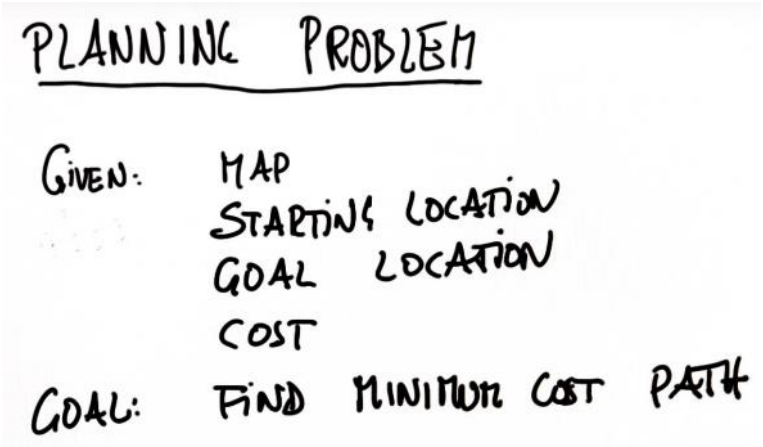
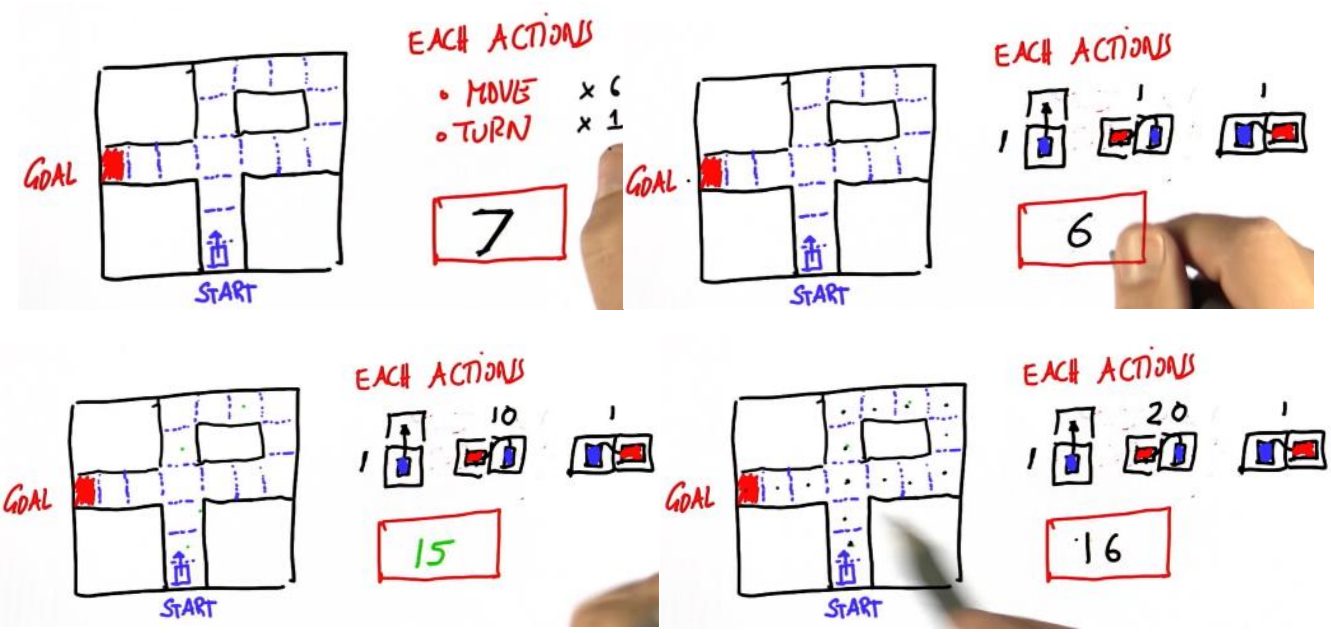
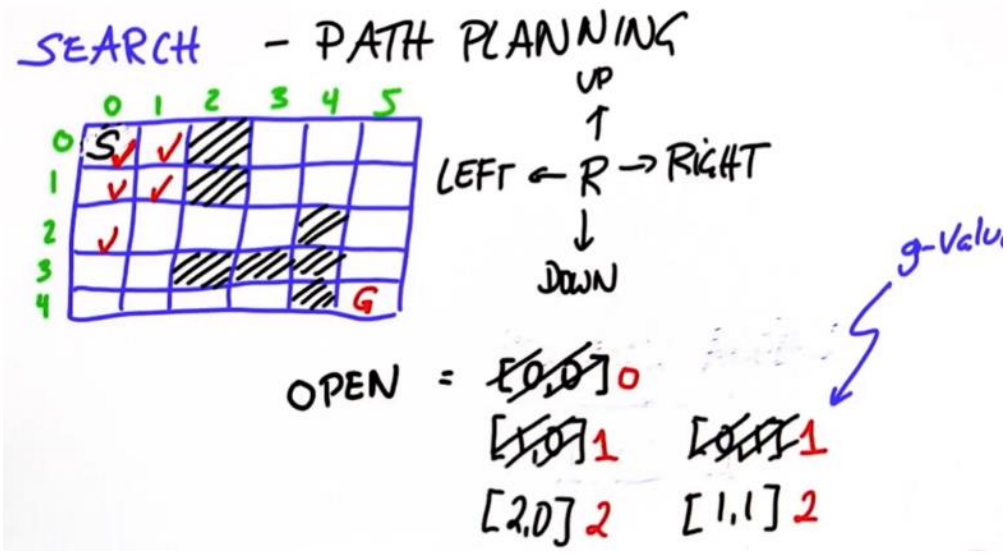




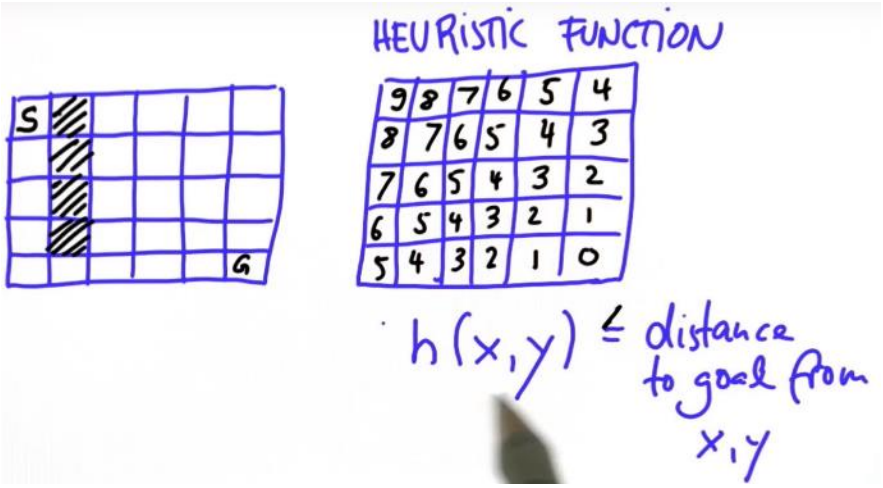
First Search

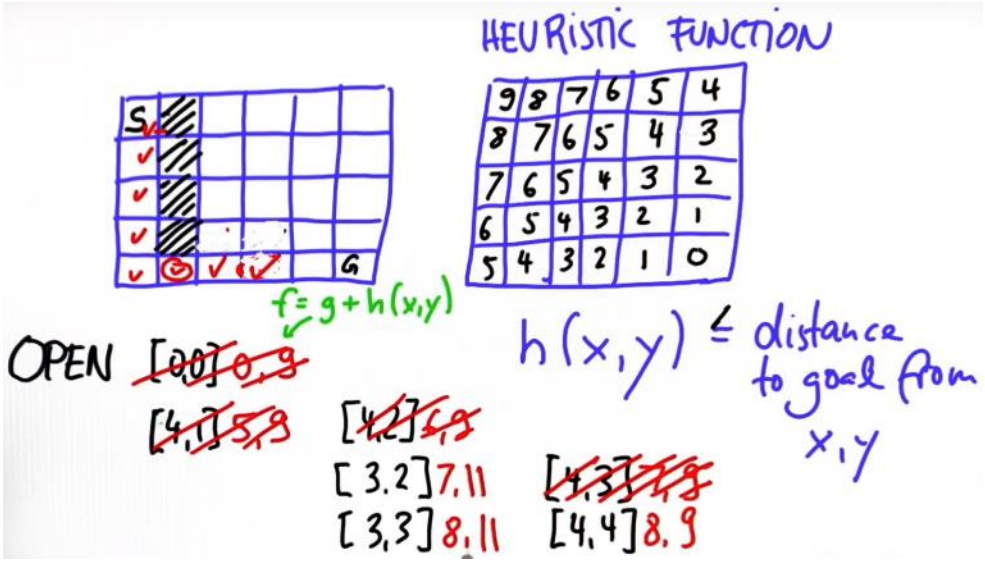


A*

If the heuristic function is null, then the performance is the same as the First Search
Heuristic function should be an optimistic guess

Heuristic function supposes no obstacles:

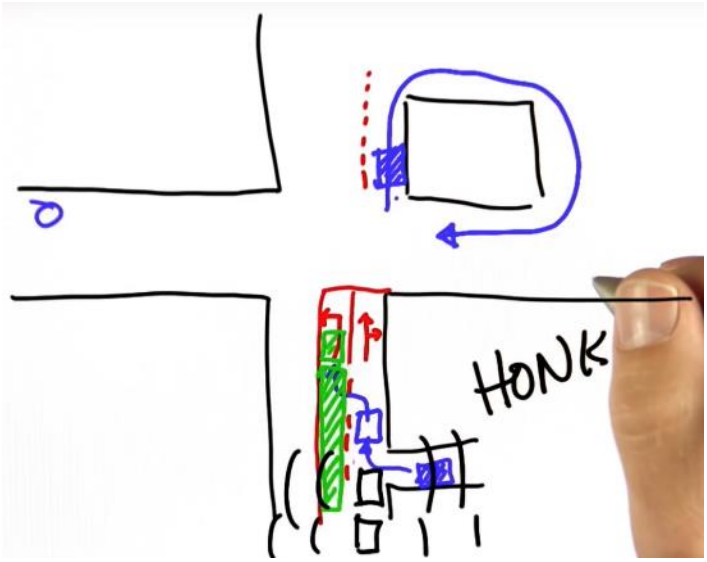


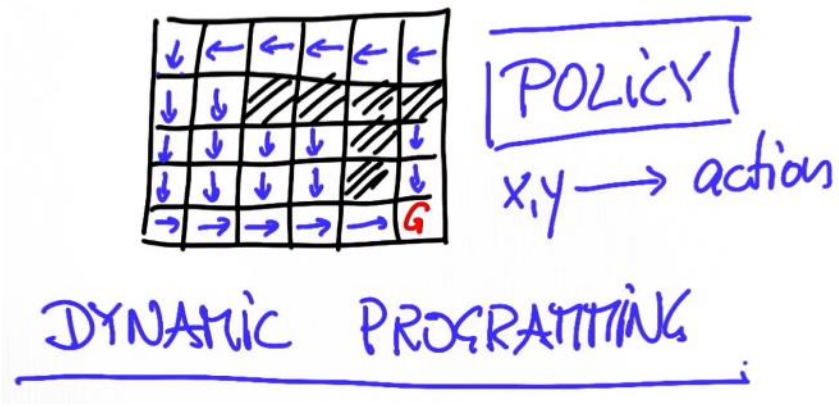


DYNAMIC PROGRAMMING

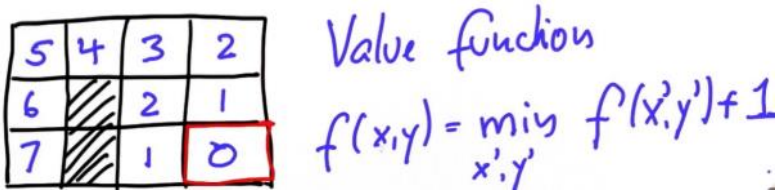
GIVEN
- MAP
- GOAL

OUTPUT: BEST PATH FROM ANYWHERE

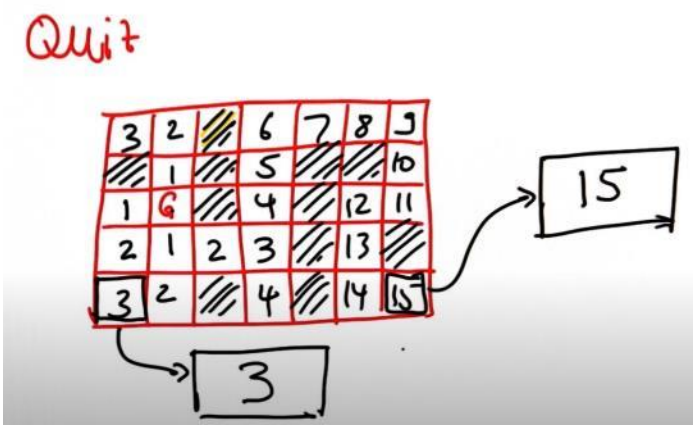


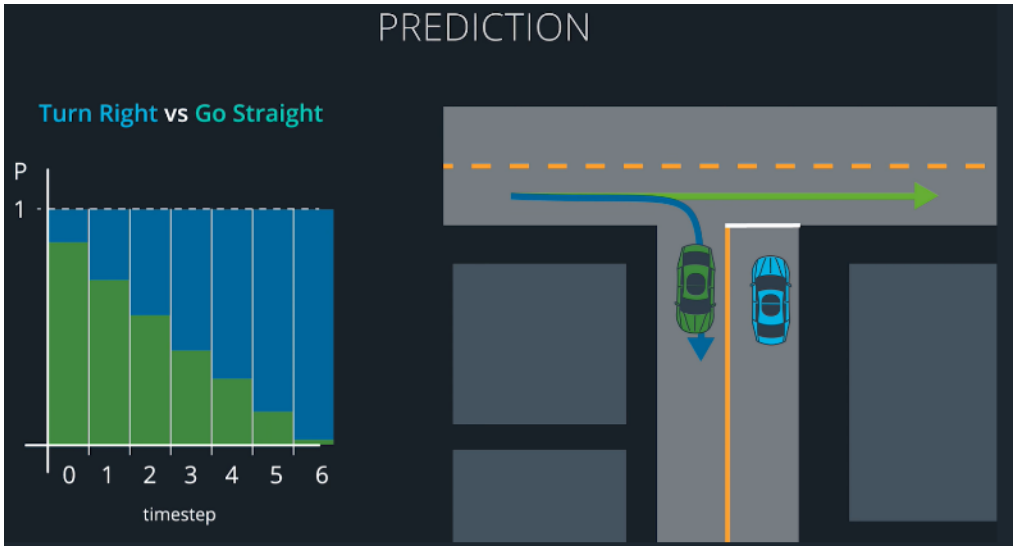


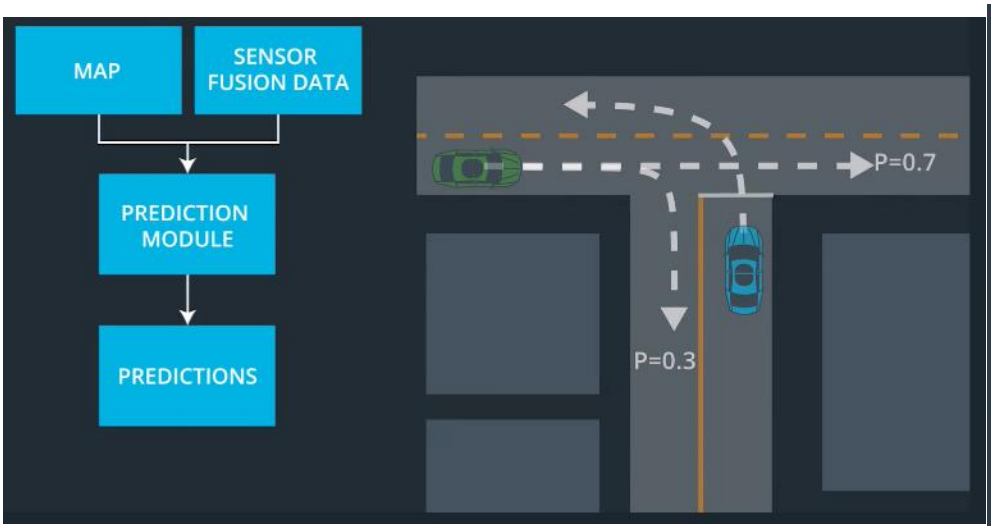
Cost = 1



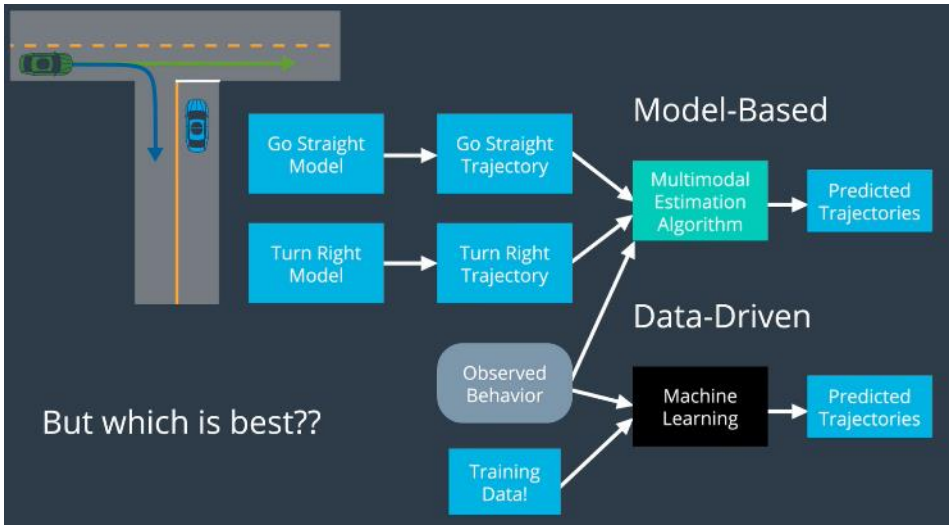
Plan for every location







1. Introduction and Overview
   a. High level Overview
   b. Inputs and Outputs
2. How Prediction is Done
   a. Model-based vs Data-driven Approaches
   b. Data-driven approach example: trajectory clustering
3. Model Based Approaches
   a. Process Models
   b. Multi-modal estimators
4. Hybrid Approaches
   a. Intent Classification -> Trajectory Generation
   b. Naive Bayes



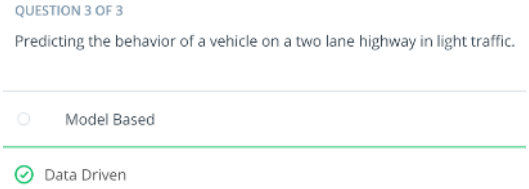
QUESTION 1 OF 3

Determining maximum safe turning speed on a wet road.

