ME41105 - IV Assignment 3

Motion planning

Dariu Gavrila, Julian Kooij
Ewoud Pool, András Pálffy, Joris Domhof, Thomas Hehn

Intelligent Vehicles group
Delft University of Technology

December 6, 2018

TU Delft

About the assignment

Make the assignments in student pairs, you receive both one grade. Please read through this whole document first such that you have an overview of what you need to do.

This assignment contains *Questions* and *Exercises*:

- You should address all of the questions in a 2 or 3 page report (excluding plots and figures). *Please provide separate answers for each Question in your report, using the same Question number as in this document.* Your answer should address all the issues raised in the Question, but typically should not be longer than a few lines.
- The Exercises are tasks for you to do, typically implementing a function or performing an experiment. Threrefore, first study the relevant provided code before working on an exercise, as the code may always contains comments refering to each specific exercise. *If you do not fully understand the exercise, it may become more clear after reading the relevant code comments!* Do not directly address the exercises in your report. Instead, you should submit your solution code together with the report. Experimental results may be requested in accompanying questions.

You will be graded on:

1. Quality of your answers in the report: Did you answer the Questions correctly, and demonstrate understanding of the issue at hand? All Questions are weighted equally.
2. Quality of your code: Does the code work as required?
3. Quality of presentation: Is your report readable (sentences easy to understand, no grammar mistakes, clear figures)? Is the code you wrote clear and commented?

Submitting

- Before you start, go to the course's Brightspace page, and enroll with your partner in a lab group (found under the 'Collaboration' page).
- To submit, upload **two** items on the Brightspace 'Assignments' page:

pdf attachment	A *pdf* with your report. Do not forget to add your student names and ids on the report.
zip attachment	A *zip* archive with your Matlab code for this assignment Do NOT add the data files! They are large and we have them already ...
deadline	**Saturday, December 22 2018, 23:59**

- Note that only a single submission is required for your group, and only your last group submission is kept by Brightspace. **You are responsible for submitting on time**. So, do not wait till the last moment to submit your work, and *verify* that your files were uploaded correctly. Connection problems and forgotten attachments are not a valid excuses. The due deadline is automatically enforced by Brightspace. If your submission is not on time, you receive an automatic '1'.
- You may only hand in work by you and your lab partner, done this year. You are responsible for not sharing your work neither publicly, nor privately with other students outside your group. Do not put your code or report on public servers. If we believe that you have

used material from other groups, or that you have submitted material that is not yours, it will be reported to the exam committee. This may ultimately result in a failing grade or an expulsion.

- If code is submitted that was written with ill intent, e.g. to manipulate files in the users home directory that were not specified by the task, you will immediately fail the course.

Getting assistance

The primary occasion to obtain help with this assignment is during the lab practicum contact hours of the Intelligent Vehicles course. An instructor and student assistants will be present at the practicum to give you feedback and support. If you find errors, ambiguously phrased exercises, or have another question about this lab assignment, please use the Brightspace lab support forum. This way, all students can benefit from the questions and answers equally. If you cannot discuss your issue on the forum, please contact Julian Kooij (J.F.P.Kooij@tudelft.nl) directly.

Remember that for help on what a specific Matlab command `somefunction` does or how to use it, use can type from the Matlab command line `help somefunction`, or `doc somefunction`.

