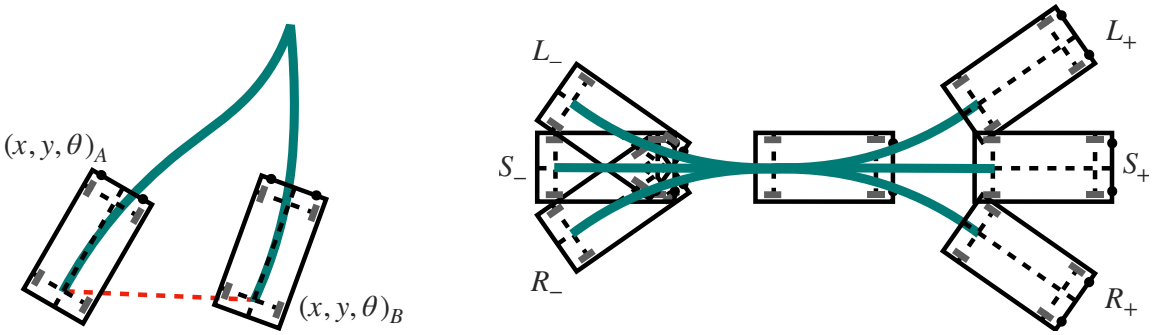
Homework #7
Friday 2/23/23/24 11:59pm

For the last regular homework set, I would like to plan for a car, include the classic parallel parking problem. I would like us to see how the various algorithms and concepts are meant to be a starting point and are, in practice, often adjusted to best suit a problem's needs.

For planning a car's movements, in particular, we can not just plan arbitrary movements, but have to obey/incorporate the steering limitations. So we can not just connect two states A and B with a straight line (shown in dashed red on left):



Connecting two states requires solving the boundary value problem (BVP) to determine the combination of arcs and straight lines required (shown in solid greed on left). To avoid this, we are not going to use RRT, PRM, or even A*.

Instead we are going to use an algorithm halfway between EST and Dijkstra's. Like an EST, we are going to grow a tree in the continuous state space. But with three modifications:

(1) We are going to grow the tree, not in arbitrary directions, but based on six possible actions. From each node, we will drive the car forward or backward, steering left, straight, or right (shown on right). This is an initial value problem (IVP) and much simpler to code.

(2) We are going to overlay a 3D grid on the 3D state space. Two nodes that fall into the same grid cell are considered to reach the same space and we will only save one of them. This means the search tree will evolve with a uniform density of exactly one node per cell. We will also size the actions/grid such that each action (approximately) moves from one grid cell to a neighbor.

(3) Like Dijkstra's we are going to use a *cost-to-reach* and *ondeck queue* to determine from which leaf the tree will continue to grow. As such, we will not only find a solution, but will be able to shape the solution based on the selection of costs.

The algorithm is shown on the next page. It is effectively Algorithm 22 CAR_GRID_SEARCH in "Principles of Robot Motion: Theory, Algorithms, and Implementations." This may not be as efficient as an RRT (at reaching the goal), but is much faster to code.

You will notice a structure like Dijkstra's algorithm, including the sorted ONDECK queue, marking a node DONE when fully processed, and adding child nodes to the ONDECK queue if the space hasn't been seen or the cost is lower. But you'll also see the GRID being used to determine whether a space has been seen, the use of actions to determine children, and the need to check whether a path is clear.

Grid-Based Car Planning Algorithm:

