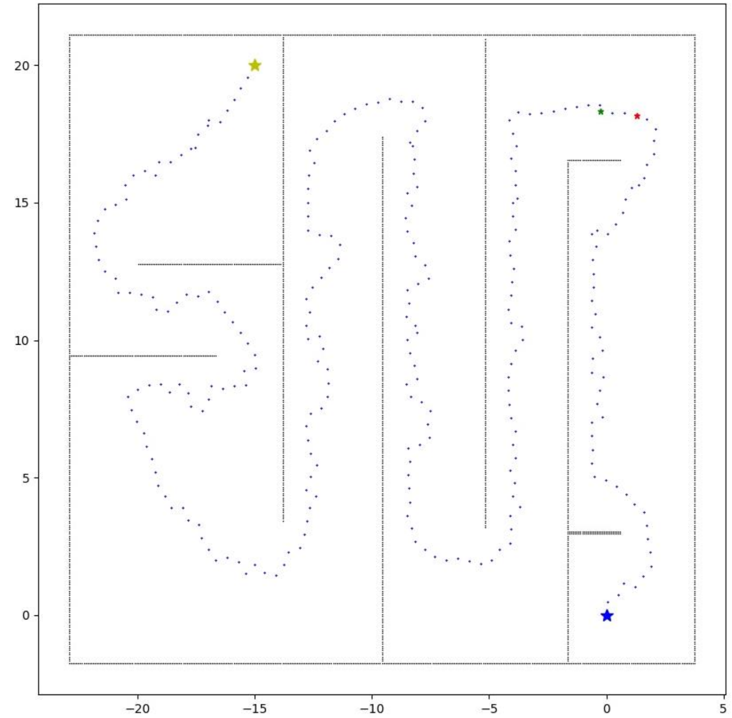
- Tuning $L_f$ will change the behavior of pure pursuit the most
- The waypoints are a sequence of points, and could also have a velocity component at positions
- Pure pursuit doesn't take dynamics into account, thus it might produce dynamically infeasible arcs

# The pipeline of using pure pursuit

- Create a new map using SLAM algorithm
- Create a list of waypoints using a global / local planner
  - a. Easiest way is to record waypoints driven by teleoperation
- Pick waypoints to track at each frame
- Set steering angle to track the target waypoint
- Update the waypoint to track as you go

# Practice Goal






- The goal of this lab is to implement the RRT algorithm.
- In this lab, we will implement the planner which can generate reference trajectory and tune the pure pursuit controller to follow the trajectory.

# Overview

- For this lab, we will implement a ‘**RRTPlanner**’ that makes the reference trajectory in the simulation environment.
- We need to **implement RRT algorithm** to generate a reference trajectory.
- Then, we need to **tune the Pure Pursuit controller** to follow the reference trajectory.

# Racecar simulation environment

# How to start simulation environment

- **Step 1: Clone the repository into your workspace**
  - mkdir : command for making folder
  - cd : command for changing directory
  - git clone: command for cloning a local git repository

```
hmcl1@hmcl1-System-Product-Name:~$ mkdir -p ~/simulation/src
hmcl1@hmcl1-System-Product-Name:~$ cd ~/simulation/src
hmcl1@hmcl1-System-Product-Name:~/simulation/src$ git clone https://github.com/HMCL-UNIST/Bootcamp.git
Cloning into 'MEN491_2023'...
remote: Enumerating objects: 262, done.
remote: Counting objects: 100% (262/262), done.
remote: Compressing objects: 100% (214/214), done.
remote: Total 262 (delta 76), reused 191 (delta 35), pack-reused 0
Receiving objects: 100% (262/262), 3.13 MiB | 6.40 MiB/s, done.
Resolving deltas: 100% (76/76), done.
```

# How to start simulation environment

- **Step 2: Install required packages**
  - git checkout : command for switching branches
  - pip install : command for installing python module

Step 2-1
```
sudo apt-get install python3-pip
```
Step 2-2
```
pip3 install --upgrade --ignore-installed pip setuptools
```
Step 2-3
```
sudo apt-get install unixodbc-dev
```
Step 2-4
```
pip3 install pyodbc
```
Step 2-5
```
pip3 install llvmlite==0.34.0
```
Step 2-6
```
pip3 install pyglet==1.5.26
```

# How to start simulation environment

- **Step 3-1: change setup.py located in** (~/f1tenth-riders-quicksatrt/gym)