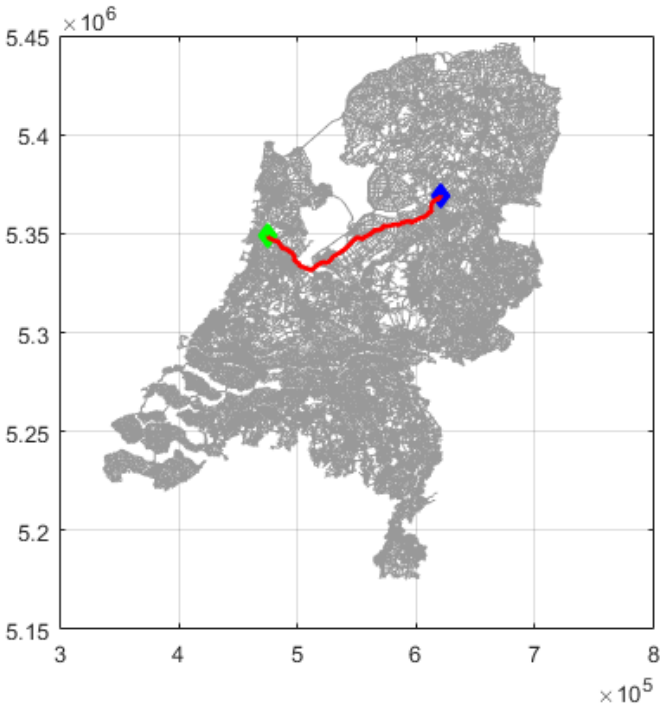

Figure 1: Road map route planning.

Motion planning

In this last lab session we will look at various motion planning techniques. The first two parts concern graph planning. The third part addresses trajectory optimization and obstacle avoidance.

1 Long-term route planning

Exercises refer to code sections in `assignment_nl_graph_planning.m`.

In this assignment you will implement the $A^\star$ path planning algorithm, and use it for long-term route planning in a graph of the road network in The Netherlands. The graph is represented as V vertices, connected with E edges. Each vertex has an associated 2D locations in the map, the edges are straight road pieces that connect the vertex locations. Figure 1 illustrates the road map, and a path planning problem consisting of moving from the blue marker to the green one.

The provided code contains a bare-bone implementation of a basic best-first search algorithm from Appendix A. Here, vertices are referred to by their index, which is an integer $1 \leq v \leq V$. Given a start and goal vertex index, it searches for a shortest path through the graph. Here, the 'shortest' path is the one with minimum length, hence Euclidean distance $g(v_1, v_2)$ is used as cost function. In practice, route planners could use other cost functions such as 'fastest path', using expected travel time instead of distance as a cost function, e.g. by preferring highways over small inner-city roads. We shall not make distinction between types of roads, and just try to find the shortest one for now. The algorithm from Appendix A is *not* (yet) the $A^\star$ algorithm,

as it does not use a heuristic to explore vertices closer to the goal first. Instead, it simply selects the vertex from the frontier which currently has the shortest path length.

Question 1.1. Is it generally true that for any graph, the path with shortest distance is also the one with the least number of vertices? If yes, give a proof. If no, give a counter example of a simple graph where a shortest path does not have the least number of vertices.

Load the map, and visualize the road network. Note that this can take a while to load. The code also provides various planning problems, each defining a different start and goal vertex. For the moment, stick with the first problem which has the start and goal vertex relatively close together. The found shortest path should have a length of 42.912 km.

Exercise 1.1. Implement the backtracking step in `search_shortest_path` to complete the graph search algorithm from Algorithm 1 (see Appendix A). In this step, we create the shortest path by iteratively following the back pointers from goal to start.

If everything is correct, you should see that the algorithm explored vertices in a circular shape around the start position.

Question 1.2. Explain in words why this explored area is (almost) circular. In your explanation, address in what order are the vertices in this area have been explored, and what determines approximately the radius of the area.

Now we move from the basic best-first algorithm to the $A^\star$ algorithm described in Appendix B, using the heuristic $h(v, goal)$ to estimate the remaining distance of a vertex v to the goal vertex $goal$. For h, we can use the *same* Euclidean distance function that is used as the cost function g. Conceptually, this heuristic underestimates the remaining distance from the vertex to the goal as if there would be a straight road. Clearly, this heuristic is an optimistic estimate of the true remaining distance. Carefully compare the pseudo-code in Algorithms 1 and 2 to see what will change.

Exercise 1.2. Implement the missing code in `search_shortest_path_astar` to complete the $A^\star$ algorithm, by adding the heuristic function h as part of the scoring function. You also need to add the backtracking step again, as in the previous exercise. Rerun the same route planning problem as before, you should find the same shortest path length. You can now also try the other planning problems to verify that your implementation is working.

Question 1.3. What are the shortest route distances (in kilometers) that you found for each of the given planning problems? For the first path planning problem, how many vertices needed to be *explored* by Algorithms 1 and 2 to find the shortest path?

Question 1.4. Is the benefit from using the $A^\star$ over the basic best-first search the same for each path planning problem? If not, in what kind of problems is the benefit the greatest?

2 Planning to park

Exercises refer to code sections in `assignment_parking_planning.m`.

