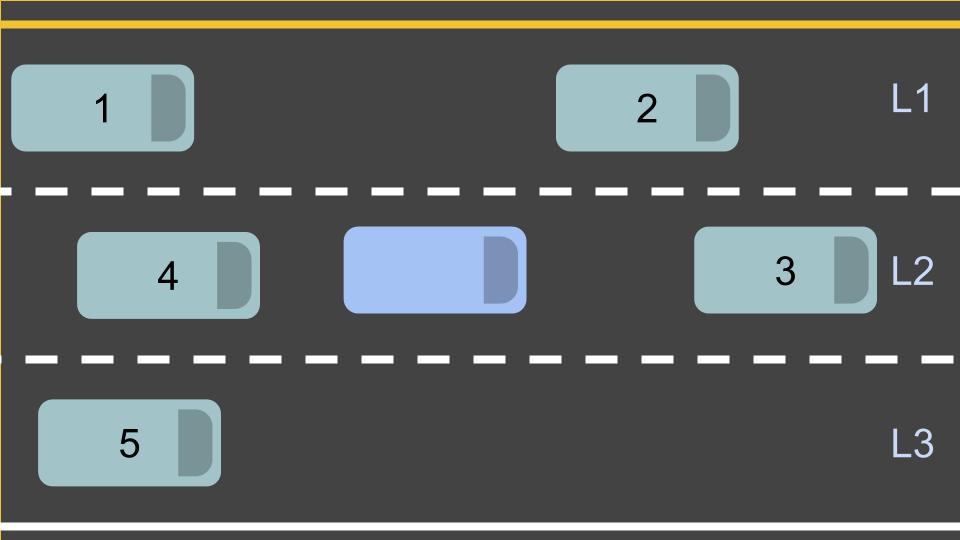
**Understanding Output**

It's possible to suggest a wide variety of behaviors by specifying only a few quantities. For example by specifying only a target lane, a target vehicle (to follow), a target speed, and a time to reach these targets, we can make suggestions as nuanced as "stay in your lane but get behind that vehicle in the right lane so that you can pass it when the gap gets big enough."

Look at the picture below and 5 potential json representations of output and see if you can match the json representation with the corresponding verbal suggestion.



**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,

}

# 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]

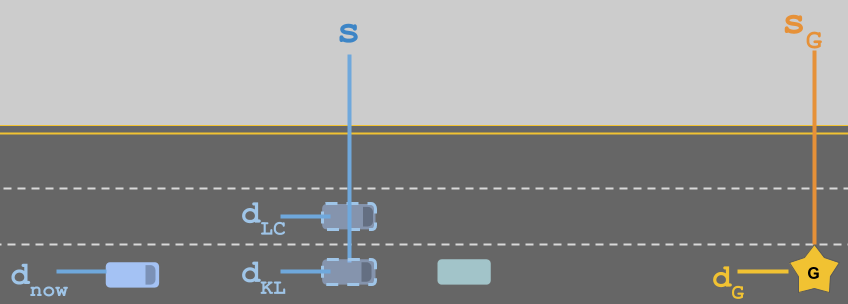
**if** cost < min\_cost:

min\_cost = cost

best\_next\_state = state

**return** best\_next\_state

Obviously we are glossing over some important details here. Namely: what **are** these cost functions and how do we create them? We'll talk about that next!



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 - sΔ*s*=*sG*​−*s* how much distance the vehicle will have before it has to get into the goal lane.
* \Delta d = d\_G - d\_{LC/KL}Δ*d*=*dG*​−*dLC*/*KL*​ the lateral distance between the goal lane and the options being considered. In this case \Delta d\_{KL} = d\_G - d\_{KL}Δ*dKL*​=*dG*​−*dKL*​ would be zero and \Delta d\_{LC} = d\_G - d\_{LC}Δ*dLC*​=*dG*​−*dLC*​ would not.

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

So we want a cost function that penalizes large |\Delta d|∣Δ*d*∣ and we want that penalty to be bigger when \Delta sΔ*s* 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**: \text{cost} = |\Delta d| + \frac{1}{\Delta s}cost=∣Δ*d*∣+Δ*s*1​

**Option 2**: \text{cost} = \frac{|\Delta d|}{\Delta s}cost=Δ*s*∣Δ*d*∣​

**Option 3**: \text{cost} = 1 - e^{- \frac{|\Delta d|}{\Delta s}}cost=1−*e*−Δ*s*∣Δ*d*∣​

In this example, we found that the ratio \LARGE \frac{|\Delta d|}{\Delta s}Δ*s*∣Δ*d*∣​ was important. If we call that ratio x*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 article](https://en.wikipedia.org/wiki/Sigmoid_function) if you're interested.