- [x] Model Based
- [ ] Data Driven

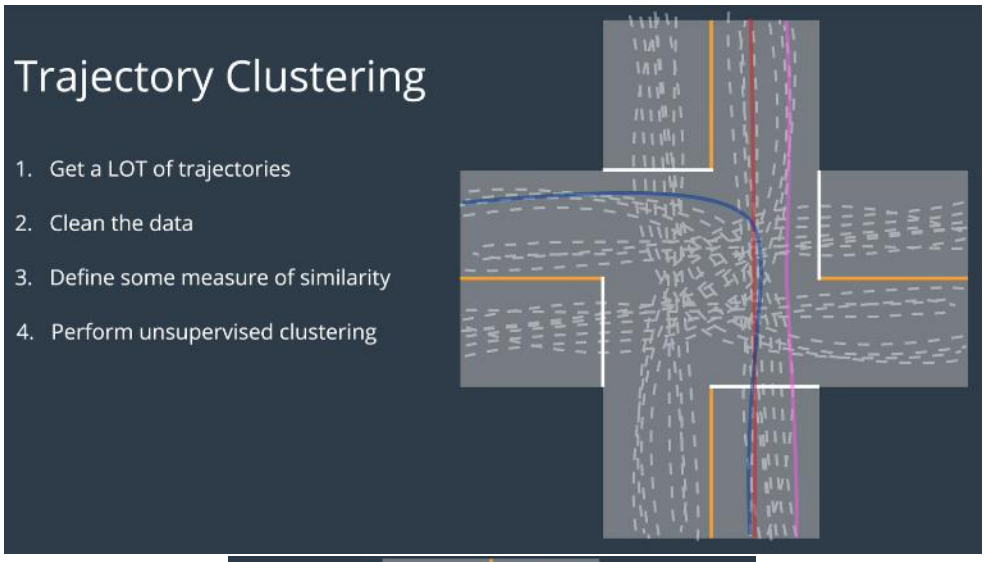
In this situation we could use a model based approach to incorporate our knowledge of physics (friction, forces, etc...) to figure out exactly (or almost exactly) when a vehicle would begin to skid on a wet road.

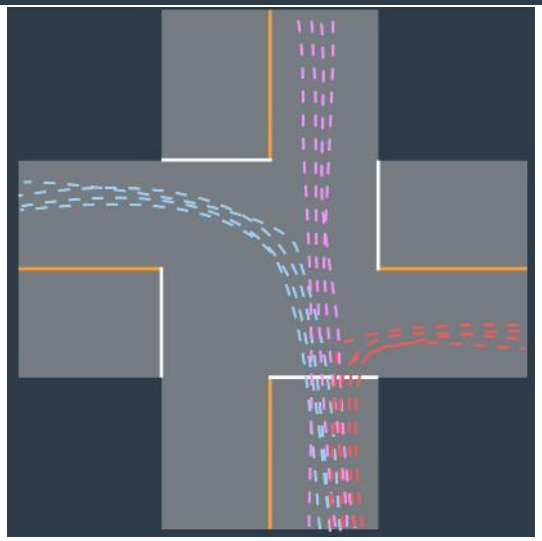
Even with data driven approaches this would still be a very hard problem but since we don't even know what this object is, a model based approach to prediction would be nearly impossible.



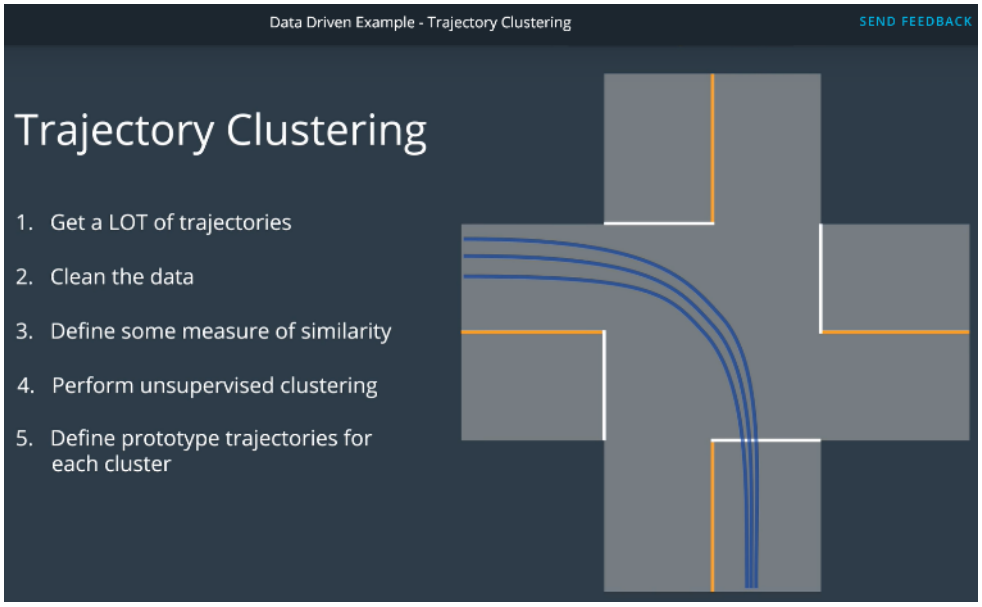
You could really use either approach (or a **hybrid approach**) in this situation.
On the one hand there are very few behaviors we need to model in a highway driving situation and the physics are all very well understood so model based approaches could work.

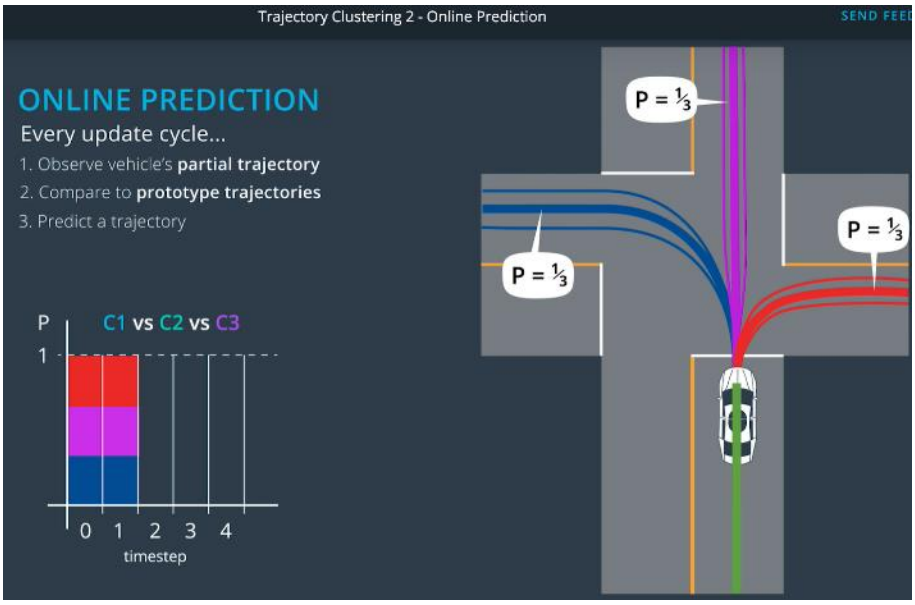
On the other hand it would be relatively easy to collect a **lot** of training data in similar situations so a purely data driven approach could work too.

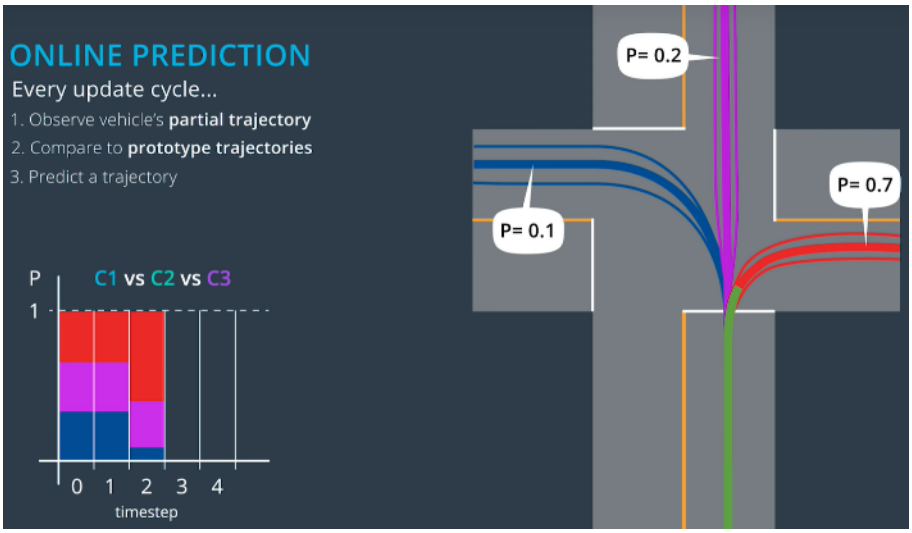


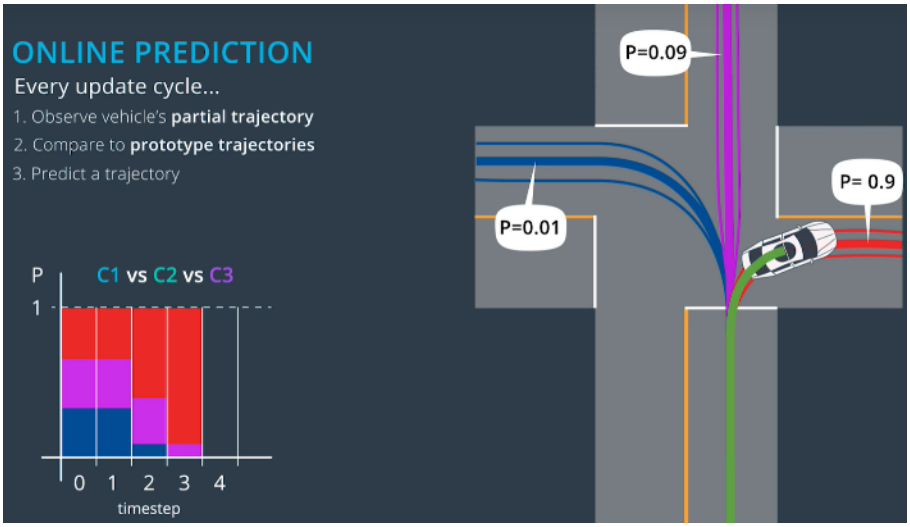


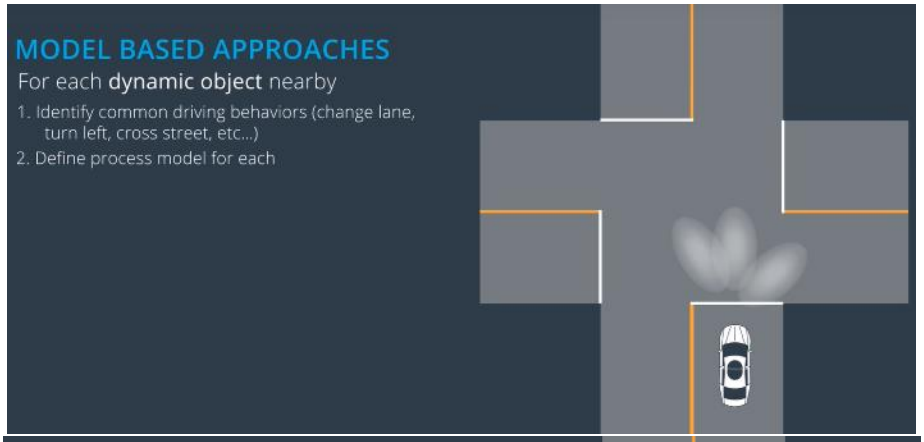
Actually there will be twice as many possibilities are. A set which involves stopping at a intersection, and the other one that goes without stopping

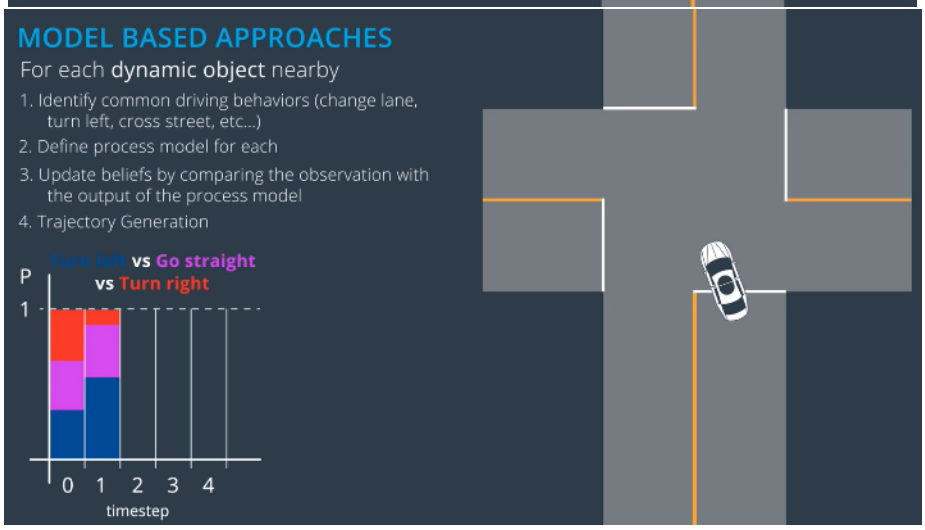

Data Driven Example - Trajectory Clustering
SEND FEEDBACK
Trajectory Clustering
1. Get a LOT of trajectories
2. Clean the data
3. Define some measure of similarity
4. Perform unsupervised clustering
5. Define prototype trajectories for each cluster

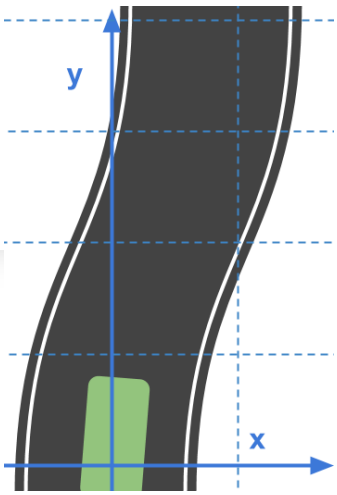

Trajectory Clustering 2 - Online Prediction
SEND FEE
ONLINE PREDICTION
Every update cycle...
1. Observe vehicle's partial trajectory
2. Compare to prototype trajectories
3. Predict a trajectory
P
C1 vs C2 vs C3
1
0 1 2 3 4
timestep
P = ⅓
P = ⅓
P = ⅓


ONLINE PREDICTION
Every update cycle...
1. Observe vehicle's partial trajectory
2. Compare to prototype trajectories
3. Predict a trajectory
P
C1 vs C2 vs C3
1
0 1 2 3 4
timestep
P= 0.2
P= 0.7
P= 0.1









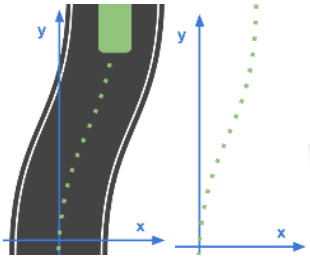
Frenet Coordinates

Before we discuss process models, we should mention "Frenet Coordinates", which are a way of representing position on a road in a more intuitive way than traditional (x, y) Cartesian Coordinates.

With Frenet coordinates, we use the variables s and d to describe a vehicle's position on the road. The s coordinate represents distance *along* the road (also known as **longitudinal displacement**) and the d coordinate represents side-to-side position on the road (also known as **lateral displacement**).

Why do we use Frenet coordinates? Imagine a curvy road like the one below with a Cartesian coordinate system laid on top of it...