Finding the shortest path in a graph is not only useful for planning a route in a road network, but can also be used for more fine-grained planning tasks. In this exercise you will apply your

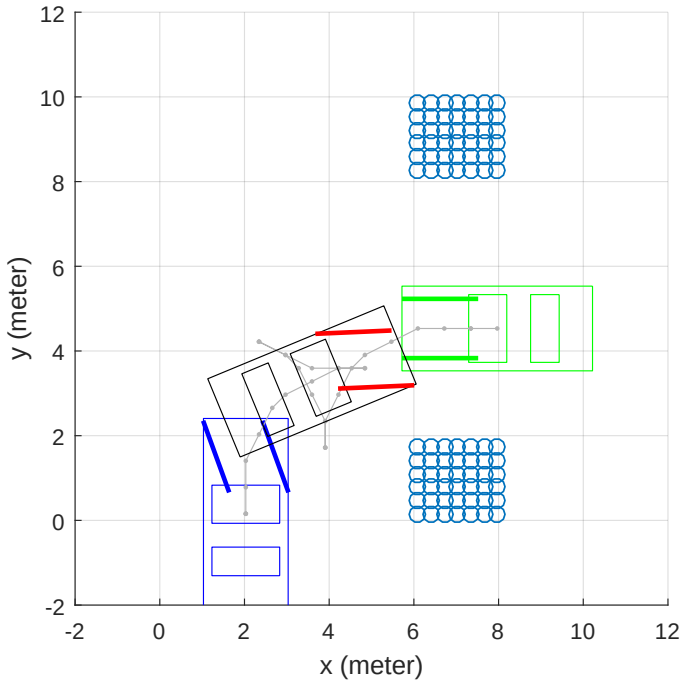

Figure 2: Parking a vehicle in a small space. The blue vehicle shows the start position, the green one the goal. The objective is to determine a physically feasible path to reach the goal from the start position. The gray dotted line shows the discovered path using $A^\star$. The gray vehicle shows an intermediate vehicle position along the path.

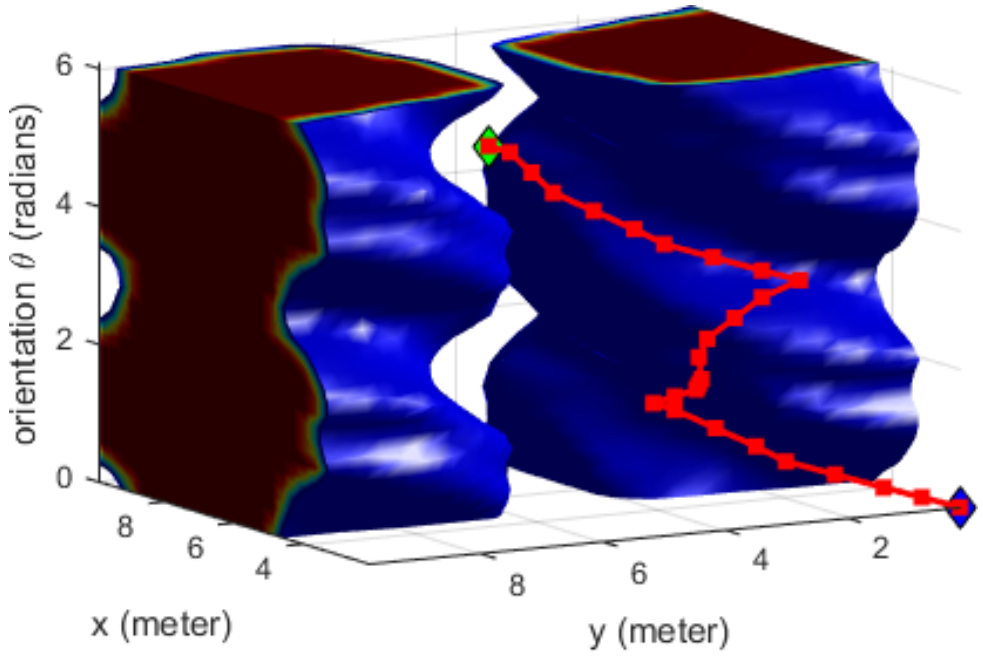

Figure 3: Parking a vehicle in a small space can be seen as a path planning problem in the configuration space.