Open | setup.py ~/simulation/src/MEN491_2023/f1tenth-riders-quickstart/gym

```python
from setuptools import setup

setup(name='f110_gym',
      version='0.2',
      author='Hongrui Zheng',
      author_email='billyzheng.bz@gmail.com',
      url='https://f1tenth.org',
      install_requires=[
            'gym==0.18.0',
            'numpy', 'Pillow', 'scipy', 'numba', 'pyyaml']
      )
```

- **Step 3-2: Install gym**

```
hmcl1@hmcl1-System-Product-Name:~/simulation/src$ cd ~/simulation/src/MEN491_2023/f1tenth-riders-quickstart/
hmcl1@hmcl1-System-Product-Name:~/simulation/src/MEN491_2023/f1tenth-riders-quickstart$ pip3 install --user  -e gym
```

# How to start simulation environment

- **If you have a problem,**

```
CMake Error at /usr/local/share/cmake-3.9/Modules/FindPackageHandleStandardArgs
.cmake:137 (message):
  Could NOT find SDL (missing: SDL_LIBRARY SDL_INCLUDE_DIR)
Call Stack (most recent call first):
```

-> sudo apt-get install libsdl-image1.2-dev

-> sudo apt-get install libsdl-dev

- **If you have until a problem,**

-> rosdep install –y –from-paths src –ignore-src –rosdistro $ROS_DISTRP –os=ubuntu:xenial

# Gym Environment

- **Change main.py** (~/f1tenth-riders-quicksatrt/pkg/src/pkg)

```python
# import your drivers here
from drivers import SimpleDriver

# choose your drivers here (1-4)
drivers = [SimpleDriver()]

# choose your racetrack here (SILVERSTONE, SILVERSTONE_OBS)
RACETRACK = 'SILVERSTONE'

```

**Changing driver**

- For importing your drivers, write the class name which you want to import.
- Possible classes are declared in **drivers.py**.