Given *startnode* and *goalnode*
Start with an empty ONDECK queue and GRID
Place the *startnode* in the ONDECK queue and save in GRID
while ONDECK is not empty **do**
  Pop *node* from the ONDECK queue (with the lowest cost)
  Mark *node* as done
  if *node* is in same grid cell as *goalnode* **then**
    Traverse back from *node* via parents to *startnode* to identify *path*
    Return SUCCESS and *path*
  end if
  for all possible actions $L_{\pm}$, $S_{\pm}$, $R_{\pm}$ **do**
    Integrate *node* forward in time to a *child* node (saving the parent and action to get there)
    if *child* in same grid cell as *node* **then**
      Integrate *child* forward in time again for a new/replacement *child* (same action)
    end if
    if *child* and path between *node* and *child* are without collisions **then**
      if *child*'s grid cell is empty **then**
        Save *child* in GRID and place ONDECK
      else
        read *prev* node from the *child*'s grid cell
        if *prev* is not done **and** *child*'s cost is lower than *prev*'s cost **then**
          Remove *prev* from ONDECK and GRID
          Save *child* in GRID and place ONDECK
        end if
      end if
    end if
  end for
end while
Return FAILURE

To keep things manageable, we are again providing skeleton code `carplanner_skeleton.py` (in `hw7democode.zip`), very similar to the A* and EST code in homework sets 3 and 4. As before, it uses the standard Shapely library to check collisions between the car (box) and the walls (line segments). And it contains the sections:

1. Parameter definitions. You will want to *change the scenario and the costs settings* as you explore in Problems 4-6.
2. A visualization utility class.
3. The Node class. Add the *forward simulation* (IVP) in Problem 1, and the *cost* in Problem 2.
4. The actual planner code. You will need to *write the planner* in Problem 3.
5. The test/main code. Disable the test functions after complete Problems 1 and 2.

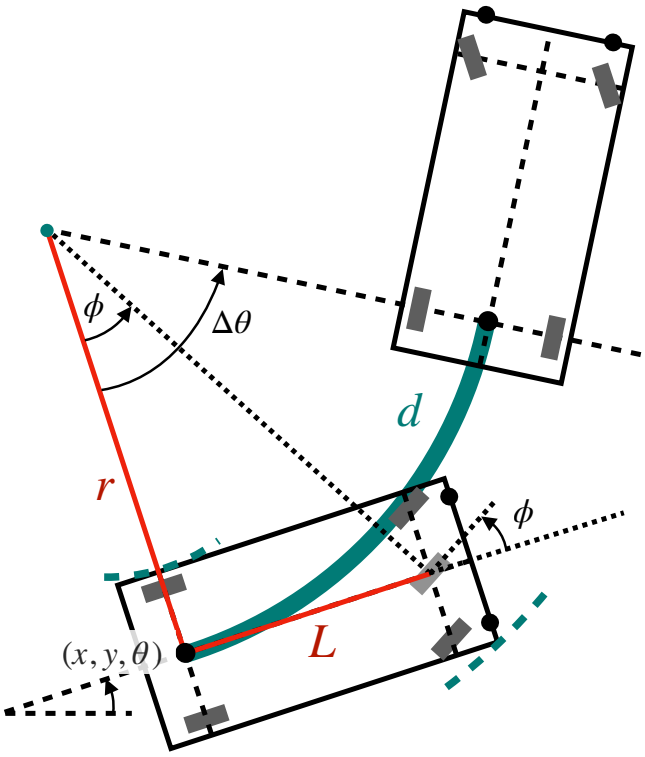
Please read through the code. The four places that need editing are marked with "FIXME".

Problem 1 (Forward Simulation) - 24 points:

Consider the car as a box 4m long, 2m wide, with front and read axles 0.5m from their respective bumpers. That means, the car has a wheelbase (front to rear axle) of 3m. We'll add headlights in the visualization to show the direction.

The car's states (x, y, θ) are the position of the center of the rear axle and the orientation angle w.r.t. the x axis. Note the orientation wraps at 360°, that is two cars pointing 10° and 370° are in the same orientation. The steering angle ϕ is positive turning left, negative turning right.

We begin by coding the initial value problem. Assume the car is at some starting state (x_A, y_A, θ_A). As we drive, where will the car end up?