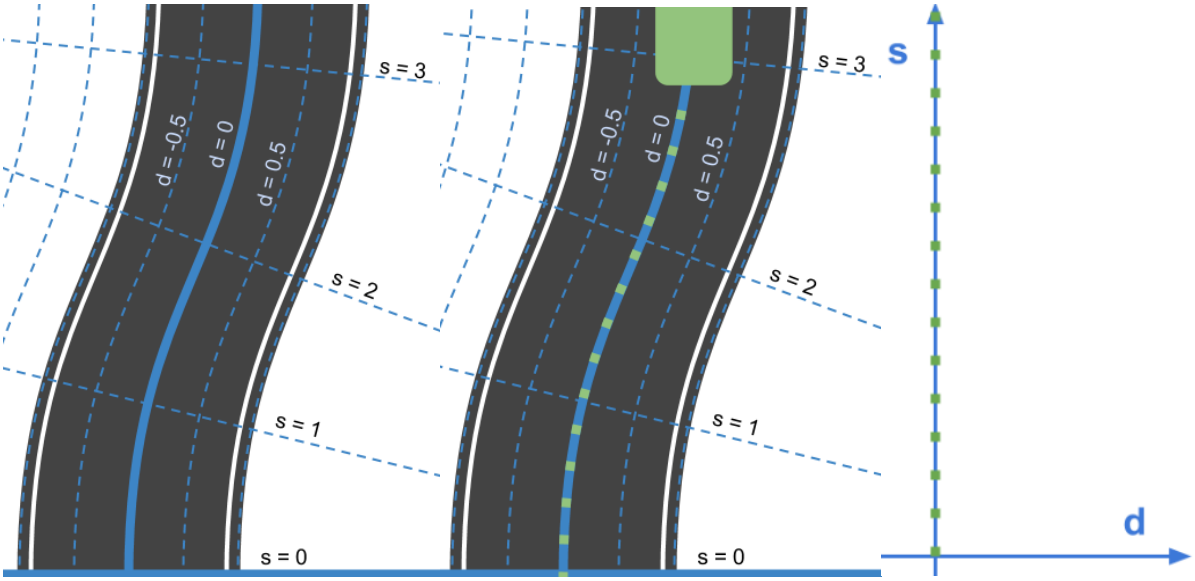
Using these Cartesian coordinates, we can try to describe the path a vehicle would normally follow on the road...



And notice how curvy that path is! If we wanted *equations* to describe this motion it wouldn't be easy!

$x(t) = ?$

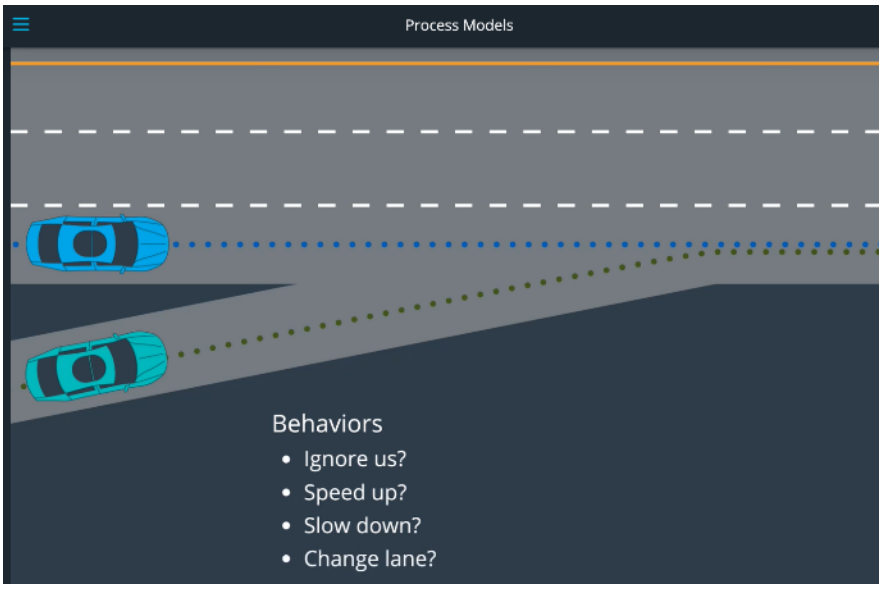
$y(t) = ?$

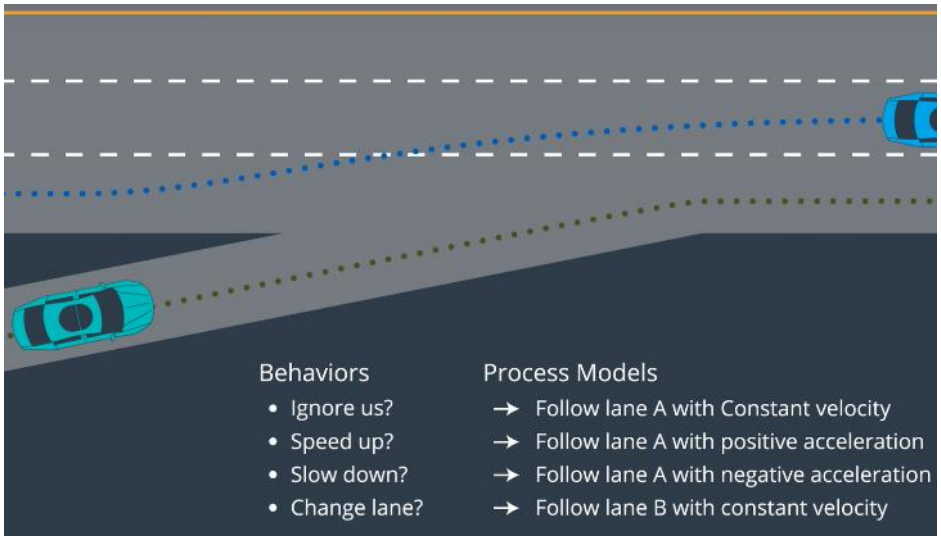


In fact, if this vehicle were moving at a constant speed of v_0 we could write a mathematical description of the vehicle's position as:

$s(t) = v_0 t$

$d(t) = 0$





Process Models

SEND FEEDBACK

Four models for lane following

Linear point model (constant velocity)

$$\begin{bmatrix} \dot{s} \\ \dot{d} \end{bmatrix} = \begin{bmatrix} \dot{s_0} \\ 0 \end{bmatrix} + \mathbf{W}$$

NON-linear point model (constant acceleration with curvature)

$c(s) = c_0 + c_1 s$

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{v} \\ \dot{\omega} \\ \dot{a} \\ \dot{c_0} \\ \dot{c_1} \end{bmatrix} = \begin{bmatrix} (v+at)cos(\theta) \\ (v+at)sin(\theta) \\ \omega \\ a \\ 0 \\ 0 \\ vc_1 \\ 0 \end{bmatrix} + \mathbf{W}$$

Kinematic bicycle model with controller (PID controller on distance and angle)

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} v\,cos(\theta) \\ v\,sin(\theta) \\ \frac{v}{L}\,tan(\delta) \\ a \end{bmatrix} + \mathbf{W}$$

Dynamic bicycle model with controller (PID controller on distance and angle)

$$\begin{bmatrix} \ddot{s} \\ \ddot{d} \\ \ddot{\theta} \end{bmatrix} = \begin{bmatrix} \dot{\theta}\dot{d} + a_s \\ -\dot{\theta}\dot{s} + \frac{2}{m}(F_{c,f}\,cos\,\delta + F_{c,r}) \\ \frac{2}{I_z}(l_f F_{c,f} - l_r F_{c,r}) \end{bmatrix} + \mathbf{W}$$

$$\delta_t = -J_P CTE - J_D \dot{CTE} - J_I \sum_{i=0}^{t} CTE_i$$

Notes on Notation

1. Matrix Notation

When you see something like the following:

$$F_{CV} = \text{diag}[F_2, F_2], F_2 = \begin{bmatrix} 1 & T \\ 0 & 1 \end{bmatrix}$$

it means that F is a 4x4 matrix, with F_2 as blocks along the diagonal. Written out fully, this means:

$$F_{CV} = \begin{bmatrix} 1 & T & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & T \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2. State Space

The process models all use cartesian coordinates. The state space is

$$\mathbf{x} = \begin{bmatrix} x \\ \dot{x} \\ y \\ \dot{y} \end{bmatrix}$$

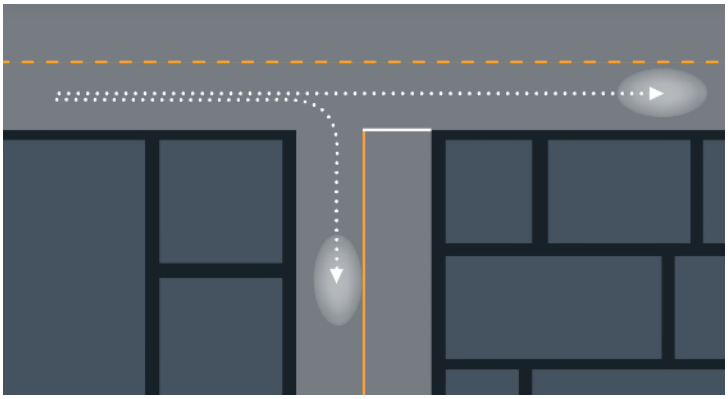
3. Variables

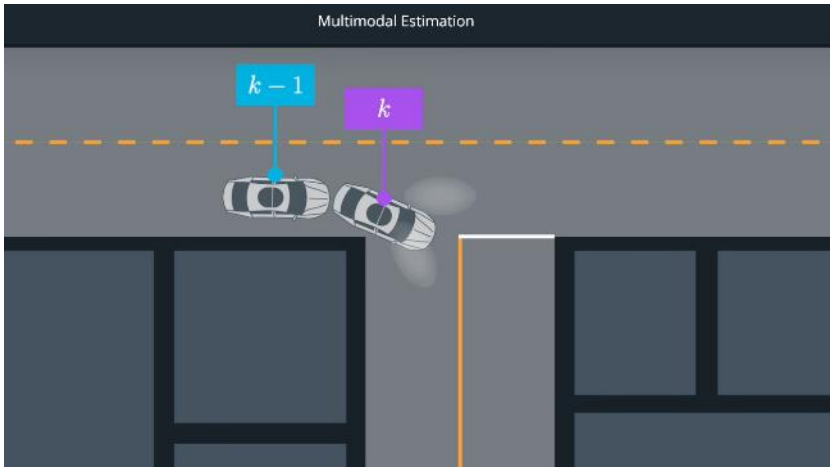
The equation $x_k = Fx_{k-1} + Gu_{k-1} + Gw_k, \;\; w_k \sim \mathcal{N}(0, Q)$ should be read as follows:

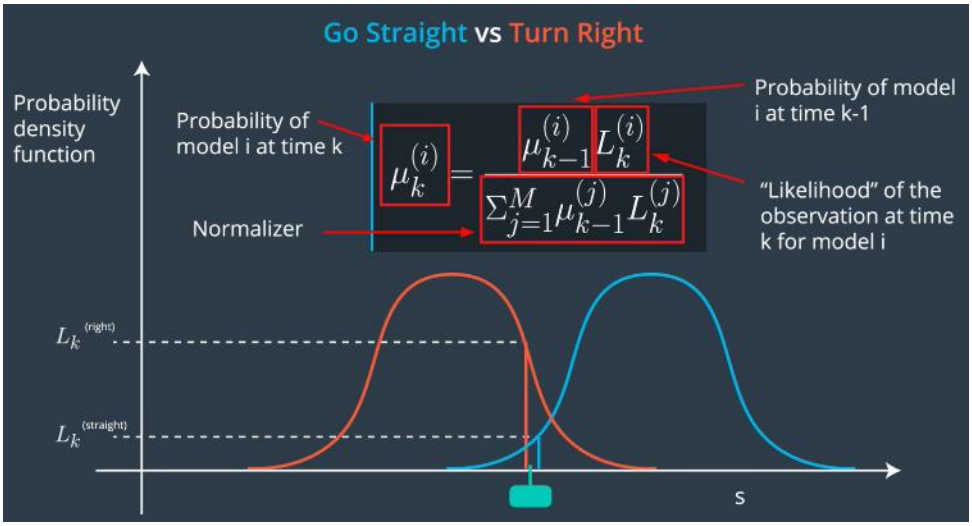
> the **predicted state at time k** (x_k) is given by **evolving** (F) the **previous state** (x_{k-1}), **incorporating** (G) the **controls** (u_{k-1}) given at the previous time step, and **adding normally distributed noise** (w_k).

AUTONOMOUS MULTIPLE MODEL ALGORITHM - VARIABLES

- Consider some set of M process models / behaviors
- Probabilities for process models $\mu_1, \; \mu_2, \; \ldots, \mu_M$







1. Data-Driven Approaches

Data-driven approaches solve the prediction problem in two phases:

1. Offline training
2. Online Prediction

1.1 Offline Training

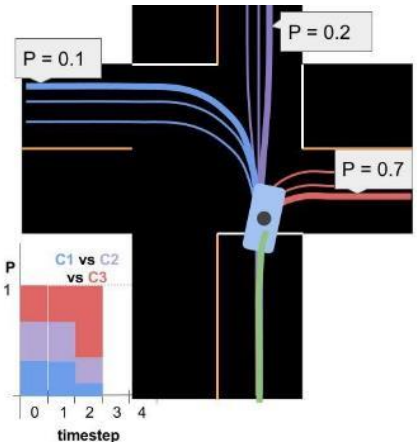
In this phase the goal is to feed some machine learning algorithm a lot of data to train it. For the trajectory clustering example this involved:

1. **Define similarity** - we first need a definition of similarity that agrees with human common-sense definition.
2. **Unsupervised clustering** - at this step some machine learning algorithm clusters the trajectories we've observed.
3. **Define Prototype Trajectories** - for each cluster identify some small number of typical "prototype" trajectories.

1.2 Online Prediction

Once the algorithm is trained we bring it onto the road. When we encounter a situation for which the trained algorithm is appropriate (returning to an intersection for example) we can use that algorithm to actually predict the trajectory of the vehicle. For the intersection example this meant:

1. **Observe Partial Trajectory** - As the target vehicle drives we can think of it leaving a "partial trajectory" behind it.
2. **Compare to Prototype Trajectories** - We can compare this partial trajectory to the *corresponding parts* of the prototype trajectories. When these partial trajectories are more similar (using the same notion of similarity defined earlier) their likelihoods should increase relative to the other trajectories.
3. **Generate Predictions** - For each cluster we identify the most likely prototype trajectory. We broadcast each of these trajectories along with the associated probability (see the image below).



2. Model Based Approaches

You can think of model based solutions to the prediction problem as also having an "offline" and online component. In that view, this approach requires:

1. *Defining* process models (offline).
2. *Using* process models to compare driver behavior to what would be expected for each model.
3. *Probabilistically classifying* driver intent by comparing the likelihoods of various behaviors with a multiple-model algorithm.
4. *Extrapolating* process models to generate trajectories.

2.1 Defining Process Models

You saw how process models can vary in complexity from very simple...

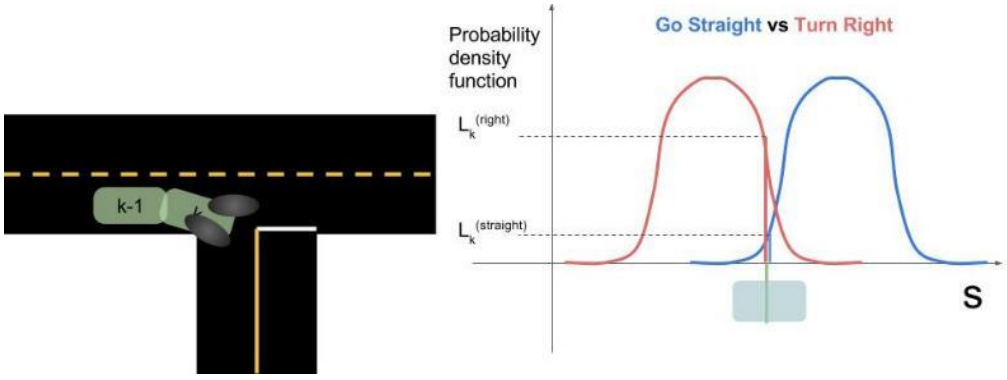
$$\begin{bmatrix} \dot{s} \\ \dot{d} \end{bmatrix} = \begin{bmatrix} s_0 \\ 0 \end{bmatrix} + \mathbf{w}$$

to very complex...

$$\begin{bmatrix} \ddot{s} \\ \ddot{d} \\ \ddot{\theta} \end{bmatrix} = \begin{bmatrix} \dot{\theta}\,\dot{d} + a_s \\ -\dot{\theta}\,\dot{s} + \frac{2}{m}(F_{c,f}\cos\delta + F_{c,r}) \\ \frac{2}{I_z}(l_f F_{c,f} - l_r F_{c,r}) \end{bmatrix} + \mathbf{w}$$

2.2 Using Process Models

Process Models are first used to compare a target vehicle's observed behavior to the behavior we would expect for each of the maneuvers we've created models for. The pictures below help explain how process models are used to calculate these likelihoods.



On the left we see two images of a car. At time $k-1$ we predicted where the car would be if it were to go straight vs go right. Then at time k we look at where the car actually is. The graph on the right shows the car's observed s coordinate along with the probability distributions for where we *expected* the car to be at that time. In this case, the s that we observe is substantially more consistent with turning right than going straight.