In the previous quizzes, you designed a cost function to choose a lane when trying to reach a goal in highway driving:

\text{cost} = 1 - e^{- \frac{|\Delta d|}{\Delta s}}cost=1−*e*−Δ*s*∣Δ*d*∣​

Here, \Delta dΔ*d* was the lateral distance between the goal lane and the final chosen lane, and \Delta sΔ*s* 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 \Delta sΔ*s* distance to the goal.
* The goal lane.

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

* |\Delta d|∣Δ*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.

You can implement your solution in cost.cpp below.

Here is one possible solution for the previous quiz:

**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.*

**int** delta\_d = 2.0 \* goal\_lane - intended\_lane - final\_lane;

**double** cost = 1 - exp(-(std::abs(delta\_d) / distance\_to\_goal));

**return** cost;

}

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.

One possible solution for the previous quiz is the following:

**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.*

**double** speed\_intended = lane\_speeds[intended\_lane];

**double** speed\_final = lane\_speeds[final\_lane];

**double** cost = (2.0\*target\_speed - speed\_intended - speed\_final)/target\_speed;

**return** cost;

}

# Cost Function Matching

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

##### Cost Function 1

\begin{cases} 1 & \ddot{s} \geq a\\_{\text{max}} \\ 0 & \ddot{s} < a\\_{\text{max}} \end{cases}{10​*s*¨≥*a*\_max*s*¨<*a*\_max​

##### Cost Function 2

\begin{cases} 1 & d \geq d\\_{\text{max}} \\ 1 & d \leq d\\_{\text{min}} \\ 0 & d\\_{\text{min}} < d < d\\_{\text{max}} \end{cases}⎩⎪⎨⎪⎧​110​*d*≥*d*\_max*d*≤*d*\_min*d*\_min<*d*<*d*\_max​

##### Cost Function 3

\begin{cases} 1 & \dot{s} \geq v\\_{\text{speed limit}} \\ 0 & \dot{s} < v\\_{\text{speed limit}} \end{cases}{10​*s*˙≥*v*\_speed limit*s*˙<*v*\_speed limit​

##### Cost Function 4

\frac{1}{1 + e^{-(d - d\\_{\text{lane center}})^2}}1+*e*−(*d*−*d*\_lane center)21​

##### Cost Function 5

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

# Implement a Behavior Planner

In this exercise you will implement a behavior planner and cost functions for highway driving. The planner will use prediction data to set the state of the ego vehicle to one of 5 values and generate a corresponding vehicle trajectory:

* "KL" - Keep Lane
* "LCL" / "LCR"- Lane Change Left / Lane Change Right
* "PLCL" / "PLCR" - Prepare Lane Change Left / Prepare Lane Change Right

The objective of the quiz is to navigate through traffic to the goal in as little time as possible. Note that the goal lane and s value, as well as the traffic speeds for each lane, are set in main.cpp below. Since the goal is in the slowest lane, in order to get the lowest time, you'll want to choose cost functions and weights to drive in faster lanes when appropriate. We've provided two suggested cost functions in cost.cpp.

### Instructions

1. **Implement the choose\_next\_state method in the vehicle.cpp class.** You can use the [Behavior Planning Pseudocode concept](https://classroom.udacity.com/nanodegrees/nd013/parts/6047fe34-d93c-4f50-8336-b70ef10cb4b2/modules/27800789-bc8e-4adc-afe0-ec781e82ceae/lessons/56274ea4-277d-4d1e-bd95-ce5afbad64fd/concepts/e9f08f5f-0b8f-488d-8940-45bc474b4913) as a guideline for your implementation. In this quiz, there are a couple of small differences from that pseudocode: you'll be returning a best trajectory corresponding to the best state instead of the state itself. Additionally, the function inputs will be slightly different in this quiz than in the classroom concept. For this part of the quiz, we have provided several useful functions:
   1. successor\_states() - Uses the current state to return a vector of possible successor states for the finite state machine.
   2. generate\_trajectory() - Returns a vector of Vehicle objects representing a vehicle trajectory, given a state and predictions. Note that trajectory vectors might have size 0 if no possible trajectory exists for the state.
   3. calculate\_cost() - Included from cost.cpp, computes the cost for a trajectory.
2. **Choose appropriate weights for the cost functions in cost.cpp to induce the desired vehicle behavior.** Two suggested cost functions have been implemented based on previous quizzes, but you are free to experiment with your own cost functions. The get\_helper\_data() function in cost.cpp provides some preprocessing of vehicle data that may be useful in your cost functions. See if you can get the vehicle to move into a fast lane for a portion of the track and then move back to reach the goal!
3. Hit Test Run and see how your car does! How fast can you get to the goal?

## Extra Practice

Provided in one of the links below is a zip file python\_3\_practice, which is the same problem written in Python 3 - you can optionally use this file for additional coding practice. In the python\_3\_solution link, the solution is provided as well. If you get stuck on the quiz see if you can convert the python solution to C++ and pass the classroom quiz with it. You can run the python quiz with python simulate\_behavior.py.