We note the steering angle ϕ relates the turning radius r and the wheelbase L. And considering the distance traveled (arc length) d, we can thereby relate the steering angle to the change in heading

$$\tan(\phi) = \frac{L}{r} \qquad \text{and} \qquad d = r\Delta\theta \qquad \Longrightarrow \qquad \Delta\theta = \frac{d}{r} = \frac{d\ \tan(\phi)}{L}$$

In the following, we are going to limit ourselves to six possible actions $\mathrm{L}_\pm$, $\mathrm{S}_\pm$, $R_\pm$: driving forward/backward a fixed distance, while turning left/straight/right. That is

$$d \in \{+d_0, -d_0\} \qquad \text{and} \qquad \phi \in \{+\phi_0, 0, -\phi_0\}$$

From the above, instead of specifying the steering angle, we can alternatively specify the change in heading. This means we can **size the actions to match the spacing of the grid** we plan to overlay: d_{step} in translation x and y, θ_{step} in heading θ.

$$d \in \{+d_{\text{step}}, -d_{\text{step}}\} \qquad \text{and} \qquad \Delta\theta \in \{+\theta_{\text{step}}, 0, -\theta_{\text{step}}\} \qquad \text{making} \quad \phi_0 = \text{atan}\left(\frac{L\ \theta_{\text{step}}}{d_{\text{step}}}\right)$$

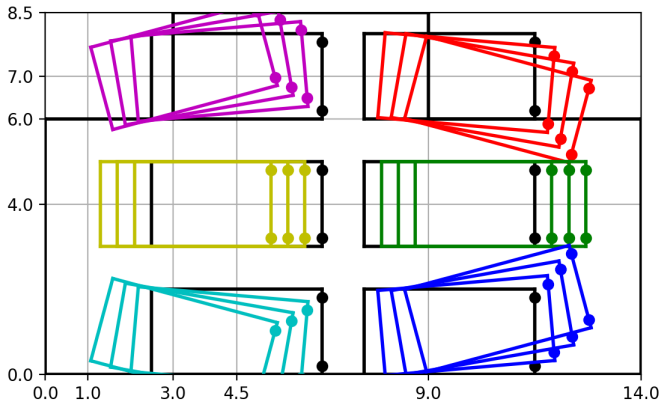
(a) Assume a starting state (x_A, y_A, θ_A). For each of the six possible actions $\mathrm{L}_\pm$, $\mathrm{S}_\pm$, $R_\pm$, that is with $d \in \{+d_{\text{step}}, -d_{\text{step}}\}$ and $\Delta\theta \in \{+\theta_{\text{step}}, 0, -\theta_{\text{step}}\}$, what is the ending state (x_B, y_B, θ_B)?

Note, as we have assumed a travel distance, we do not need to know speeds or time. You may answer this with a single formula or with different cases.

(b) Please code the above. In the `Node` class, you will see a `nextNode(self, forward, steer)` function. Assume `forward` is either +1 or -1, and `steer` is either +1, 0, or -1. Note `steer` is the sign of the steering angle ϕ, *not* of the heading change $\Delta\theta$. You can read the global variables `dstep` and `thetastep` to determine the step sizes.

The function should return a new `Node()` instantiated at the ending coordinates. Please submit the code for this function.

The main code is set up to test this function via `testdriving()`. Running the code, you should see



showing three consecutive steps for each of the six possible actions. The top row has a negative `steer`, the right side a positive `forward`. Please submit a screenshot to confirm the functionality.

Problem 2 (Cost To Reach) - 16 points:

To evaluate and prioritize different possible plans, we compute a cost to reach each node as

$$c = c_{\text{step}}\ N_{\text{step}} + c_{\text{steer}}\ N_{\text{steeringAnglesChanges}} + c_{\text{reverse}}\ N_{\text{reversals}}$$

where

N_{step} is the number of steps or movements taken. This makes the total path length $d_{\text{step}} N_{\text{step}}$.

$N_{\text{steeringAnglesChanges}}$ is the number of times a steering angle is changed.

$N_{\text{reversals}}$ is the number of times the forward/backward direction of movement is reversed.

Please update the above `nextNode(self, forward, steer)` function to set the new (child) node's `cost` considering the starting node's `cost` and the changes in `forward` and `steer`.