2.3 Classifying Intent with Multiple Model Algorithm

In the image at the top of the page you can see a bar chart representing probabilities of various *clusters* over time. Multiple model algorithms serve a similar purpose for model based approaches: they are responsible for maintaining beliefs for the probability of each maneuver. The algorithm we discussed is called the **Autonomous Multiple Model** algorithm (AMM). AMM can be summarized with this equation:

$$\mu_k^{(i)} = \frac{\mu_{k-1}^{(i)} L_k^{(i)}}{\sum_{j=1}^{M} \mu_{k-1}^{(j)} L_k^{(j)}}$$

or, if we ignore the denominator (since it just serves to normalize the probabilities), we can capture the essence of this algorithm with

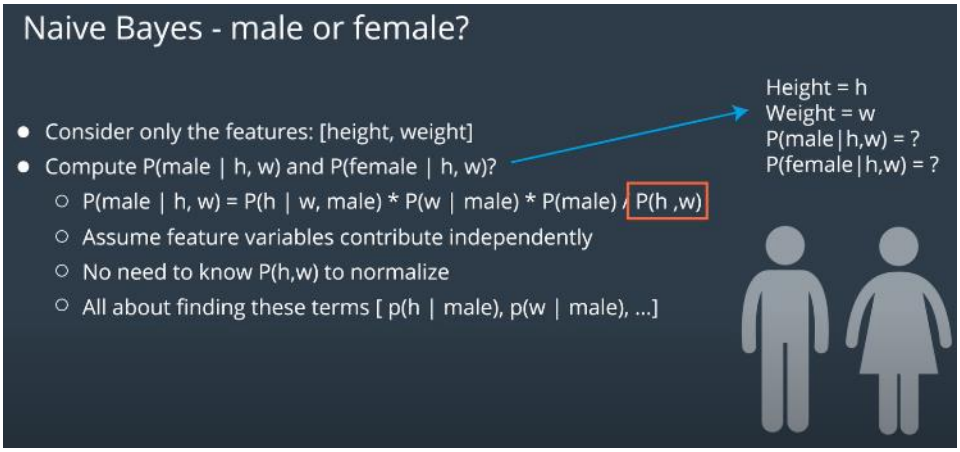
$$\mu_k^{(i)} \propto \mu_{k-1}^{(i)} L_k^{(i)}$$

where the $\mu_k^{(i)}$ is the probability that model number i is the correct model at time k and $L_k^{(i)}$ is the **likelihood** for that model (as computed by comparison to process model).

The paper, **"A comparative study of multiple model algorithms for maneuvering target tracking"** is a good reference to learn more.

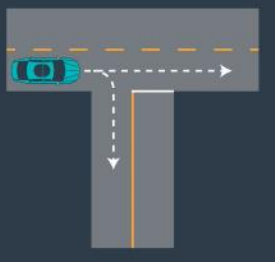
2.4 Trajectory Generation

Trajectory generation is straightforward once we have a process model. We simply iterate our model over and over until we've generated a prediction that spans whatever time horizon we are supposed to cover. Note that each iteration of the process model will necessarily add uncertainty to our prediction.



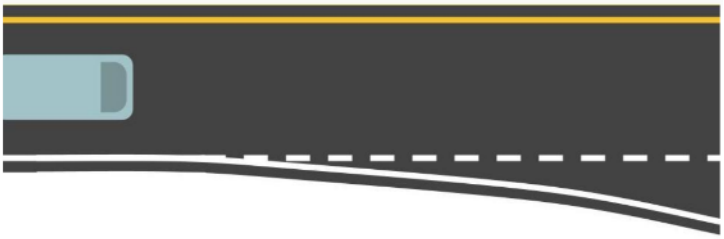
Gaussian Naive Bayes: “assume individual probabilities have gaussian distributions”.

P(h | male) ~ N(mu_male_height, var_male_height)



In practice the problem is all about:

1. Selecting relevant features
2. Identifying the means / variances for different classes
   a. Can be guessed or...
   b. Learned!



QUIZ QUESTION

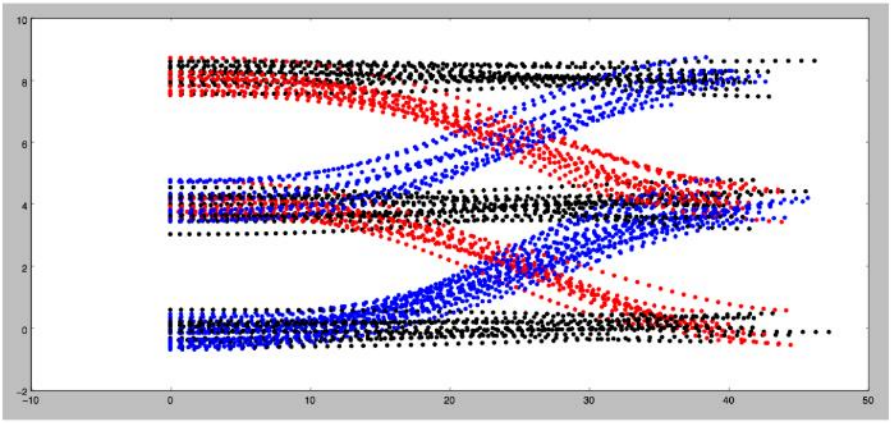
A car on a highway is approaching an exit ramp. We want to classify the driver's intent as "go straight" or "exit right". Which of the following state variables would be **least** useful to this classification?

- ✓ s
- $srate of change (ds/dt)$
- d
- $drate of change (dd/dt)$

Implementing Naive Bayes

In this exercise you will implement a Gaussian Naive Bayes classifier to predict the behavior of vehicles on a highway. In the image below you can see the behaviors you'll be looking for on a 3 lane highway (with lanes of 4 meter width). The dots represent the d (y axis) and s (x axis) coordinates of vehicles as they either...

1. change lanes left (shown in blue)
2. keep lane (shown in black)
3. or change lanes right (shown in red)



Your job is to write a classifier that can predict which of these three maneuvers a vehicle is engaged in given a single coordinate (sampled from the trajectories shown below).

Each coordinate contains 4 features:

- s
- d
- $\dot{s}$
- $\dot{d}$

You also know the **lane width** is 4 meters (this might be helpful in engineering additional features for your algorithm).

Instructions

1. Implement the `train(data, labels)` method in the class `GNB` in `classifier.cpp`.

   Training a Gaussian Naive Bayes classifier consists of computing and storing the mean and standard deviation from the data for each label/feature pair. For example, given the label "change lanes left" and the feature $\dot{s}$, it would be necessary to compute and store the mean and standard deviation of $\dot{s}$ over all data points with the "change lanes left" label.

   Additionally, it will be convenient in this step to compute and store the prior probability p(C_k) for each label C_k. This can be done by keeping track of the number of times each label appears in the training data.

2. Implement the `predict(observation)` method in `classifier.cpp`.

   Given a new data point, prediction requires two steps:

   1. **Compute the conditional probabilities for each feature/label combination**. For a feature x and label C with mean μ and standard deviation σ (computed in training), the conditional probability can be computed using the formula here:

      $$p(x = v|C) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp^{-\frac{(v-\mu)^2}{2\sigma^2}}$$

      Here v is the value of feature x in the new data point.

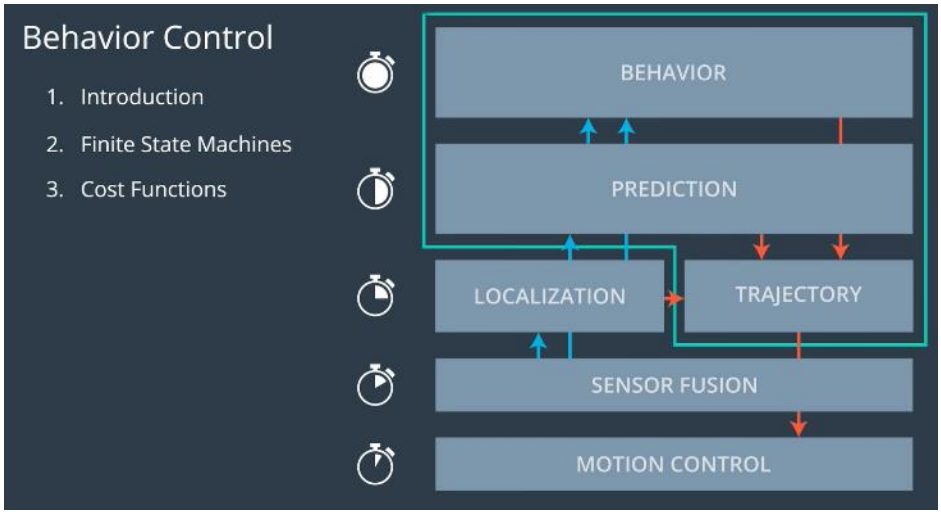
   2. **Use the conditional probabilities in a Naive Bayes classifier.** This can be done using the formula here:

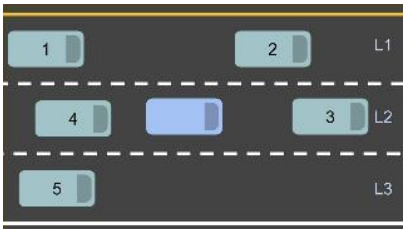
      $$y = \underset{k \in (1,\ldots,K)}{argmax} \; p(C_k) \prod_{i=1}^{n} p(x_i = v_i|C_k)$$

      In this formula, the argmax is taken over all possible labels C_k and the product is taken over all features x_i with values v_i.

3. When you want to test your classifier, run `Test Run` and check out the results.

NOTE: You are welcome to use some existing implementation of a Gaussian Naive Bayes classifier. But to get the **best** results you will still need to put some thought into what **features** you provide the algorithm when classifying. Though you will only be given the 4 coordinates listed above, you may find that by "engineering" features you may get better performance. For example: the raw value of the d coordinate may not be that useful. But `d % lane_width` might be helpful since it gives the *relative* position of a vehicle in it's lane regardless of which lane the vehicle is in.





Output A

```
{
    "target_lane_id" : 2,
    "target_leading_vehicle_id": 3,
    "target_speed" : null,
    "seconds_to_reach_target" : null,
}
```

Output B

```
{
    "target_lane_id" : 3,
    "target_leading_vehicle_id": null,
    "target_speed" : 20.0,
    "seconds_to_reach_target" : 5.0,
}
```

Output C

```
{
    "target_lane_id" : 2,
    "target_leading_vehicle_id": null,
    "target_speed" : 15.0,
    "seconds_to_reach_target" : 10.0,
}
```

Output D

```
{
    "target_lane_id" : 2,
    "target_leading_vehicle_id": 2,
    "target_speed" : null,
    "seconds_to_reach_target" : 5.0,
}
```

Output E

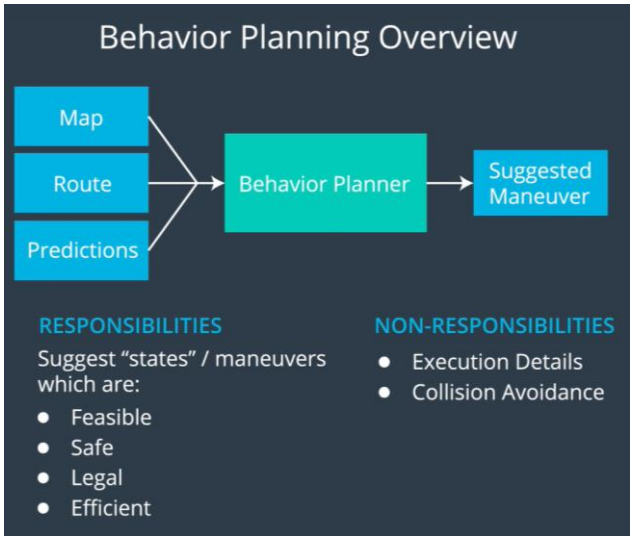
```
{
    "target_lane_id" : 1,
    "target_leading_vehicle_id": 2,
    "target_speed" : null,
    "seconds_to_reach_target" : 5.0,
}
```

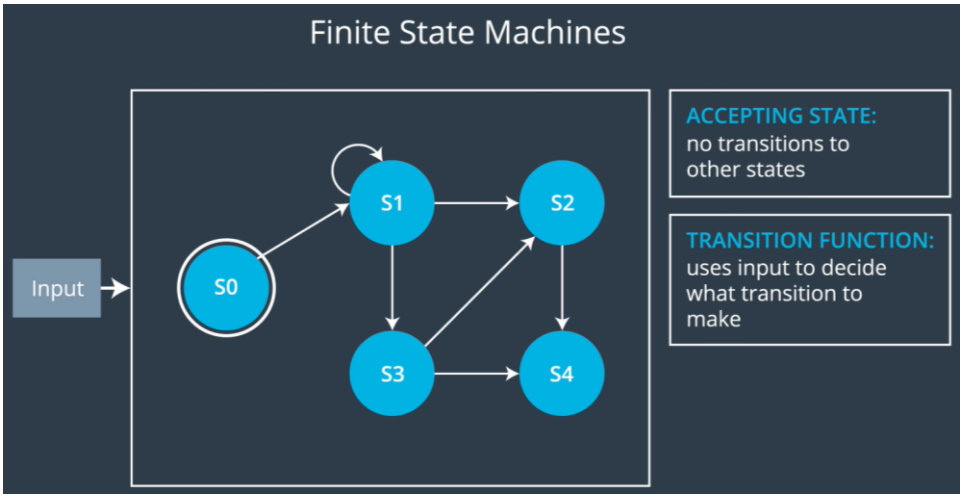
QUIZ QUESTION

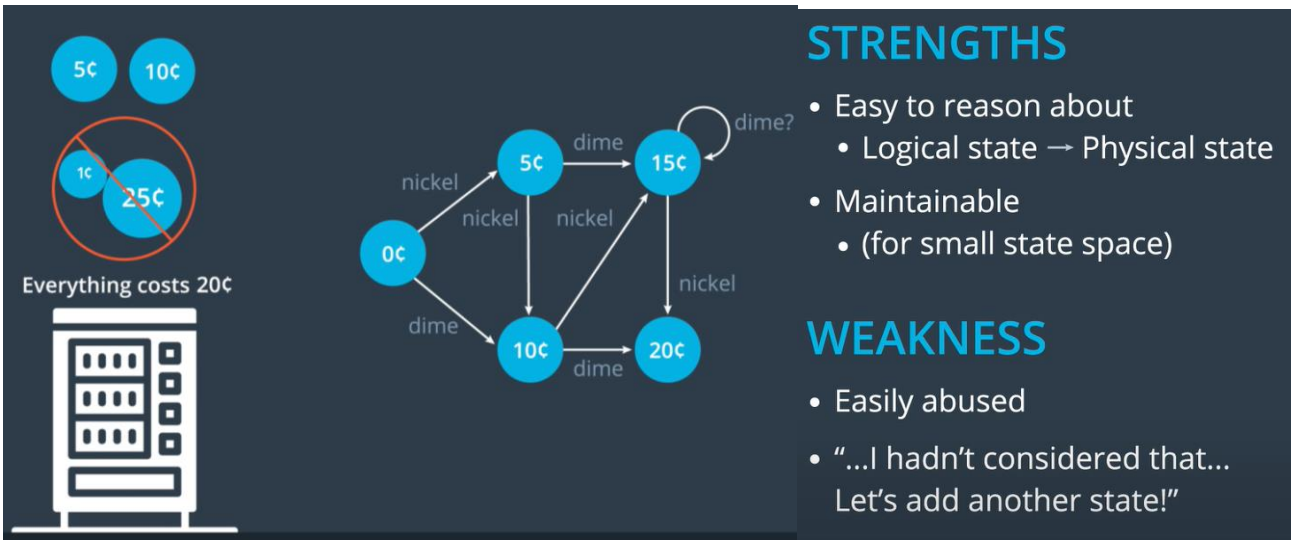
Match the verbal description of each behavior to the corresponding `json` representation.

Submit to check your answer choices!

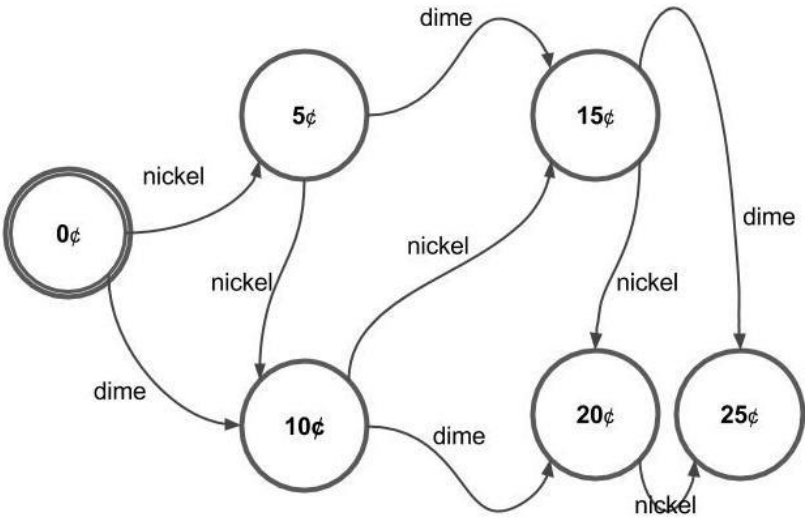
VERBAL DESCRIPTION	JSON OUTPUT
"Just stay in your lane and keep following the car in front of you"	A
"Let's pass this car! Get in the left lane and follow that car"	E
"Let's pass this car! But first we have to match speeds with the car in the left lane. Stay in this lane but get behind that car in the left lane and match their speed."	D
"Whoa! This car we've been following is going too fast. Stop trying to follow it and just try to go the speed limit."	C
"Get in the right lane soon."	B







Requiring exact change



QUESTION 1 OF 3

Which state in the vending machine example was the **start state**?