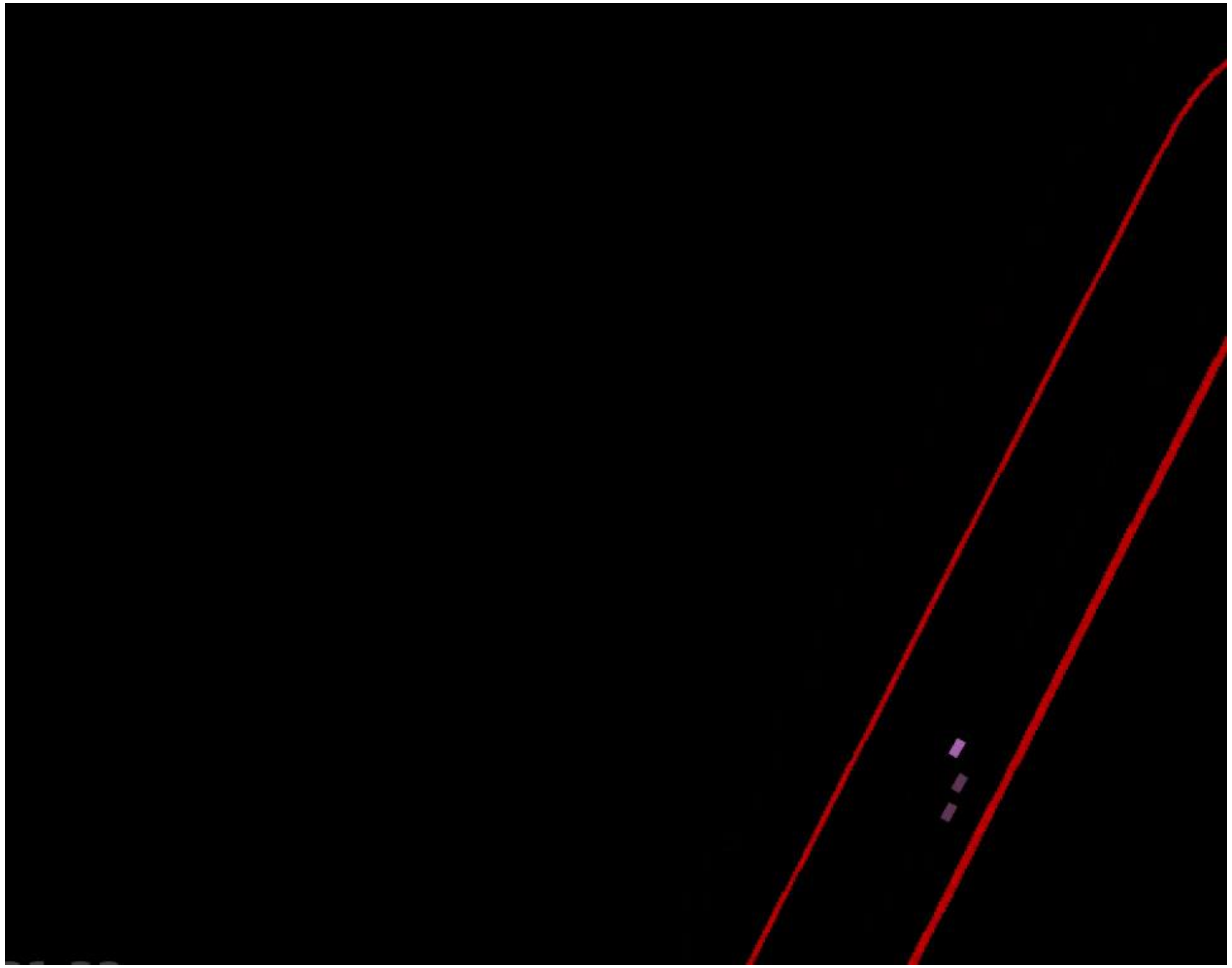
**Multi-Agent Racing**

- For spawning your drivers, write the class name which you want to import
- To practice racing multiple drivers against each other, simply choose multiple drivers
- You may race the same driver against itself by choosing it twice
- Four driver are available to add for simulation

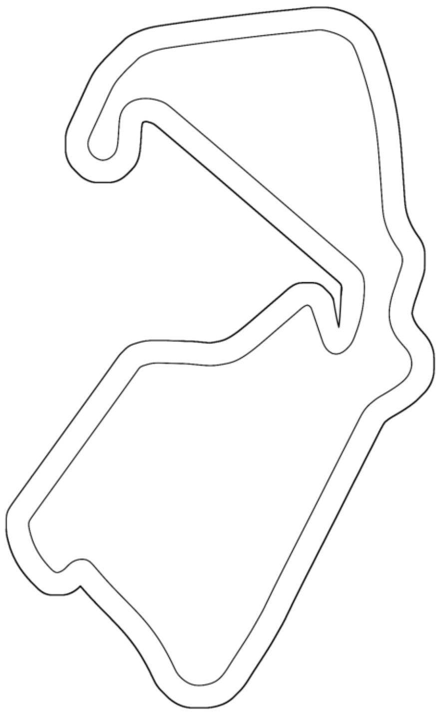
**Changing map**

- You can choose between using the ordinary Silverstone map or the Silverstone Obstacle map

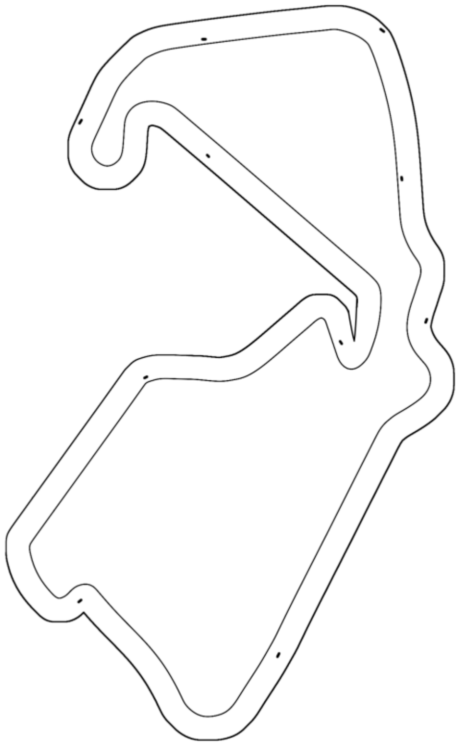
# Multi-Agent Racing



# Changing Map



Silverstone map



Silverstone Obstacle map

# Structure of a Driver

- **Let's see drivers.py** (~/f1tenth-riders-quicksatrt/pkg/src/pkg)

```python
import numpy as np

# drives straight ahead at a speed of 5
class SimpleDriver:

    def process_lidar(self, ranges):
        speed = 5.0
        steering_angle = 0.0
        return speed, steering_angle


# drives toward the furthest point it sees
class AnotherDriver:

    def process_lidar(self, ranges):
        # the number of LiDAR points
        NUM_RANGES = len(ranges)
        # angle between each LiDAR point
        ANGLE_BETWEEN = 2 * np.pi / NUM_RANGES
        # number of points in each quadrant
        NUM_PER_QUADRANT = NUM_RANGES // 4

        # the index of the furthest LiDAR point (ignoring the points behind the car)
        max_idx = np.argmax(ranges[NUM_PER_QUADRANT:-NUM_PER_QUADRANT]) + NUM_PER_QUADRANT
        # some math to get the steering angle to correspond to the chosen LiDAR point
        steering_angle = max_idx * ANGLE_BETWEEN - (NUM_RANGES // 2) * ANGLE_BETWEEN
        speed = 5.0

        return speed, steering_angle
```