$A^\star$ implementation from Section 1 to automatic parking around static obstacles. This can be done by formulating the parking problem as a path-planning problem in a discrete configuration space, see Appendix C. The vehicle state shall be represented by a four dimensional vector $\mathbf{x} = [x, y, \theta, \omega]^\top$. Here (x, y) is the vehicle's 2D groundplane position, θ its orientation, and ω the angles of the wheels that determine the change in orientation when moving forward or backward.

For cell decomposition, the configuration space is discretized into a four dimensional grid with 32 spatial x bins, 32 spatial y bins, 32 bins for θ in the $[0, 2\pi]$ domain. For ω we have three bins, corresponding to wheels turned left, center, and turned right. We therefore obtain a grid with $V = 32 \times 32 \times 32 \times 3 = 98304$ cells/vertices.

In this parking scenario, each cell in the discretized configuration space has four possible edges, which correspond to four moves that the vehicle can take, namely (1) move forward, (2) move backward, (3) turn steer/wheels left, (4) turn steer/wheels right. Note that some vertices can have less edges, e.g. the vehicle is not allowed to move outside the spatial bounds of the represented configuration space, and the wheel cannot be turned beyond its physical bounds.

There are two visualizations available, namely a top-down view showing the vehicle in the 2D groundplane (including wheel angle), see Figure 2, and a view of the 3D configuration space (x, y, θ) (which ignores the steering angle ω in the fourth dimension!), see Figure 3. A vehicle state is represented as a point in this configuration space.

> *Exercise 2.1.* Run the code block to visualize the free and occupied space in 2D and 3D, and show the vehicle state given by a vertex index `idx`. The code shows that this vertex index is obtained by selecting a cell in the 4D grid space. Try changing the 4D cell coordinates by picking other values for each of the four dimensions. Note that each combination results in a different vertex index, and also how the plotted 2D and 3D vehicle state changes.

> **Question 2.1.** You should see some curved 3D structures similar to those in Figure 3. What do these structures represent, and why do they have these 'twisted' shapes? If you change the vehicle cell such that the diamond marker disappears into the twisted shapes, what can you see happening in the 2D ground plane plot?

> *Exercise 2.2.* Run the $A^\star$ algorithm on the given path planning problems. Here the Euclidean distance is again used for both cost function g and heuristic h, but this time it is computed on the 4D state vectors in the configuration space (we ignore the fact that the orientation angle is circular here, so the distance between $\theta = 0$ and $\theta = 2\pi$ is 2π instead of 0).

Currently, the path planning does not take the 'occupied' or 'obstructed' space into account yet. Therefore, we should remove edges from 'unoccupied' vertices (corresponding 'unobstructed' vehicle state) to the occupied vertices. We can do this by updating the lists of vertices in the `reachable` cell array.

> *Exercise 2.3.* Remove the inaccessible states from the lists of reachable vertices, by completing the code in `remove_unreachable_cells`, and adding a call to `remove_unreachable_cells` to your main script. Rerun the script from the start of the end. The path planner should now avoid colliding the vehicle with the obstacles.

Notice that sometimes the vehicle still moves in a physically unrealistic way: When the planned vehicle path moves horizontally, the vehicle orientation is not always horizontal too. You will probably see this effect occurring in `planning_problem_idx = 2`.

Question 2.2. Explain why this problem occurs. Would increasing the number of spatial bins for x and y improve the relation between the vehicle orientation and motion direction? Why? And how about if we instead increase the number of bins for orientation θ, would that improve the motion? Why?

3 Trajectory planning and obstacle avoidance

Consider that our vehicle has planned its route over the map (first part of Assignment 3), and is currently driving along the road. We assume that the vehicle is capable of localizing itself with respect to the environment (see previous exercise on self-localization from Lab 2), and can detect other moving objects (see detection assignment from Lab 1) and track them (see multi-object tracking assignment from Lab 2). The vehicle can also estimate the future positions of the tracked objects by extrapolating their current paths using constant velocity dynamics. In this final assignment the objective is now to determine the control input of our vehicle for the short-term (up to several seconds) such that it follows the lane, and avoids possible collisions with other objects. The approach in this assignment is based on the method described in [1].