- ✓ 0 cents

QUESTION 2 OF 3

Which state(s) were **accepting states**?

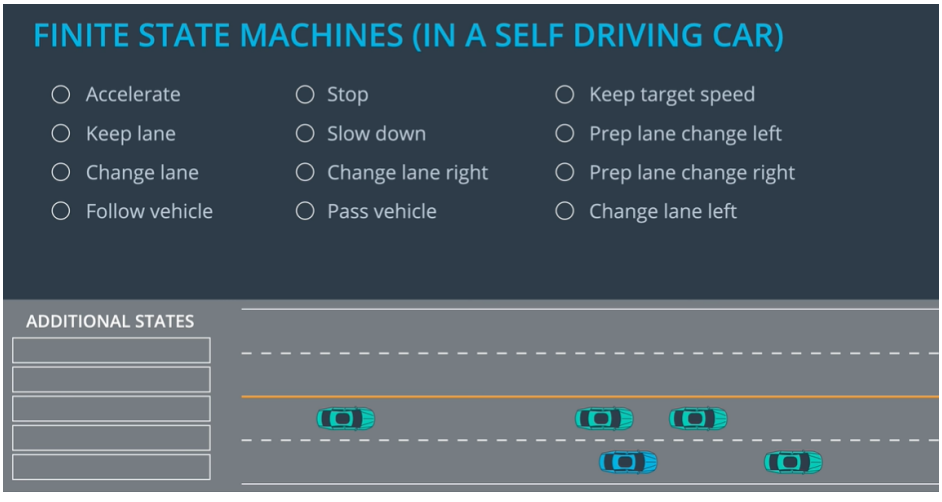
- ☐ 0 cents
- ☐ 5 cents
- ☐ 10 cents
- ☐ 15 cents
- ☐ 20 cents
- ✓ 25 cents

QUESTION 3 OF 3

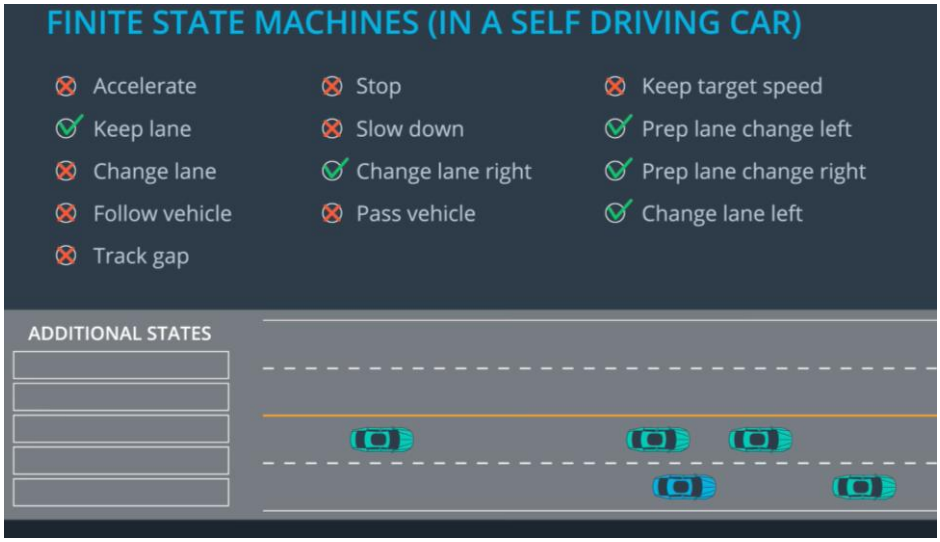
If we wanted this machine to also accept pennies (1 cent coin) how many additional **states** would we have to add?

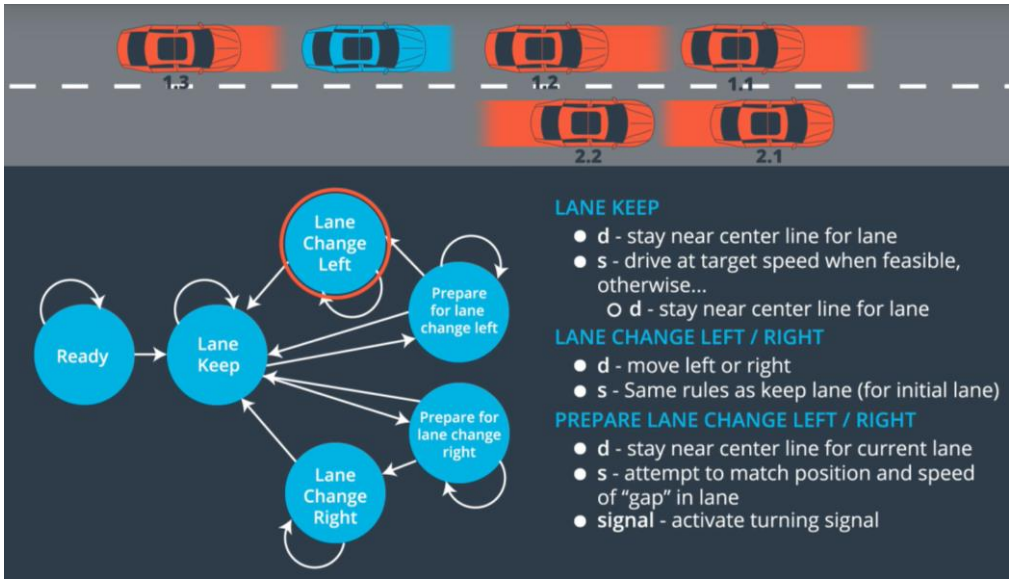
- ○ 0
- ○ 1
- ✓ 20
- ○ 25

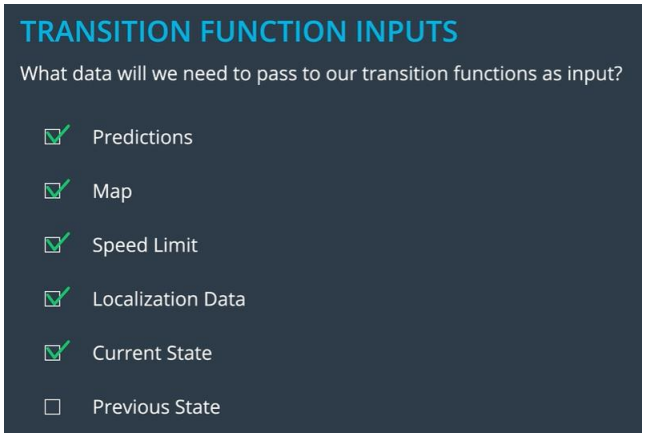
We would need one state for each integer between 0 and 25 (which is 26). We already have 6 states. So we would need 20 **new** states.



There is not a correct answer:
For the sake of the course





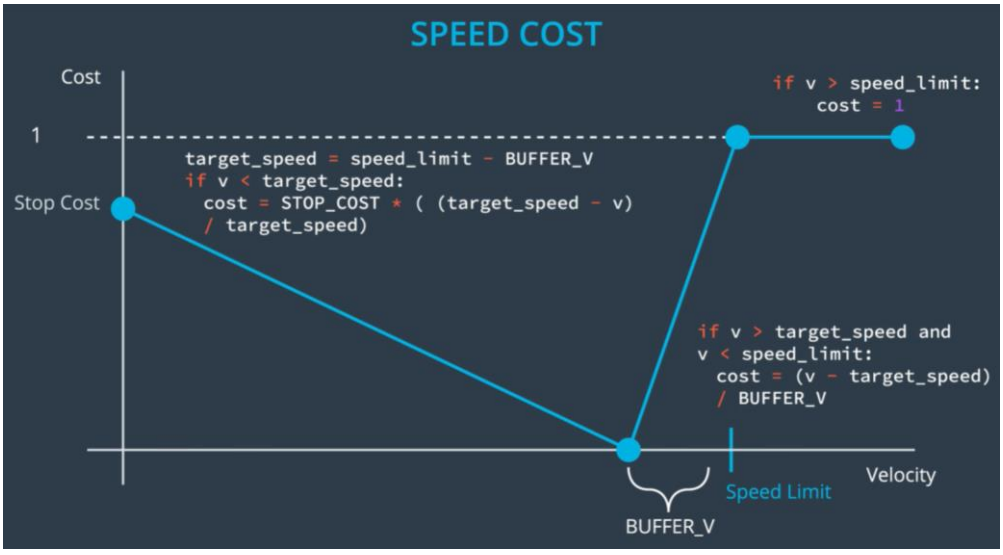


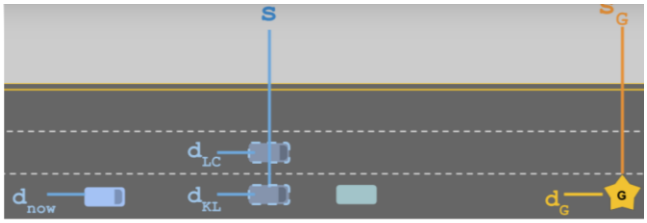
Behavior Planning Pseudocode

One way to implement a transition function is by generating rough trajectories for each accessible "next state" and then finding the best. To "find the best" we generally use **cost functions**. We can then figure out how costly each rough trajectory is and then select the state with the lowest cost trajectory.

We'll discuss this in more detail later, but first read carefully through the pseudocode below to get a better sense for how a transition function might work.

```
def transition_function(predictions, current_fsm_state, current_pose, cost_functions, weights):
    # only consider states which can be reached from current FSM state.
    possible_successor_states = successor_states(current_fsm_state)

    # keep track of the total cost of each state.
    costs = []
    for state in possible_successor_states:
        # generate a rough idea of what trajectory we would
        # follow IF we chose this state.
        trajectory_for_state = generate_trajectory(state, current_pose, predictions)

        # calculate the "cost" associated with that trajectory.
        cost_for_state = 0
        for i in range(len(cost_functions)) :
            # apply each cost function to the generated trajectory
            cost_function = cost_functions[i]
            cost_for_cost_function = cost_function(trajectory_for_state, predictions)

            # multiply the cost by the associated weight
            weight = weights[i]
            cost_for_state += weight * cost_for_cost_function
         costs.append({'state' : state, 'cost' : cost_for_state})

    # Find the minimum cost state.
    best_next_state = None
    min_cost = 9999999
    for i in range(len(possible_successor_states)):
        state = possible_successor_states[i]
        cost  = costs[i]
        if cost < min_cost:
            min_cost = cost
            best_next_state = state

    return best_next_state
```





In the image above, the blue self driving car (bottom left) is trying to get to the goal (gold star). It's currently in the correct lane but the green car is going very slowly, so it considers whether it should perform a lane change (LC) or just keep lane (KL). These options are shown as lighter blue vehicles with a dashed outline.

If we want to design a cost function that deals with lane choice, it will be helpful to establish what the relevant variables are. In this case, we can define:

- $\Delta s = s_G - s$ how much distance the vehicle will have before it has to get into the goal lane.
- $\Delta d = d_G - d_{LC/KL}$ the lateral distance between the goal lane and the options being considered. In this case $\Delta d_{KL} = d_G - d_{KL}$ would be zero and $\Delta d_{LC} = d_G - d_{LC}$ would not.

Before we define an actual cost function, let's think of some of the properties we want it to have...

QUESTION 1 OF 3

First, thinking **only** about delta d: Would we prefer the absolute value of Δd to be **big** or **small**?

- Big
- ✓ Small

So we want a cost function that penalizes large $|\Delta d|$ and we want that penalty to be bigger when Δs is small.

Furthermore, we want to make sure that the **maximum** cost of this cost function never exceeds one and that the **minimum** never goes below zero.

Which of the following proposals meets these criteria?

Option 1: cost $= |\Delta d| + \frac{1}{\Delta s}$

Option 2: cost $= \frac{|\Delta d|}{\Delta s}$

Option 3: cost $= 1 - e^{-\frac{|\Delta d|}{\Delta s}}$

QUESTION 3 OF 3

Which of the options shown above meet the criteria we want (assume Δs is always positive)?

- Option 1
- Option 2
- ✓ Option 3

SUBMIT

In this example, we found that the ratio $\frac{|\Delta d|}{\Delta s}$ was important. If we call that ratio x we can then use that ratio in any function with bounded range. These functions tend to be useful when designing cost functions. These types of functions are called Sigmoid Functions. You can learn more in the Wikipedia

QUESTION 2 OF 3

Now let's think about how s factors into our considerations of lane cost. Should costs associated with lane change be more important when we are **far** from the goal (in s coordinate) or **close** to the goal?

- Far
- ✓ Close

$$\text{cost} = 1 - e^{-\frac{|\Delta d|}{\Delta s}}$$

Here, Δd was the lateral distance between the goal lane and the final chosen lane, and Δs was the longitudinal distance from the vehicle to the goal.

In this quiz, we'd like you to implement the cost function in C++, but with one important change. The finite state machine we use for vehicle behavior also includes states for planning a lane change right or left (PLCR or PLCL), and the cost function should incorporate this information. We will provide the following four inputs to the function:

- Intended lane: the intended lane for the given behavior. For PLCR, PLCL, LCR, and LCL, this would be the one lane over from the current lane.
- Final lane: the immediate resulting lane of the given behavior. For LCR and LCL, this would be one lane over.
- The Δs distance to the goal.
- The goal lane.

Your task in the implementation will be to modify $|\Delta d|$ in the equation above so that it satisifes:

- $|\Delta d|$ is smaller as both intended lane and final lane are closer to the goal lane.
- The cost function provides different costs for each possible behavior: KL, PLCR/PLCL, LCR/LCL.
- The values produced by the cost function are in the range 0 to 1.

main.cpp | cost.cpp | cost.h

```cpp
#include "cost.h"
#include <cmath>

double goal_distance_cost(int goal_lane, int intended_lane, int final_lane,
                          double distance_to_goal) {
  // The cost increases with both the distance of intended lane from the goal
  //   and the distance of the final lane from the goal. The cost of being out
  //   of the goal lane also becomes larger as the vehicle approaches the goal.

  /**
   * TODO: Replace cost = 0 with an appropriate cost function.
   */
  double delta_d = (goal_lane - intended_lane) + (goal_lane - final_lane);
  double cost = 1.0 - exp(- fabs(delta_d) / distance_to_goal);

  return cost;
}
```

```
Costs for (intended_lane, final_lane, goal_distance):
-------------------------------------------------------------
The cost is 0.981684 for (2, 2, 1.0)
The cost is 0.32968 for (2, 2, 10.0)
The cost is 0.0392106 for (2, 2, 100.0)
The cost is 0.0295545 for (1, 2, 100.0)
The cost is 0.0198013 for (1, 1, 100.0)
The cost is 0.00995017 for (0, 1, 100.0)
The cost is 0 for (0, 0, 100.0)
```

Implement a Second Cost Function in C++

In most situations, a single cost function will not be sufficient to produce complex vehicle behavior. In this quiz, we'd like you to implement one more cost function in C++. We will use these two C++ cost functions later in the lesson. The goal with this quiz is to create a cost function that would make the vehicle drive in the fastest possible lane, given several behavior options. We will provide the following four inputs to the function:

- Target speed: Currently set as 10 (unitless), the speed at which you would like the vehicle to travel.
- Intended lane: the intended lane for the given behavior. For PLCR, PLCL, LCR, and LCL, this would be the one lane over from the current lane.
- Final lane: the immediate resulting lane of the given behavior. For LCR and LCL, this would be one lane over.
- A vector of lane speeds, based on traffic in that lane: {6, 7, 8, 9}.

Your task in the implementation will be to create a cost function that satisifes:

- The cost decreases as both intended lane and final lane are higher speed lanes.
- The cost function provides different costs for each possible behavior: KL, PLCR/PLCL, LCR/LCL.
- The values produced by the cost function are in the range 0 to 1.

You can implement your solution in `cost.cpp` below.

main.cpp | cost.cpp | cost.h

```cpp
#include "cost.h"

double inefficiency_cost(int target_speed, int intended_lane, int final_lane,
                         const std::vector<int> &lane_speeds) {
  // Cost becomes higher for trajectories with intended lane and final lane
  //   that have traffic slower than target_speed.

  /**
   * TODO: Replace cost = 0 with an appropriate cost function.
   */
  double speed_intended = lane_speeds[intended_lane];
  double speed_final = lane_speeds[final_lane];

  double cost = ((target_speed - speed_intended) + (target_speed - speed_final)) / target_spe

  return cost;
}
```

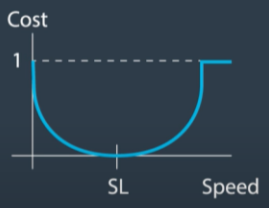
```
Costs for (intended_lane, final_lane):
-----------------------------------------
The cost is 0.2 for (3, 3)
The cost is 0.3 for (2, 3)
The cost is 0.4 for (2, 2)
The cost is 0.5 for (1, 2)
The cost is 0.6 for (1, 1)
The cost is 0.7 for (0, 1)
The cost is 0.8 for (0, 0)
```

COST FUNCTION DESIGN

DIFFICULTIES

- Solving new problems without "unsolving" old ones.
  - Regression Testing!
- Balancing costs of drastically different magnitudes
  - Feasibility >> Safety >> Legality >> Comfort >> Efficiency
  - Weights can change depending on situation
- Reasoning about individual CFs
  - Specificity of cost function responsibility
  - Binary vs discrete vs continuous cost functions
  - ALL CFs output between -1 and 1
  - Parametrization (when possible)
  - Thinking in terms of vehicle state (position, velocity, acceleration)

Class	Position	Velocity	Acceleration
Feasibility			
Safety			
Legality		Speed < speed limit?	
Comfort			
Efficiency		Speed ~= speed limit	



Class	Position	Velocity	Acceleration
Feasibility	Avoids Collision?		Acceleration is feasible for car?
Safety	Buffer Distance	Speed ~= traffic speed	
Legality	Stays on Road?	Speed < speed limit?	
Comfort	Near center of current lane		Low change in acceleration (jerk)
Efficiency	Desired Lane	Speed ~= speed limit	

Obeys traffic rules?