If you edit the `main()` function, you can test this change via `testcosts()`. This executes 8 successive actions: S_+, S_+, L_+, L_+, R_+, R_+, S_-, S_-. Assuming the default costs of

$$c_{\text{step}} = 1 \qquad c_{\text{steer}} = 10 \qquad c_{\text{reverse}} = 100$$

you should see the output

```
<XY  4.00, 4.00 @   0.0 deg> (fwd  1, str  0, cost    0)    = starting node
<XY  4.40, 4.00 @   0.0 deg> (fwd  1, str  0, cost    1)    = S+
<XY  4.80, 4.00 @   0.0 deg> (fwd  1, str  0, cost    2)    = S+
<XY  5.20, 4.02 @   5.0 deg> (fwd  1, str  1, cost   13)    = L+
<XY  5.60, 4.07 @  10.0 deg> (fwd  1, str  1, cost   14)    = L+
<XY  5.99, 4.12 @   5.0 deg> (fwd  1, str -1, cost   25)    = R+
<XY  6.39, 4.14 @   0.0 deg> (fwd  1, str -1, cost   26)    = R+
<XY  5.99, 4.14 @   0.0 deg> (fwd -1, str  0, cost  137)    = S-
<XY  5.59, 4.14 @   0.0 deg> (fwd -1, str  0, cost  138)    = S-
```

matching 8 steps, 3 changes in steering, and 1 reversal for a total cost of 138. Please submit the updated `NextNode()` code.

Problem 3 (Planner Code) - 24 points:

Please code the above algorithm in the `planner(startnode, goalnode)` function and submit.

In the code note that

```
ind = node.indices()
```

determines the `node`'s indices into the `grid`. So we can test whether the matching cell is empty

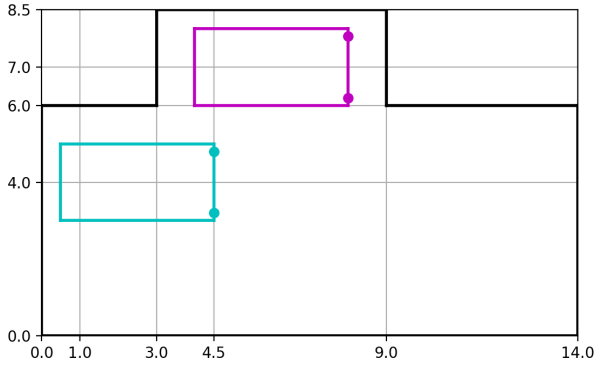
```
if grid[ind] is None:
```

or retrieve the node associated with the cell, or save the current node into the grid.

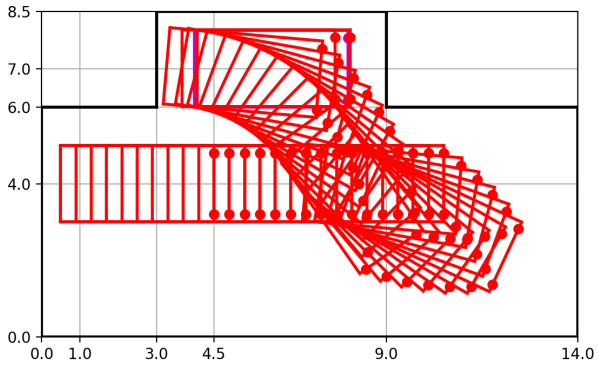
```
prevnode = grid[ind]
grid[ind] = node
```

Problem 4 (Parallel Parking) - 16 points:

To debug and test the code, please plan the parallel parking problem. The skeleton code is set up to do just this. The start and final states, as well as the parking spot, are as shown on here and given in the code.



Using the default costs, the final path should look like



Note, for me, this sampled nearly 45,000 nodes and took about 10 seconds. We can see that the planner is not particular fast. Indeed, I would consider this a brute-force breadth first search. To improve and guide the search, as A* improves Dijkstra's, would require solving the BVP and more coding.