- There are two types of drivers
- You can add your code in this file and import it by main.py
- In simulation, output value is **speed** and **steering angle**

# Structure of a Driver

- Let's see **drivers.py**

```python
import numpy as np

# drives straight ahead at a speed of 5
class SimpleDriver:

    def process_lidar(self, ranges):
        speed = 5.0
        steering_angle = 0.0
        return speed, steering_angle


# drives toward the furthest point it sees
class AnotherDriver:

    def process_lidar(self, ranges):
        # the number of LiDAR points
        NUM_RANGES = len(ranges)
        # angle between each LiDAR point
        ANGLE_BETWEEN = 2 * np.pi / NUM_RANGES
        # number of points in each quadrant
        NUM_PER_QUADRANT = NUM_RANGES // 4

        # the index of the furthest LiDAR point (ignoring the points behind the car)
        max_idx = np.argmax(ranges[NUM_PER_QUADRANT:-NUM_PER_QUADRANT]) + NUM_PER_QUADRANT
        # some math to get the steering angle to correspond to the chosen LiDAR point
        steering_angle = max_idx * ANGLE_BETWEEN - (NUM_RANGES // 2) * ANGLE_BETWEEN
        speed = 5.0

        return speed, steering_angle
```

- Simple Driver:

Without any control, just move with constant velocity and steering angle

# Structure of a Driver

- Let's see **drivers.py**

```
import numpy as np

# drives straight ahead at a speed of 5
class SimpleDriver:

    def process_lidar(self, ranges):
        speed = 5.0
        steering_angle = 0.0
        return speed, steering_angle


# drives toward the furthest point it sees
class AnotherDriver:

    def process_lidar(self, ranges):
        # the number of LiDAR points
        NUM_RANGES = len(ranges)
        # angle between each LiDAR point
        ANGLE_BETWEEN = 2 * np.pi / NUM_RANGES
        # number of points in each quadrant
        NUM_PER_QUADRANT = NUM_RANGES // 4

        # the index of the furthest LiDAR point (ignoring the points behind the car)
        max_idx = np.argmax(ranges[NUM_PER_QUADRANT:-NUM_PER_QUADRANT]) + NUM_PER_QUADRANT
        # some math to get the steering angle to correspond to the chosen LiDAR point
        steering_angle = max_idx * ANGLE_BETWEEN - (NUM_RANGES // 2) * ANGLE_BETWEEN
        speed = 5.0

        return speed, steering_angle
```

- AnotherDriver:

With constant velocity, change the steer angle toward the furthest point it sees

# Structure of a Driver

- A Driver is just a class that has a *process_lidar* function which takes in LiDAR data and returns speed to drive at along with a steering angle.

- Ranges: an array of 1080 distances(ranges) detected by the LiDAR scanner. As the LiDAR scanner takes reading for the full $360°$, the angle between each range is $2\pi/1080$ (in radians)

- steering_angle: an angle in the range $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$ in radians, with $0°$ meaning straight ahead