Consider the following cost functions. What purpose does each serve?

Cost Function 1

$$\begin{cases} 1 & \ddot{s} \geq a\_\max \\ 0 & \ddot{s} < a\_\max \end{cases}$$

Cost Function 2

$$\begin{cases} 1 & d \geq d\_\max \\ 1 & d \leq d\_\min \\ 0 & d\_\min < d < d\_\max \end{cases}$$

Cost Function 3

$$\begin{cases} 1 & \dot{s} \geq v\_\text{speed limit} \\ 0 & \dot{s} < v\_\text{speed limit} \end{cases}$$

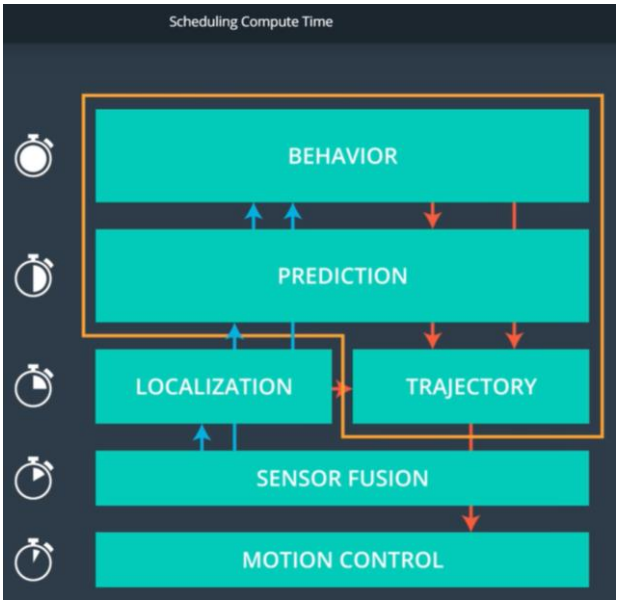
Cost Function 4

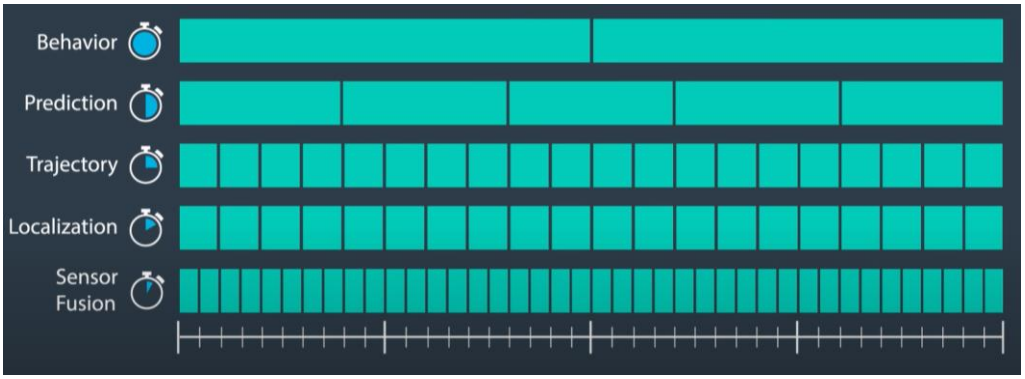
$$\frac{1}{1 + e^{-(d - d\_\text{lane center})^2}}$$

Cost Function 5

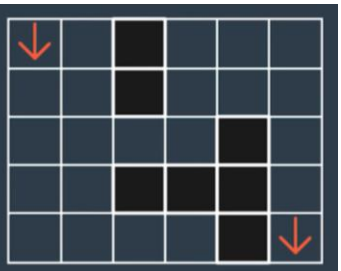
$$(\text{lane number} - \text{target lane number})^2$$

COST FUNCTION VERBAL DESCRIPTION	
Penalizes trajectories that do not stay near the center of the lane.	4
Penalizes trajectories that attempt to accelerate at a rate which is not possible for the vehicle.	1
Rewards trajectories that stay near the target lane.	5
Penalizes trajectories that drive off the road.	2
Penalizes trajectories that exceed the speed limit.	3





THE MOTION PLANNING PROBLEM



Configuration Space:

- Defines all possible configurations of our robot.
- In two dimensional world...
- Configuration space can be 2D [x, y] or...
- 3D [x,y,theta]...

Given:

- A start configuration q_{start} (from localization and sensors)
- A goal configuration q_{goal} (from behavior)
- Constraints (physics, map, traffic)

Problem:

- **Feasible** motion planning: Find a sequence of movements in **configuration space** that moves the robot from q_{start} to q_{goal} without hitting any obstacles.

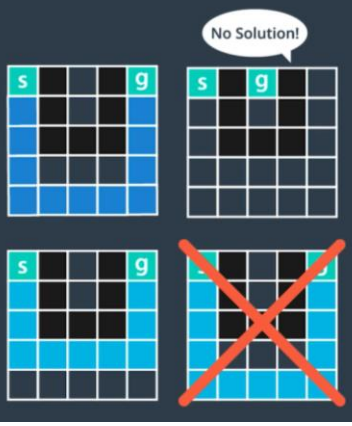
MOTION PLANNING ALGORITHMS PROPERTIES

Completeness

- If a solution exists, planner always finds a solution
- If no solution exists, terminates and reports failure.

Optimality

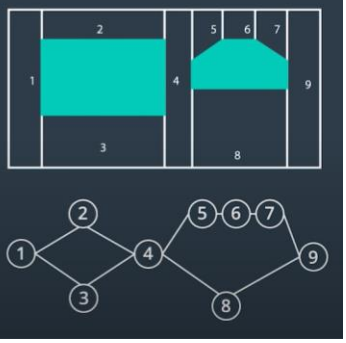
- Given a cost function for evaluating a sequence of actions, planner always returns a feasible sequence of actions with minimal cost.



MOTION PLANNING ALGORITHMS

Combinatorial Methods

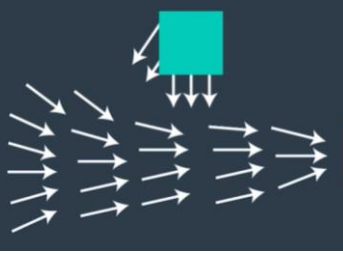
Motion Planning Algorithms



MOTION PLANNING ALGORITHMS

Potential Field Methods

Combinatorial Methods

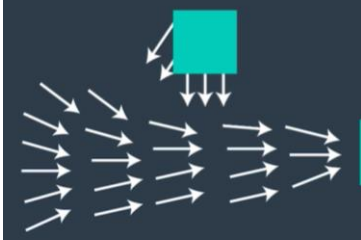


Motion Planning Algorithms

MOTION PLANNING ALGORITHMS

Potential Field Methods

Combinatorial Methods

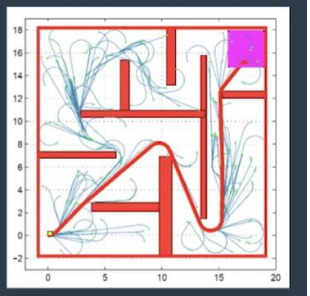


Motion Planning Algorithms

Optimal Control

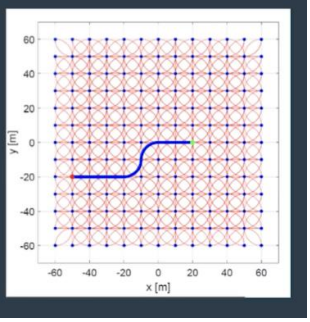
Sampling Based Methods

SAMPLING BASED METHODS



Discrete methods:

- Discretization of the configuration and input space
  - Deterministic graph search algorithms: A*, D*, D*-lite, Dijkstra's, ARA*, etc. ...



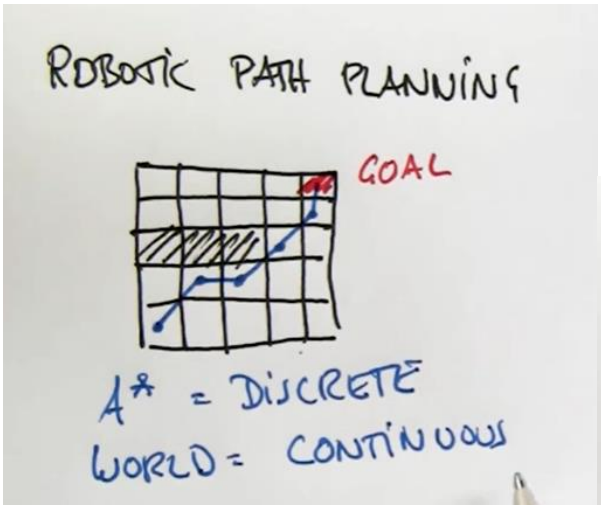
Probabilistic methods:

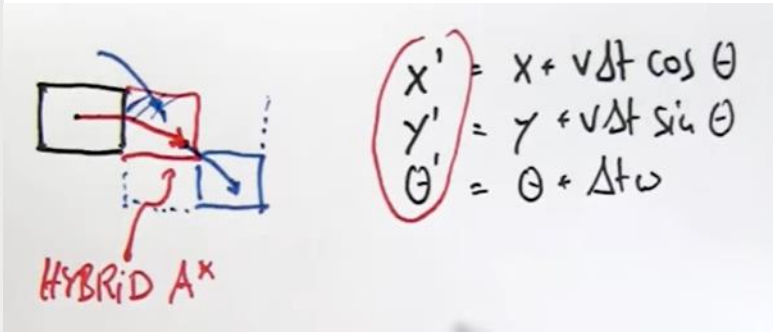
- Random exploration of the configuration and input space
  - Probabilistic graph search algorithms
    RRT, RRT*, PRM, etc ...

QUIZ QUESTION

Check all **true** statements about A*.

- [ ] It uses a continuous search space.
- [x] It uses an **optimistic** heuristic function to guide grid cell expansion.
- [x] It always finds a solution if one exists (completeness)
- [ ] Solutions it finds are **drivable**
- [x] Solutions it finds are **always optimal** assuming an admissible heuristic





A*	HYBRID A*
☒	☑ It is a continuous method
☑	☑ It uses an optimistic heuristic function to guide grid cell expansion
☑	☒ It always finds a solution if it exists
☒	☑ Solutions it finds are drivable
☑	☒ Solutions it finds are always optimal

$$x(t+\Delta t) = x(t) + v\Delta t\, cos(\theta)$$
$$y(t+\Delta t) = y(t) + v\Delta t\, sin(\theta)$$
$$\theta(t+\Delta t) = \theta(t) + \omega\Delta t$$

But that would mean that the robot can turn around its Z axis without constraints.

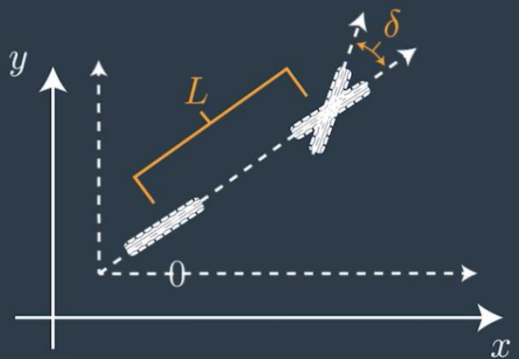
For a car, w depends on the state → bicycle model

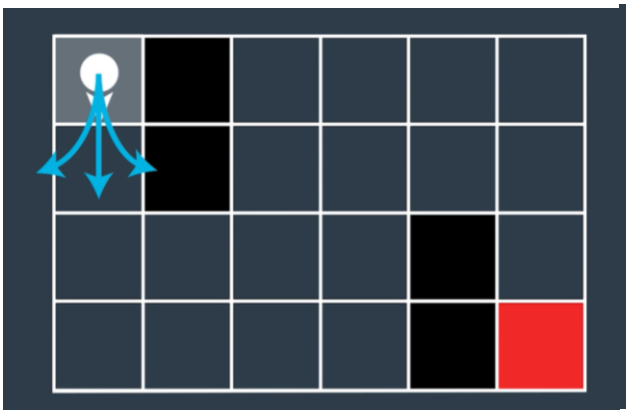
PRACTICAL CONSIDERATIONS

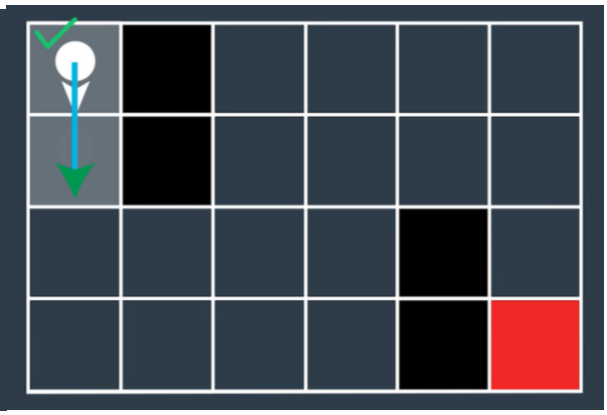
- Use maximum steering angle in both directions and 0
  - -35 deg, 0 deg, 35 deg
- For given speed, we assemble path out of 3 components:

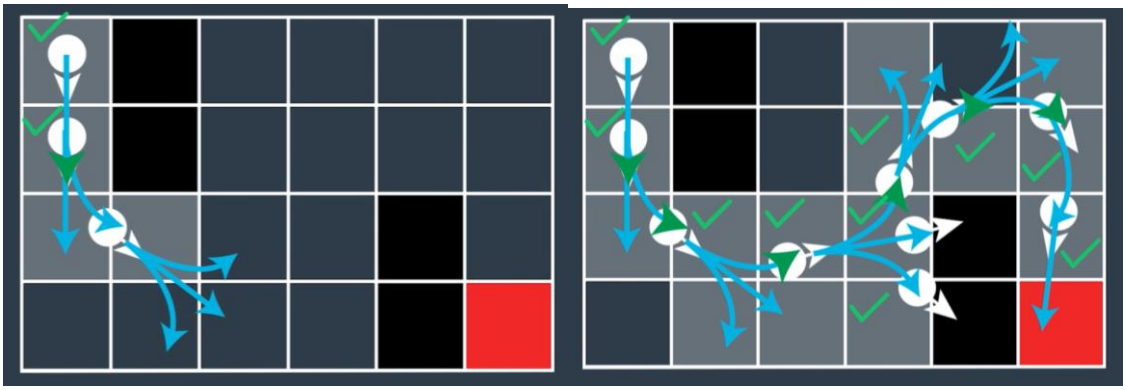
$$x(t+\Delta t) = x(t) + v\Delta t\, cos(\theta)$$
$$y(t+\Delta t) = y(t) + v\Delta t\, sin(\theta)$$
$$\theta(t+\Delta t) = \theta(t) + \omega\Delta t$$

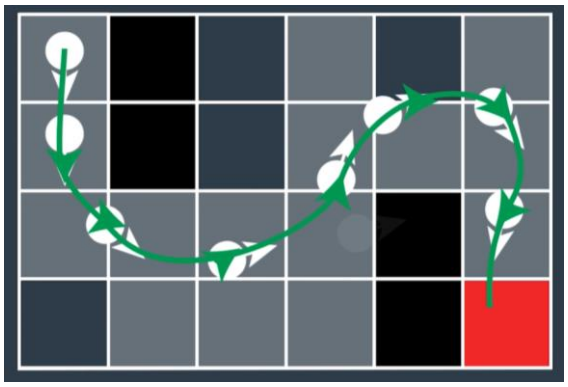
$$\omega = \frac{v}{L} tan(\delta(t))$$



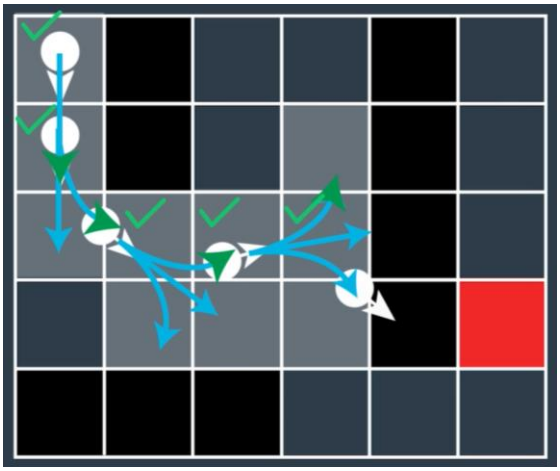








Low resolutions increases of failure chances:



If you have two individually admissible heuristic functions h_1 and h_2, which of the following **combinations** are also valid?