Exercises refer to code sections in `assignment_trajectory_planning.m`.

The first task is to implement *trajectory optimization* to optimize the control input parameters to drive the vehicle to some goal state. Please read Appendix D. For this exercise the *longitudinal control* is considered to be fixed to constant velocity. We will therefore focus only on *lateral control*, i.e. the steering. This is expressed by a *steering profile* function which expresses the steering angle as a function of the traveled distance along the road segment, see Appendix D.1.

Exercise 3.1. Complete the code in `make_steering_profile`. It should return a steering control function, which uses a cubic spline to determine the steering angle given the fraction of traveled road (a number between 0 and 1). The spline should be defined by the control points k_0, k_1 and k_2, which represent the desired steering angles at 0%, 50% and 100% of the traveled longitudinal distance.

We use a non-linear dynamical model for the vehicle, similar to the non-linear dynamics that were used in the Particle Filter assignment earlier. The dynamics will be used to predict the future vehicle states given control parameters $\mathbf{p}$, see Appendix D.2.

Exercise 3.2. Complete the code in `euler_integrate_motion` by retrieving the steering control from the steering profile function, and updating the state with the motion model. The function should now create the state sequence following equations (6)-(8).

Exercise 3.3. Run the code block that generates the future vehicle states as a function of the steering control parameters $\mathbf{p} = [k_1, k_2, s_f]$. Manually change the three parameters to make the vehicle follow the road segment. Then, complete `curv_param_state_predict`, which computes $\mathbf{x}_T(\mathbf{x}_0, \mathbf{p})$. Once the implementation is

complete, set `USE_CURV_PARAM_STATE_PREDICT = true` and verify that the vehicle still follows the path defined by your selected control parameters.

Question 3.1. What values did find for k_1, k_2, and s_f through manual optimization? Give an intuitive explanation for these numbers.

You should by now have seen that by manually changing the parameters you can obtain different trajectories. Instead of manually adapting the control parameters such that the vehicle arrives at the desired position and orientation, the goal is to let the computer determine such parameters automatically. Therefore, we need to define an objective function which computes the error to minimize as a function of the control parameters. Then, the parameters are iteratively improved using a gradient descent method, which at each iteration slightly adjusts the parameters such that the error decreases, until there the improvement is marginal. Optimizing a non-linear objective function is classic optimization problem for which we can utilize existing functions provided by our programming environment. In Matlab we can use `lsqnonlin`, which stands for *non-linear least-squares* optimization.

Exercise 3.4. Implement the error vector $C(\mathbf{p})$ in `optimize_control_parameters`, which will be optimized with `lsqnonlin` from the Matlab Optimization Toolbox.

If you run the code block, you should see that the vehicle now follows the curved path.

Question 3.2. What happens with the trajectory if you remove the orientation θ_T from the error vector $C(\mathbf{p})$ to optimize? Is it important to include the orientation θ_T in a practical application? Why?

Now that the vehicle can determine the control parameters to steer to a future target position and orientation, we can start creating several alternative trajectories. Once we have several alternatives or "candidate" paths, we can evaluate each trajectory based on more complicated criteria, such as proximity to static and moving obstacles. Read Appendix E.

Six scenarios are provided in which another vehicle is the only moving obstacle. In each scenario this other vehicle drives along a different trajectory, some very normal, others completely anomalous. The cost of a candidate trajectory should be high if the other vehicle gets close to our intelligent vehicle, and/or when we deviate from the lane center.

Exercise 3.5. Implement a cost function in `compute_trajectory_cost`. It should first compute a per-timestep cost based on the distance between the moving obstacle and the vehicle. You should then combine the costs of each time step to a final single cost value $\widehat{c}$ for the entire trajectory, and also consider the lateral offset. Ensure that your solution avoids collisions in all provided scenarios.

Question 3.3. How did you define the final cost function? How do you combine the cost of each time step to a single cost value? Motivate your choices.

In several situations, the vehicle approaches another vehicle in front of it too fast. In these cases, the current cost function prefers a path that leads our vehicle outside the road boundaries. This is acceptable if the roadside is, say, grass. But, in some situations we could be driving along a steep cliff. Steering away from the road then means killing the driver. On the other hand, crashing into the vehicle ahead has a reasonable risk of injuring everybody in both vehicles. This vehicle could be an empty bulldozer, a family van, or a schoolbus.