# How to start simulation environment with your driver

- cd **YOURWORKSPACE**/f1tenth-riders-quicksatrt/pkg/src/pkg
- python3 main.py

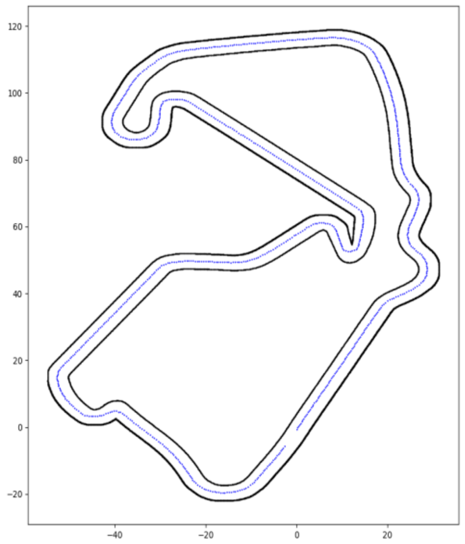
```
hmcl1@hmcl1-System-Product-Name:~/simulation/src/MEN491_2023/f1tenth-riders-quickstart$ cd pkg/src/pkg/
hmcl1@hmcl1-System-Product-Name:~/simulation/src/MEN491_2023/f1tenth-riders-quickstart/pkg/src/pkg$ code .
hmcl1@hmcl1-System-Product-Name:~/simulation/src/MEN491_2023/f1tenth-riders-quickstart/pkg/src/pkg$ python3 main.py
/home/hmcl1/.local/lib/python3.6/site-packages/numba/core/errors.py:149: UserWarning: Insufficiently recent colorama version found. Numba requires colorama >= 0.3.9
  warnings.warn(msg)
Sim elapsed time: 11.059999999999809 Real elapsed time: 7.173454284667969
```

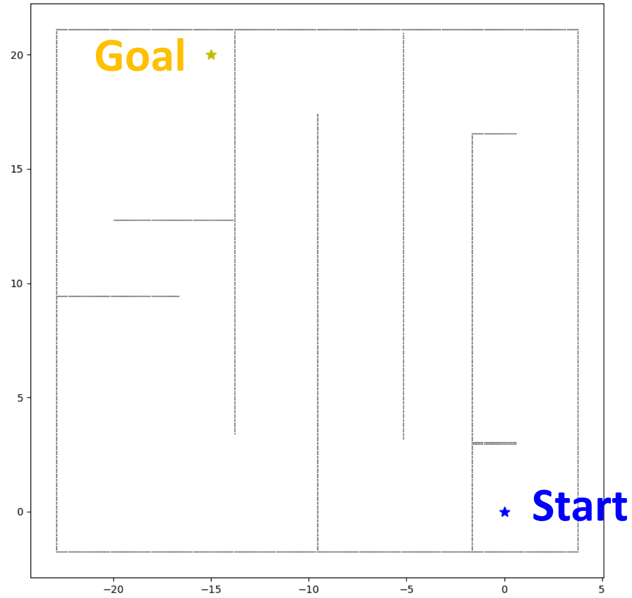
# Environment Setup

- In the file, you can find six files,
  - SIMPLE.png
  - SIMPLE.yaml
  - map_rrt.npy
  - test_rrt.py, planner_rrt.py
  - drivers_ppc_rrt.py

# SIMPLE map

- We will generate the reference trajectory for “SIMPLE” map using RRT algorithm.








UNIST

HMCL
High-assurance Mobility Control Lab

# planner_rrt.py

- planner_rrt.py contains the skeleton code for your RRT planner.
- You should generate the reference trajectory based on RRT algorithm.
- You can implement RRT algorithm based on the lecture notes

```python
############################################################
#############  MEN491 Creating Autonomous Car #############
#############          RRT ASSIGNMENT         #############
#############     20201234   Hyeonbin Lee     #############
############################################################

# ASSIGNMENT - Implement the planner using RRT algorithm

import numpy as np
import math

class Node:

    def __init__(self, n):
        self.x = n[0]
        self.y = n[1]
        self.parent = None

class RRTPlanner:

    def __init__(self, s_start, s_goal, map_points):

        #################################################
        ############# DO NOT TOUCH THIS CODE#############
        #################################################
        self.s_start = Node(s_start)
        self.s_goal = Node(s_goal)
        self.x_range = (-23.8, 2.2)
        self.y_range = (-3.0, 19.0)
        self.map = map_points
        #################################################
        #################################################

        # you should set these value appropriately
        self.step_len =
        self.goal_sample_rate =
        self.iter_max =

        # you can add some more class variables under this line
        # self.variable = value
        # like this

    def rrt_planning(self):

        # TODO implement RRT algorithm

        path = np.zeros(shape=(100,2))

        # find path from start pose to goal pose which does not collide with map
        # sample node from the in the range self.x_range, self.y_range

        return path

    # you can define more functions if you want
```

# drivers_ppc_rrt.py

- drivers_ppc_rrt.py contains the skeleton code for your pure pursuit controller.
- Implement pursuit control code.
- Then tune the parameter appropriately to follow your path generated by RRT algorithm.

```python
# ASSIGNMENT - Implement the driver using Pure Pursuit algorithm

import numpy as np
import math

class PurePursuitDriver:

    def pure_pursuit_control(self, pose_x, pose_y, pose_theta, ref):

        # TODO implement pure pursuit algorithm

        # Set the lookahead distance


        # Find the lookahead point on the reference trajectory
        lookahead_idx = 10

        # Compute the heading to the lookahead point


        # Compute the steering angle
        steering_angle = 0.0

        # Compute the speed (you may use constant speed simply)
        speed = 5.0

        return speed, steering_angle, lookahead_idx
```