QUIZ QUESTION

Check the box next to all **admissible** combinations.

- [ ] h1 + h2
- [x] (h1 + h2) / 2
- [x] min(h1, h2)
- [x] max(h1, h2)

Hybrid A* Pseudocode:

The pseudocode below outlines an implementation of the A* search algorithm using the bicycle model. The following variables and objects are used in the code but not defined there:

- `State(x, y, theta, g, f)`: An object which stores `x`, `y` coordinates, direction `theta`, and current `g` and `f` values.
- `grid`: A 2D array of 0s and 1s indicating the area to be searched. 1s correspond to obstacles, and 0s correspond to free space.
- `SPEED`: The speed of the vehicle used in the bicycle model.
- `LENGTH`: The length of the vehicle used in the bicycle model.
- `NUM_THETA_CELLS`: The number of cells a circle is divided into. This is used in keeping track of which States we have visited already.

The bulk of the hybrid A* algorithm is contained within the `search` function. The `expand` function takes a state and goal as inputs and returns a list of possible next states for a range of steering angles. This function contains the implementation of the bicycle model and the call to the A* heuristic function.

```
def expand(state, goal):
    next_states = []
    for delta in range(-35, 40, 5):
        # Create a trajectory with delta as the steering angle using
        # the bicycle model:

        # ---Begin bicycle model---
        delta_rad = deg_to_rad(delta)
        omega = SPEED/LENGTH * tan(delta_rad)
        next_x = state.x + SPEED * cos(theta)
        next_y = state.y + SPEED * sin(theta)
        next_theta = normalize(state.theta + omega)
        # ---End bicycle model-----

        next_g = state.g + 1
        next_f = next_g + heuristic(next_x, next_y, goal)

        # Create a new State object with all of the "next" values.
        state = State(next_x, next_y, next_theta, next_g, next_f)
        next_states.append(state)

    return next_states

def search(grid, start, goal):
    # The opened array keeps track of the stack of States objects we are
    # searching through.
    opened = []
    # 3D array of zeros with dimensions:
    # (NUM_THETA_CELLS, grid x size, grid y size).
    closed = [[[0 for x in range(grid[0])] for y in range(len(grid))]
        for cell in range(NUM_THETA_CELLS)]
    # 3D array with same dimensions. Will be filled with State() objects
    # to keep track of the path through the grid.
    came_from = [[[0 for x in range(grid[0])] for y in range(len(grid))]
        for cell in range(NUM_THETA_CELLS)]

    # Create new state object to start the search with.
    x = start.x
    y = start.y
    theta = start.theta
    g = 0
    f = heuristic(start.x, start.y, goal)
    state = State(x, y, theta, 0, f)
    opened.append(state)

    # The range from 0 to 2pi has been discretized into NUM_THETA_CELLS cells.
    # Here, theta_to_stack_number returns the cell that theta belongs to.
    # Smaller thetas (close to 0 when normalized  into the range from 0 to
    # 2pi) have lower stack numbers, and larger thetas (close to 2pi when
    # normalized) have larger stack numbers.
    stack_num = theta_to_stack_number(state.theta)
    closed[stack_num][index(state.x)][index(state.y)] = 1

    # Store our starting state. For other states, we will store the previous
    # state in the path, but the starting state has no previous.
    came_from[stack_num][index(state.x)][index(state.y)] = state
```

```
# While there are still states to explore:
while opened:
    # Sort the states by f-value and start search using the state with the
    # lowest f-value. This is crucial to the A* algorithm; the f-value
    # improves search efficiency by indicating where to look first.
    opened.sort(key=lambda state:state.f)
    current = opened.pop(0)

    # Check if the x and y coordinates are in the same grid cell
    # as the goal. (Note: The idx function returns the grid index for
    # a given coordinate.)
    if (idx(current.x) == goal[0]) and (idx(current.y) == goal.y):
        # If so, the trajectory has reached the goal.
        return path

    # Otherwise, expand the current state to get a list of possible
    # next states.
    next_states = expand(current, goal)
    for next_s in next_states:
        # If we have expanded outside the grid, skip this next_s.
        if next_s is not in the grid:
            continue
        # Otherwise, check that we haven't already visited this cell and
        # that there is not an obstacle in the grid there.
        stack_num = theta_to_stack_number(next_s.theta)
        if closed[stack_num][idx(next_s.x)][idx(next_s.y)] == 0
            and grid[idx(next_s.x)][idx(next_s.y)] == 0:
            # The state can be added to the opened stack.
            opened.append(next_s)
            # The stack_number, idx(next_s.x), idx(next_s.y) tuple
            # has now been visited, so it can be closed.
            closed[stack_num][idx(next_s.x)][idx(next_s.y)] = 1
            # The next_s came from the current state, and is recorded.
            came_from[stack_num][idx(next_s.x)][idx(next_s.y)] = current
```

ENVIRONMENT CLASSIFICATION

Unstructured

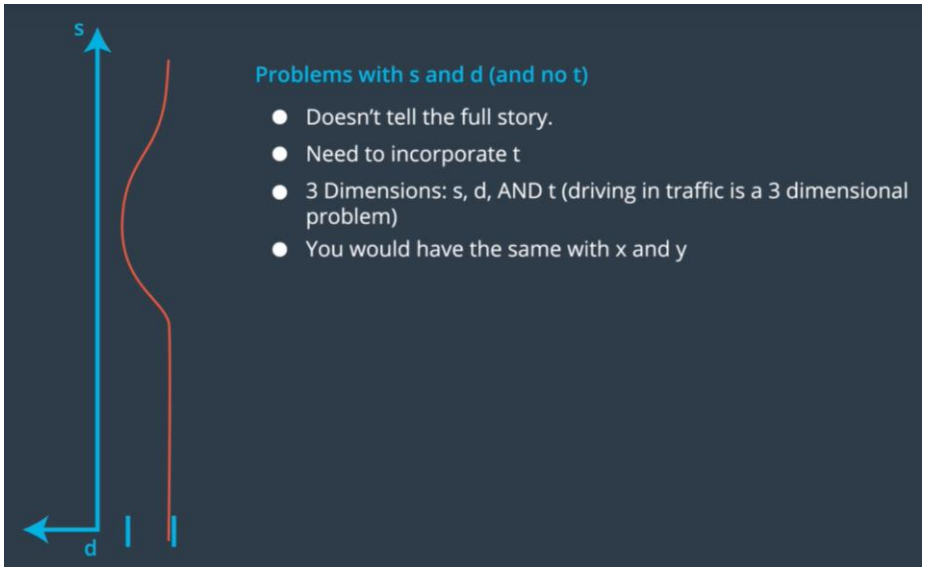
Ex: Parking lot (maze)

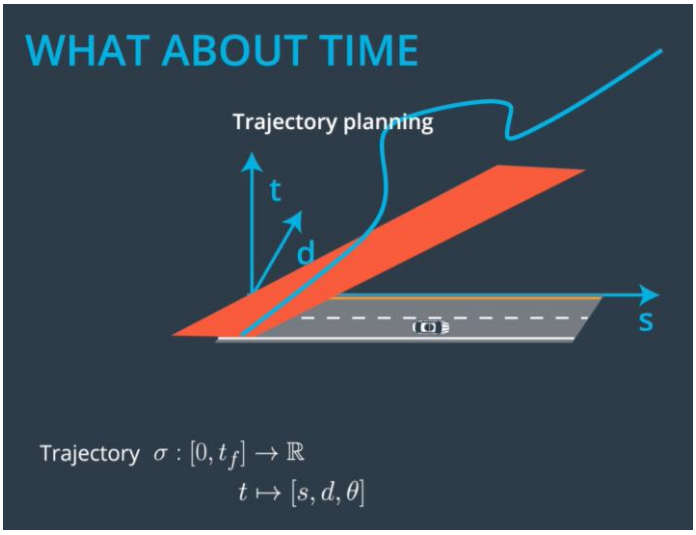
- Less specific rules and lower speeds
- No obvious reference path or trajectory

Structured

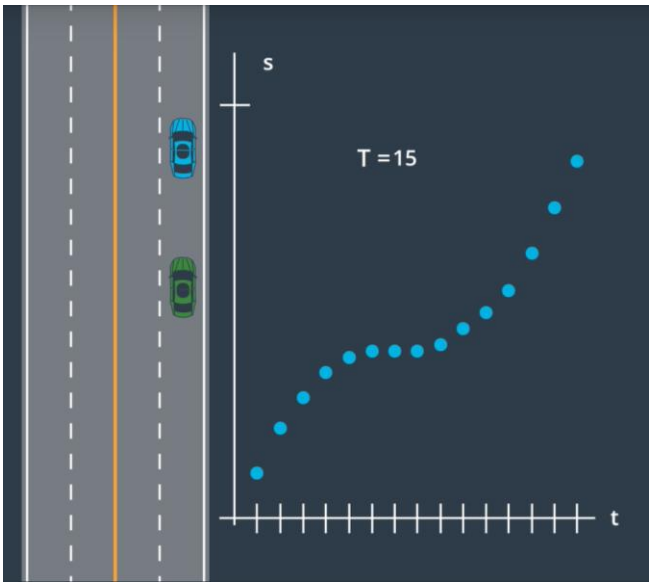
Ex: Highway, Street Driving

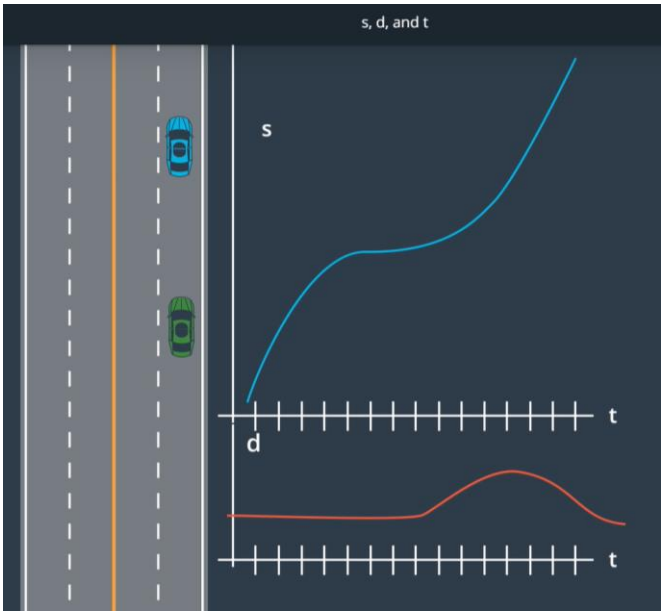
- Predefined rules regarding how to move on the road
  - Direction of traffic
  - Lane boundaries
  - Speed limits
- Road structure can be used a reference

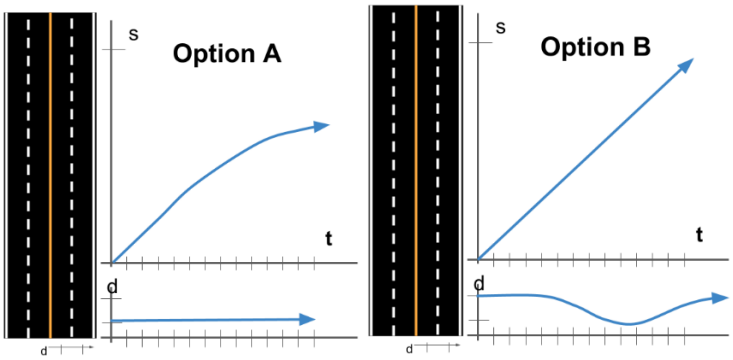


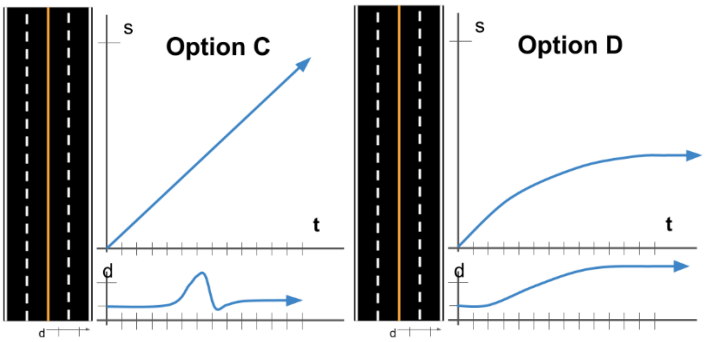


If the curve is under the red plane, it will collide assuming a car is coming from behind





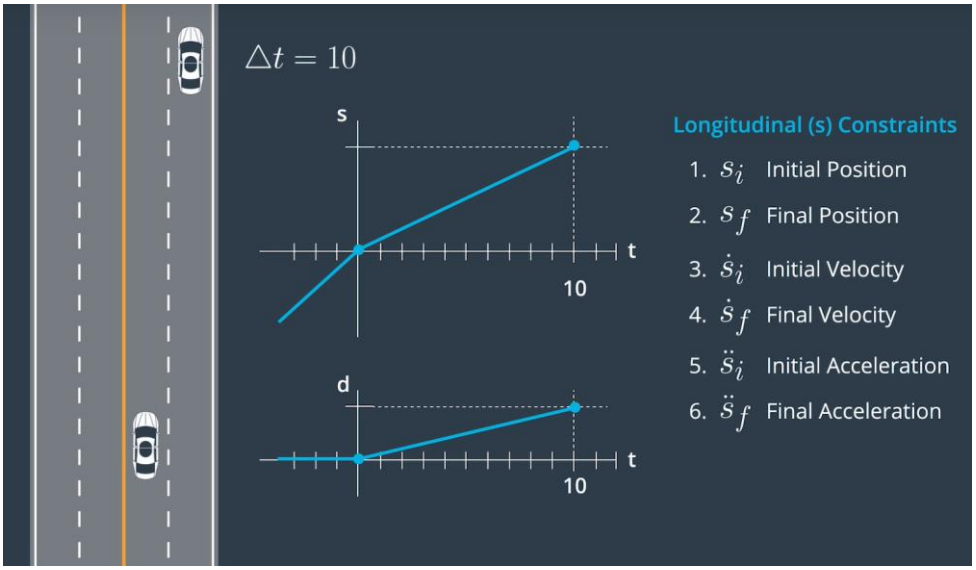




VERBAL DESCRIPTION

Slow down in lane	A
Swerve quickly	C
Pass vehicle	B
Pull over and stop	D

The following functions are not possible because transitions are instantaneous which would require infinite accelerations. Therefore, they must be smooth



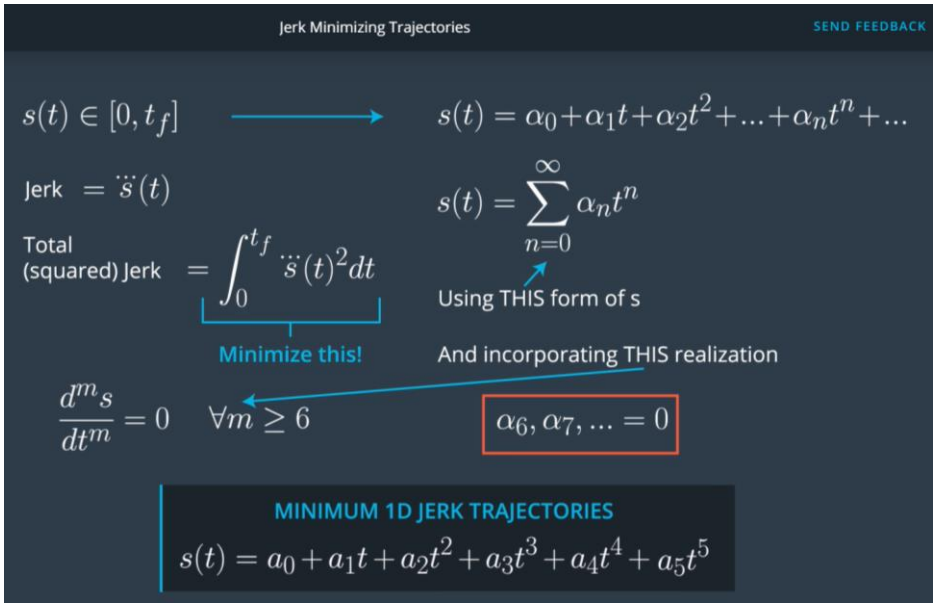
Need position continuity

Need velocity continuity

Position → Velocity → Acceleration → Jerk → Snap → Crackle → Pop

Humans feel discomfort when _____ is high

- Velocity? No (airplanes)
- Acceleration? No (freefall)
- Jerk? Yes (Bumper cars)
- Design for jerk minimization