In extreme situation of driving along a cliff, and approaching another vehicle too quickly from behind, the path planner could behave in one of two ways:

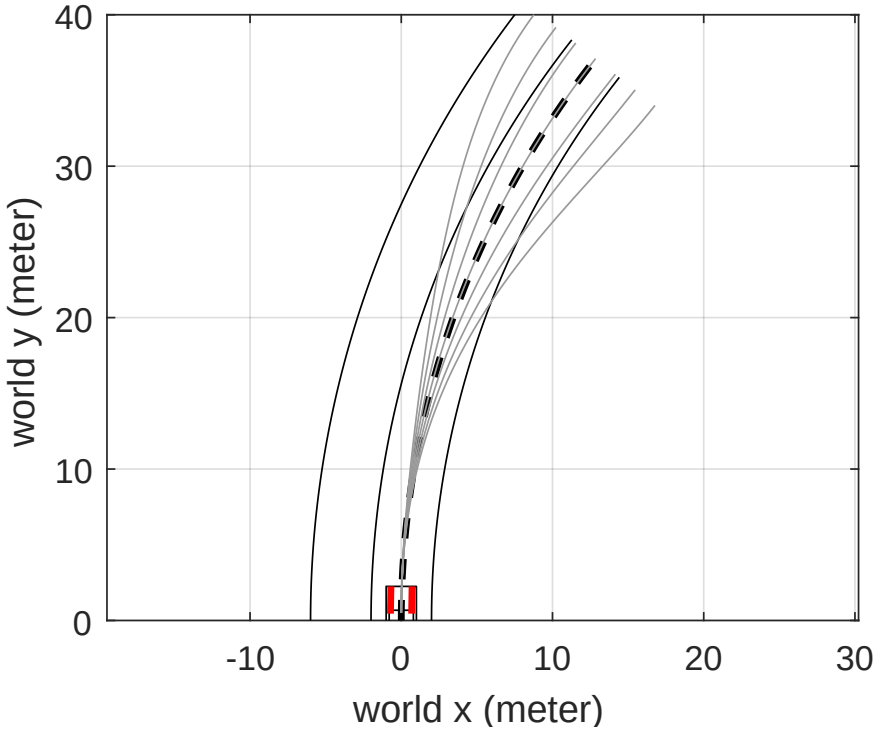

Figure 4: Creating multiple candidate trajectories with different final lateral offsets with respect to the road segment.

1. the planner steers the vehicle into the other vehicle ahead, risking injuring everybody.
2. the planner steers the vehicle off the cliff, killing its driver.

Of course, the planner should always take the safest option in less extreme situations.

Question 3.4. Pick one of these two behaviors as 'intended behavior'. What variable(s) should be detected from your surroundings, and included as new terms in the cost function to obtain this behavior? Write down the new cost function (you can add weights for new terms).

This completes the lab assignments for the Intelligent Vehicles course. If you followed through all assignments, you should now know the basic methods to plan a route to a road network, determine your location in a map, detect and track other moving obstacles, and avoid collisions while following the lanes. We hope you enjoyed making your vehicle more "intelligent" step by step!

References

[1] Dave Ferguson, Thomas M Howard, and Maxim Likhachev. Motion planning in urban environments. *Journal of Field Robotics*, 25(11-12):939–960, 2008.