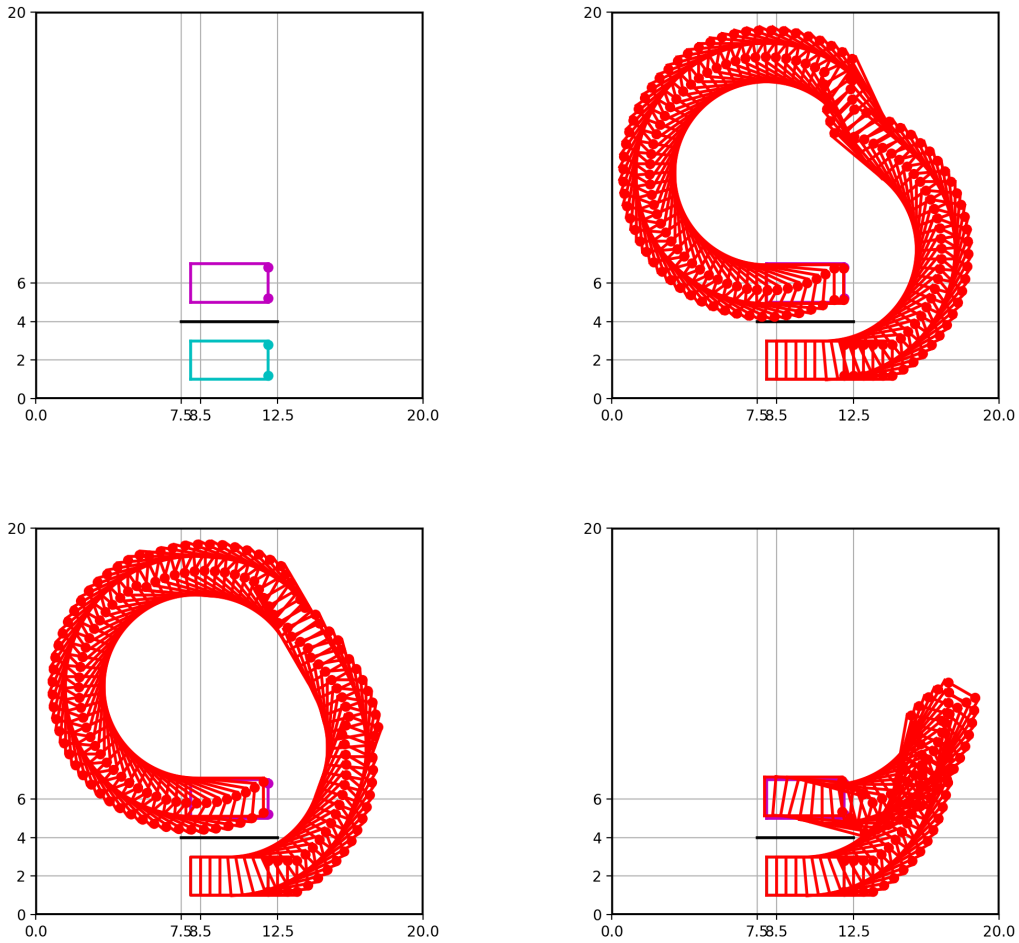
Besides the default cost, change the cost to determine the shortest possible path. What is the lowest number of steps to reach the goal? Please submit a screenshot.

Problem 5 (Changing Cost - Moving Over) - 16 points:

Now that we have a functional planner, let's try moving the car over one parking spot. In the code, change the `scenario` at the very beginning to `'move over'`.

This sets up the problem as shown in the top left. I found multiple different strategies for moving the car. The solution using default cost is shown in the top right.

Note, with the larger space and grid, this sampled anywhere between 50,000 and 150,000 nodes, taking two to three times as long.



(a) What cost values smooth out the path, using more straight segments (shown in lower left)?