MINIMUM 1D JERK TRAJECTORIES

$$s(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5$$

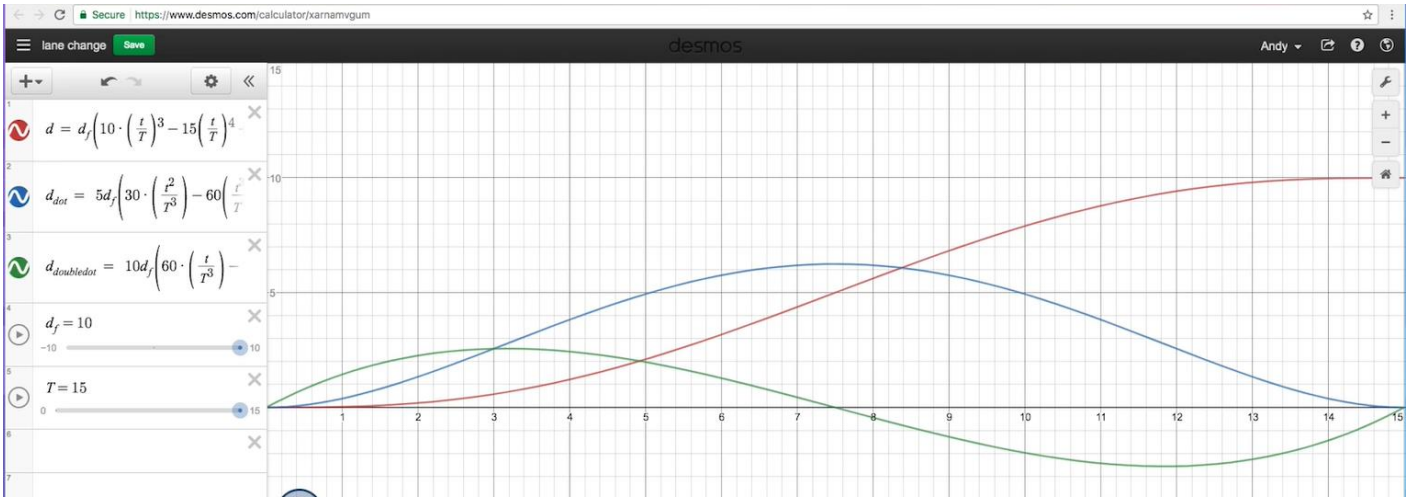
6 Coefficients → 6 Tunable Parameters → 6 Boundary Conditions

$[s_i, \dot{s}_i, \ddot{s}_i, s_f, \dot{s}_f, \ddot{s}_f]$

$[d_i, \dot{d}_i, \ddot{d}_i, d_f, \dot{d}_f, \ddot{d}_f]$

$[s_i, \dot{s}_i, \ddot{s}_i, s_f, \dot{s}_f, \ddot{s}_f]$

$[\,0\,,\,0\,,\,0\,,\,10\,,\,0\,,\,0\,]$



$$s(t) = \alpha_0 + \alpha_1 t + \alpha_2 t^2 + \alpha_3 t^3 + \alpha_4 t^4 + \alpha_5 t^5$$

$$\dot{s}(t) = \alpha_1 + 2\alpha_2 t + 3\alpha_3 t^2 + 4\alpha_4 t^3 + 5\alpha_5 t^4$$

$$\ddot{s}(t) = 2\alpha_2 + 6\alpha_6 t + 12\alpha_4 t^2 + 20\alpha_5 t^3$$

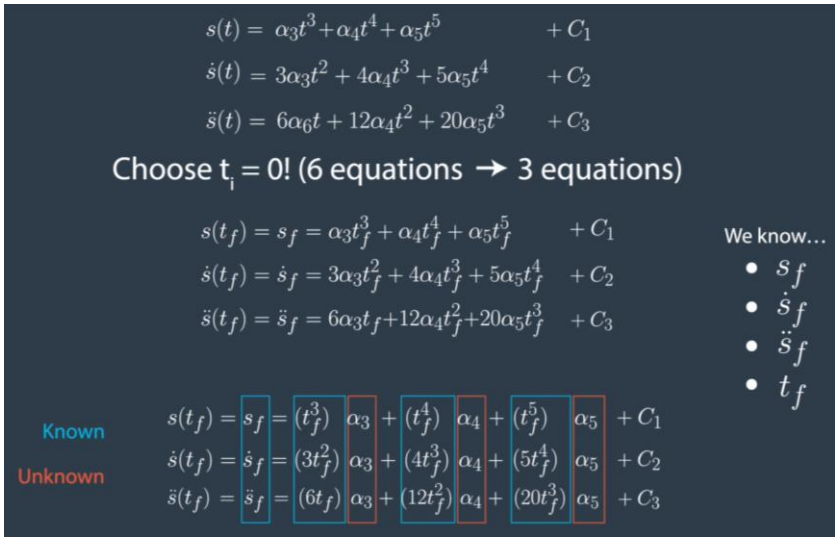
Choose t_i = 0! (6 equations → 3 equations)

$$s_i = s(0) = \alpha_0$$
$$\dot{s}_i = \dot{s}(0) = \alpha_1$$
$$\ddot{s}_i = \ddot{s}(0) = 2\alpha_2$$

$$s(t) = s_i + \dot{s}_i t + \frac{\ddot{s}_i}{2} t^2 + \alpha_3 t^3 + \alpha_4 t^4 + \alpha_5 t^5$$

$$\dot{s}(t) = \dot{s}_i + \frac{\ddot{s}_i}{2} t^2 + 3\alpha_3 t^2 + 4\alpha_4 t^3 + 5\alpha_5 t^4$$

$$\ddot{s}(t) = \ddot{s}_i + 6\alpha_3 t + 12\alpha_4 t^2 + 20\alpha_5 t^3$$



$$\begin{bmatrix} t_f^3 & t_f^4 & t_f^5 \\ 3t_f^2 & 4t_f^3 & 5t_f^4 \\ 6t_f & 12t_f^2 & 20t_f^3 \end{bmatrix} \times \begin{bmatrix} \alpha_3 \\ \alpha_4 \\ \alpha_5 \end{bmatrix} = \begin{bmatrix} s_f - C_1 \\ \dot{s}_f - C_2 \\ \ddot{s}_f - C_3 \end{bmatrix}$$

The equations for position, velocity, and acceleration are given by:

$$s(t) = s_i + \dot{s}_i t + \frac{\ddot{s}_i}{2} t^2 + \alpha_3 t^3 + \alpha_4 t^4 + \alpha_5 t^5$$

$$\dot{s}(t) = \dot{s}_i + \ddot{s}_i t + 3\alpha_3 t^2 + 4\alpha_4 t^3 + 5\alpha_5 t^4$$

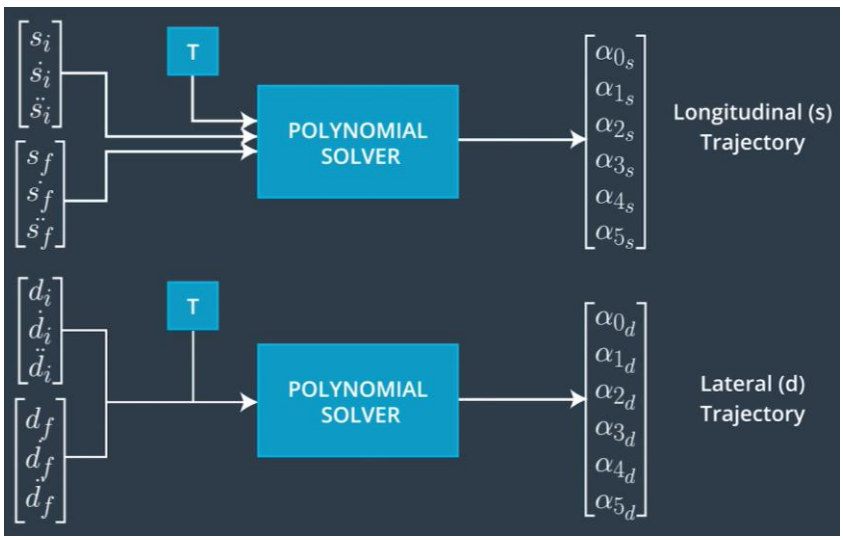
$$\ddot{s}(t) = \ddot{s}_i + 6\alpha_3 t + 12\alpha_4 t^2 + 20\alpha_5 t^3$$

and if you evaluate these at $t = 0$ you find the first three coeffecients of your JMT are:

$[\alpha_0, \alpha_1, \alpha_2] = [s_i, \dot{s}_i, \frac{1}{2}\ddot{s}_i]$

and you can get the last three coefficients by evaluating these equations at $t = T$. When you carry out the math and write the problem in matrix form you get the following:

$$\begin{bmatrix} T^3 & T^4 & T^5 \\ 3T^2 & 4T^3 & 5T^4 \\ 6T & 12T^2 & 20T^3 \end{bmatrix} \times \begin{bmatrix} \alpha_3 \\ \alpha_4 \\ \alpha_5 \end{bmatrix} = \begin{bmatrix} s_f - (s_i + \dot{s}_i T + \frac{1}{2}\ddot{s}_i T^2) \\ \dot{s_f} - (\dot{s}_i + \ddot{s}_i T) \\ \ddot{s_f} - \ddot{s}_i \end{bmatrix}$$



```
#include <cmath>
#include <iostream>
#include <vector>

#include "Dense"
#include "grader.h"

using std::vector;
using Eigen::MatrixXd;
using Eigen::VectorXd;

/**
 * TODO: complete this function
 */
vector<double> JMT(vector<double> &start, vector<double> &end, double T) {
  /**
   * Calculate the Jerk Minimizing Trajectory that connects the initial state
   * to the final state in time T.
   *
   * @param start - the vehicles start location given as a length three array
   *   corresponding to initial values of [s, s_dot, s_double_dot]
   * @param end - the desired end state for vehicle. Like "start" this is a
   *   length three array.
   * @param T - The duration, in seconds, over which this maneuver should occur.
   *
   * @output an array of length 6, each value corresponding to a coefficent in
   *   the polynomial:
   *   s(t) = a_0 + a_1 * t + a_2 * t**2 + a_3 * t**3 + a_4 * t**4 + a_5 * t**5
   *
   * EXAMPLE
   *   > JMT([0, 10, 0], [10, 10, 0], 1)
   *     [0.0  10.0  0.0  0.0  0.0  0.0]
   */
  MatrixXd A = MatrixXd(3, 3);
  A << T*T*T, T*T*T*T, T*T*T*T*T,
       3*T*T, 4*T*T*T, 5*T*T*T*T,
       6*T, 12*T*T, 20*T*T*T;

  MatrixXd B = MatrixXd(3, 1);
  B << end[0] - (start[0] + start[1]*T + .5*start[2]*T*T),
       end[1] - (start[1] + start[2]*T),
       end[2] - start[2];

  MatrixXd invA = A.inverse();

  MatrixXd C = invA * B;

  vector<double> result = {start[0], start[1], .5*start[2]};

  for(int i = 0; i < C.size(); i++){
      result.push_back(C.data()[i]);
  }
  return result;
}
```

When evaluating the **feasibility** of a potential trajectory, which of the following quantities should be checked?

QUIZ: WHAT SHOULD BE CHECKED? SOLUTION

- Max Velocity (wrt car capabilities and speed limit)
- Min Velocity
- Max Acceleration
- Min Acceleration
- Steering Angle

- Cannot exceed max speed of vehicle
- Minimum velocity → usually shouldn't be negative (no backing up)
- Lateral acceleration needs to be checked to avoid rollover / skid
- Longitudinal acceleration needs to be checked with powertrain capabilities
- Corresponds to max braking force
- Cannot exceed max / min steering

Feasibility

Longitudinal Acceleration = $\ddot{s}$

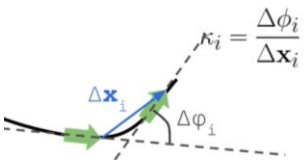
$$a_{\text{MAX BREAKING}} < \ddot{s} < a_{\text{MAX ACCEL}}$$

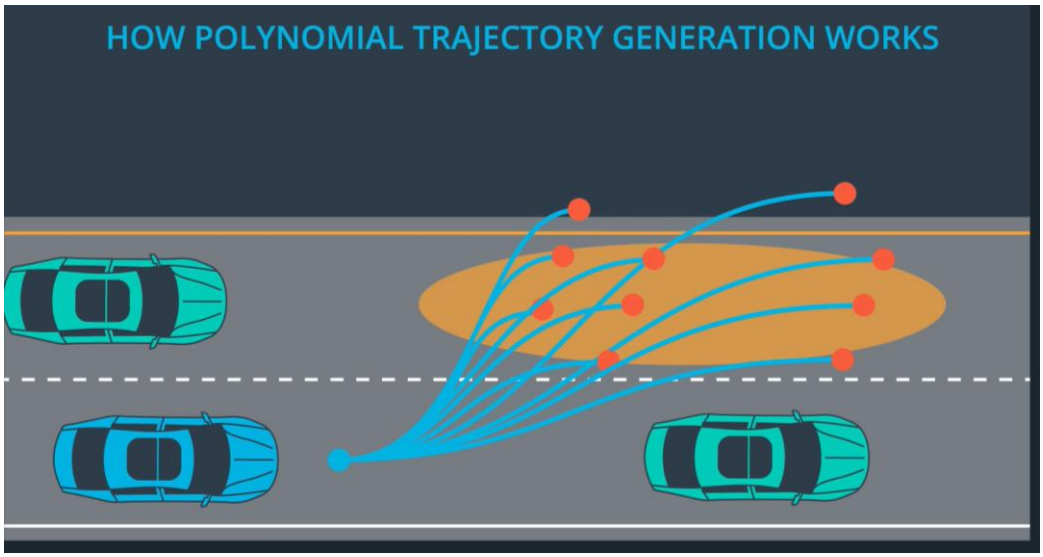
Lateral Acceleration $|\ddot{d}| < a_y$

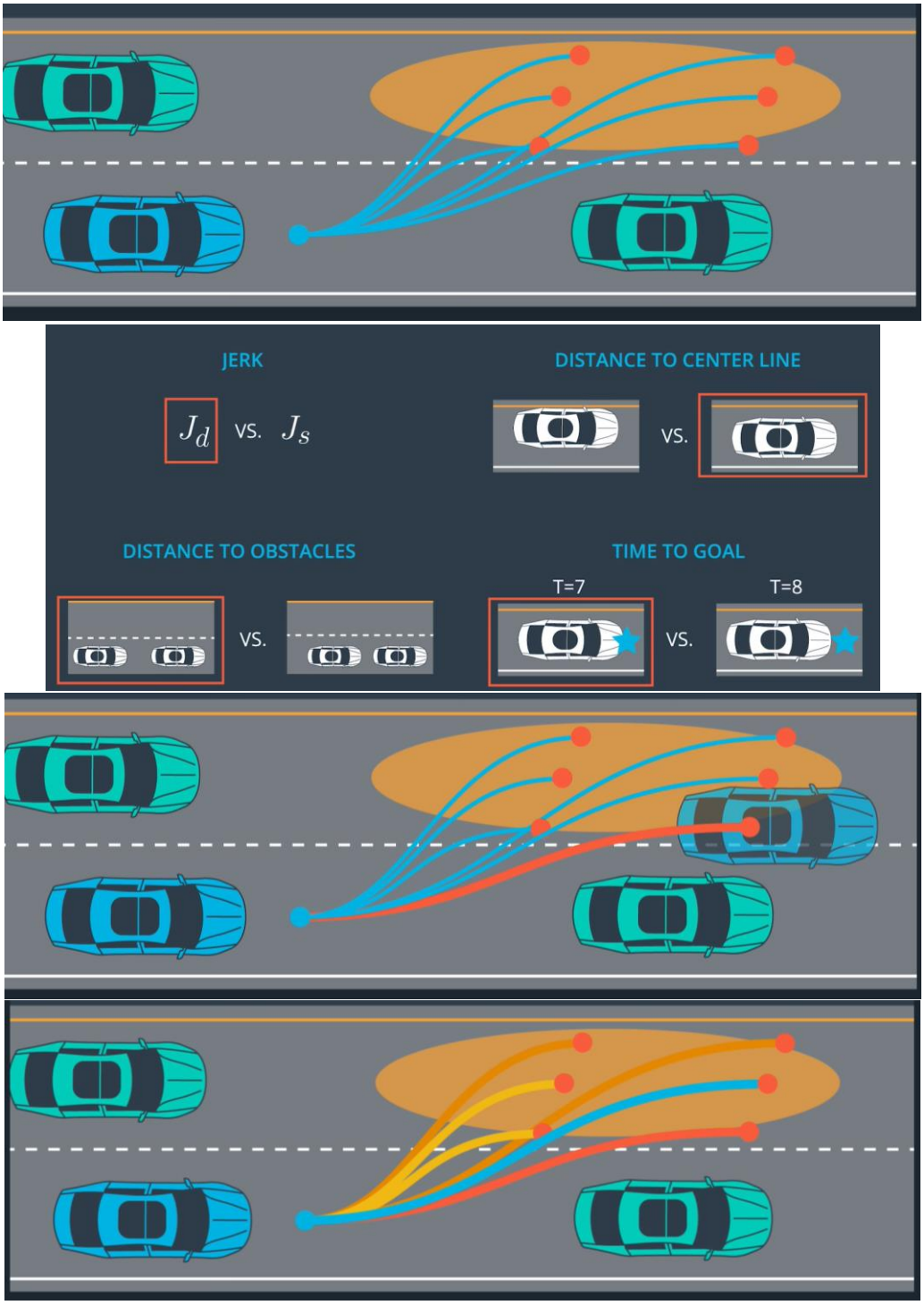
Longitudinal Speed = $\dot{s}$

$$v_{\text{MIN}} < \dot{s} < v_{\text{SPEED LIMIT}}$$

$$\tan\delta = \frac{L}{R} \rightarrow \kappa = \frac{\tan(\delta)}{L} \rightarrow \kappa_{\text{MAX}} = \frac{\tan(\delta_{\text{MAX}})}{L}$$







In reality, a real self-driving car usually uses different path planners depending on the situation.

On parking lots, hybrid A*

On low traffic highways, polynomial trajectory generation

…