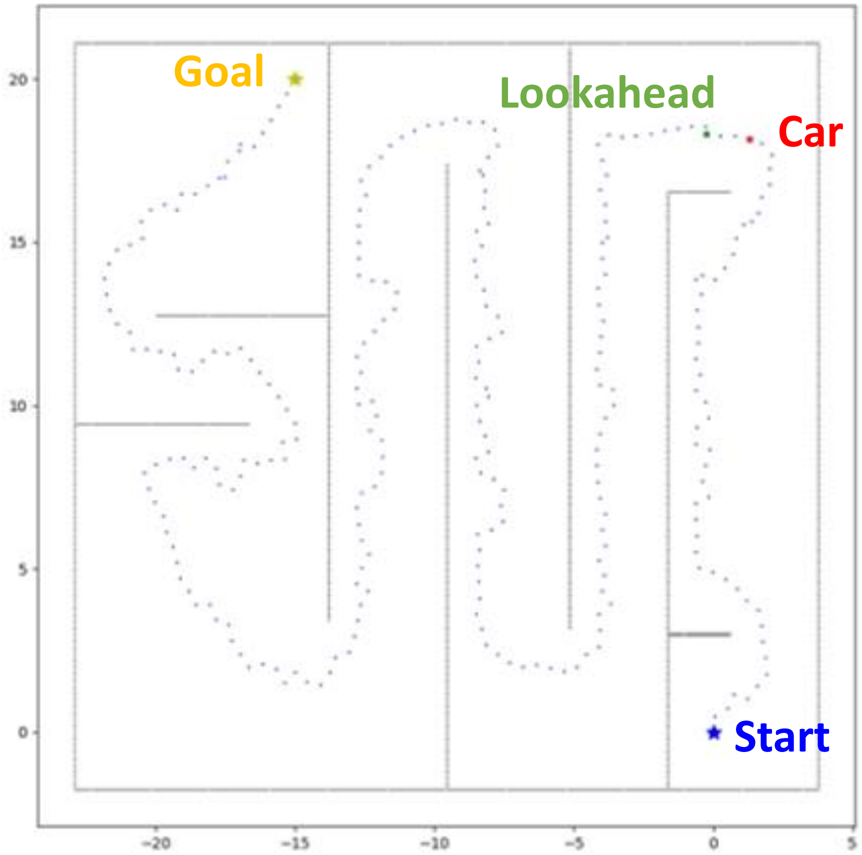
# test_rrt.py

- test_rrt.py contains the python code for testing your RRT planner and Pure Pursuit controller in the simulation environment.
- You can run the test using the command below.

```
(py36) hb@HB-HMCL:~/simulation/src/MEN492_2023/f1tenth-riders-quickstart/pkg/src/pkg$ python test_rrt.py
```

# test_ppc.py - Plot tools

- If you uncomment this code, you can use plot tools.
- You can check the start and goal position marked with blue and yellow point, the ego vehicle position with a red point, lookahead point with green and blue center line in the plot.
- You can debug your code using the plot tools like this.



# test_ppc.py – Grading

- Generate reference path successfully

```
(py36) hb@HB-HMCL:~/simulation/src/MEN492_2023/f1tenth-riders-quickstart/pkg/src/pkg$ python test_rrt.py
Path is generated successfully!
You got 2 points!
Sim elapsed time: 25.5800000000012 Real elapsed time: 165.08719658851624 Final Score : 2
```

→ 2 points (You will lose this point when the path seems to intersect the map and cause collision even if it prints "You got 2 points".)

- Reach the goal using controller based on the generated path

```
(py36) hb@HB-HMCL:~/simulation/src/MEN492_2023/f1tenth-riders-quickstart/pkg/src/pkg$ python test_rrt.py
Path is generated successfully!
You got 2 points!
Your car has successfully reached the goal!
You got 5 points! Congratulations!!
Sim elapsed time: 25.72000000000122 Real elapsed time: 166.23932313919067 Final Score : 5
```

→ 3 points

- If the collision occurs or the sim elapsed time exceeds 30 seconds, the simulation is terminated.

# test_ppc.py – Successful Example