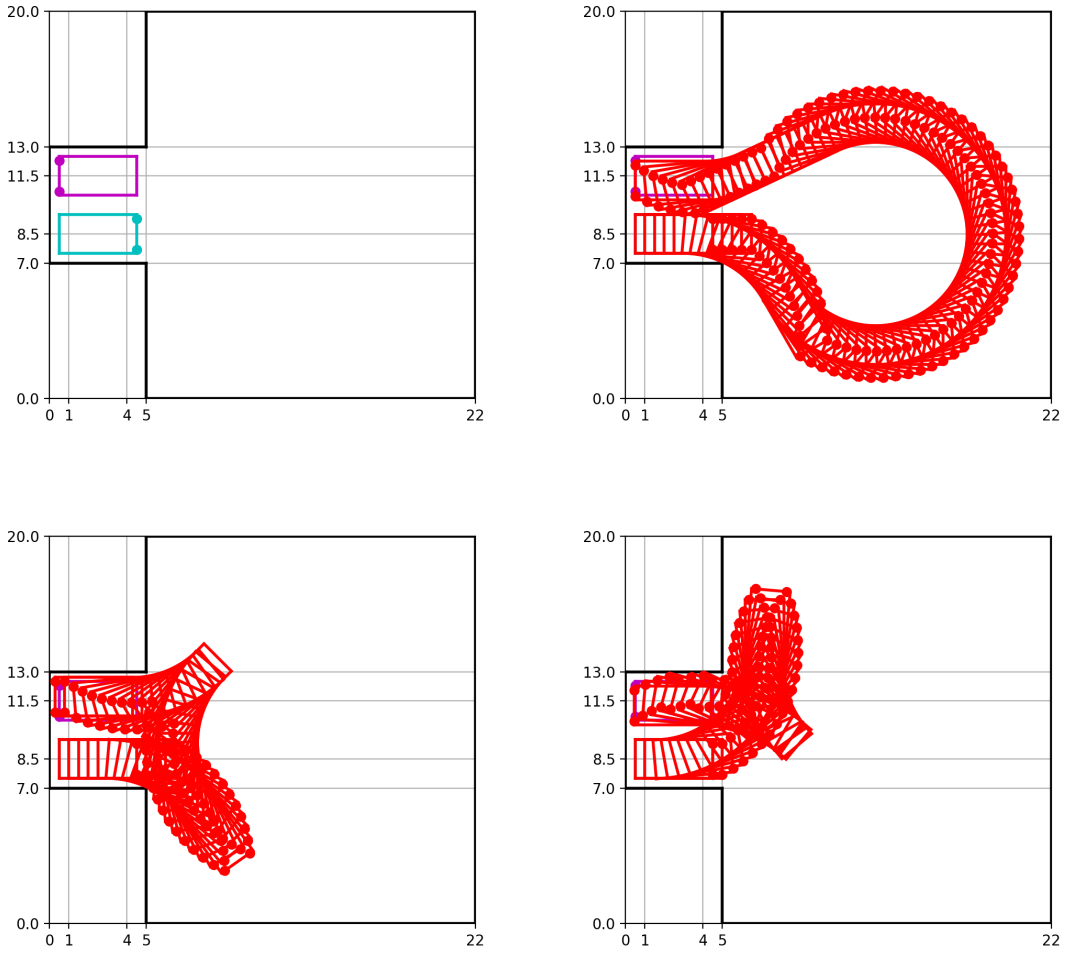
(b) What costs values allow the car to pull forward and then back up into the neighbor spot?

Please submit screenshots together with the cost values.

Problem 6 (OPTIONAL U Turn) - 8 points:

Finally, only if you are still having fun, try a U turn. In the code, change the `scenario` at the very beginning to `'u turn'`. This sets up the problem as shown in the top left. Again I found different solutions, the default again shown in the top right.

Note, this takes even longer. I saw between 120,000 and 400,000 nodes sampled, taking possibly over a minute!



If you try this, what cost levels does it require to create a three point turn (either bottom left or bottom right)?

Problem 7 (Time Spent) - 4 points:

One last time, may we ask how much time you spent on on this homework? And whether you have any special instructions for any possible penalties, current or past.