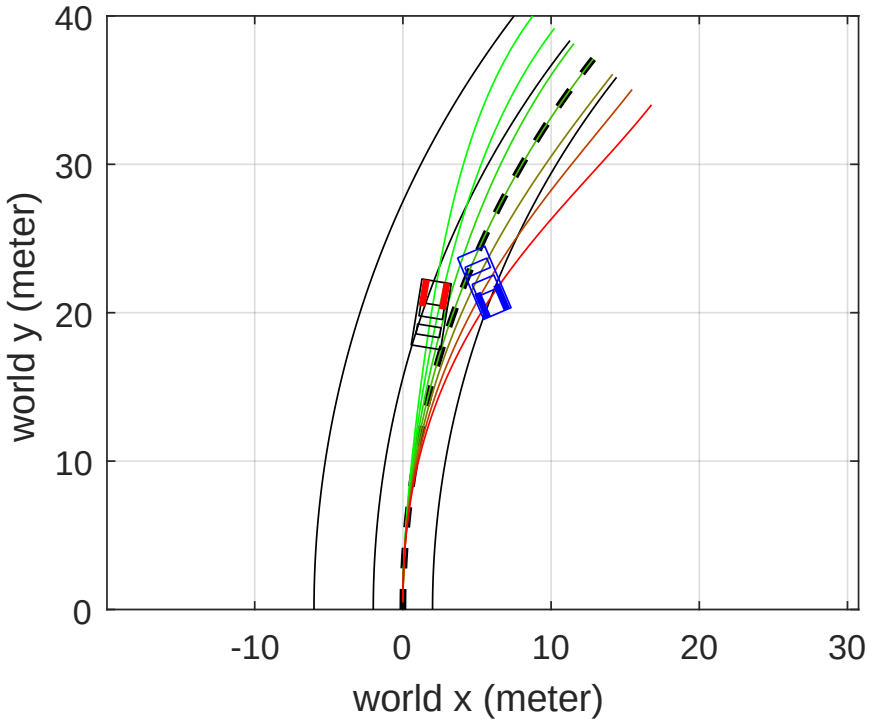

Figure 5: Evaluating the candidates with respect to the known obstacles (e.g. the blue car) and their predicted paths. Here good candidate trajectories are shown in green, bad ones in red. The vehicle selects the best candidate trajectory to execute.

Appendix

A A best-first search algorithm

Consider a graph G with V vertices and E edges, and a cost function $g(a, b)$ which computes the cost between vertices a and b. Let $N_G(v) = \{v'_1, v'_2, \cdots\}$ be the set of neighboring vertices of v in the graph, i.e. the graph contains edges (v, v'_1), (v, v'_2), etc.

The objective is to find a path with minimum cost between two vertices in the graph, namely $start$ and $goal$. Hence, the shortest path is a sequence of some n vertices $P = [v_1, \cdots, v_n]$ such that $v_1 = start$, $v_n = goal$, and total cost $\sum_{j=1}^{n-1} g(v_j, v_{j+1})$ is minimum. The sequence length n is unknown in advance.

The search algorithm in Algorithm 1 explores the graph by keeping a list of candidate vertices to further explore called the *frontier*. Initially, only the start vertex is on the frontier. All candidates are attributed a cost, and at each iteration, the algorithm selects the candidate with lowest cost from the frontier to explore further (hence the name *best-first search*). The cost $C[v]$ of a vertex v is the total cost of the shortest path from $start$ to v. For instance, if the shortest path to vertex v leads from $start$ successively through vertices a, b, and c, then $C[v] = g(start, a) + g(a, b) + g(b, c) + g(c, v)$.

When a candidate vertex v is explored, all its adjacent vertices $v' \in N_G(v)$ in the graph are retrieved, and the algorithm needs to determine if the shortest path to neighbor v' passes through v. Therefore the potential cost of v' is computed as the cost $C(v)$ plus the cost $g(v, v')$. If this is cost is lower than the best known cost to v', we have found a new shortest path to v', therefore add v' as a new candidate with its new cost. And, we register that v' shortest path originates from v in a backpointer array.

At some point, the $goal$ vertex will be explored (if there is a path from $start$, and the search can end. We then need to perform *backtracking* to construct the actual path from the backpointer array. Backtracking is done in *reverse* order by initiating at the goal vertex, and following the vertices on the path through the backpointer array until the start vertex is found. therefore the final path is reversed once more.

In Algorithm 1 a basic best-first search implementation is written out in a way that can efficiently be implemented in Matlab. For speed, it pre-allocates several of the internal arrays for all V

vertices, and avoids creating an explicit tree-structure.

Input: Start vertex $start$, goal vertex $goal$, cost function g, graph G
Output: Shortest path P

```
// Initialize backpointer array B, cost array C, and frontier Q
B = zeros(1, V) ;
C = inf(1, V) ;
C[start] = 0 ;                    // initial cost for path to start is zero
Q = [start] ;                     // initial frontier has start vertex only

// Explore graph with best-first search till goal is found
while true do
    v = argmin_{v∈Q} C[v] ;       // get vertex from frontier with lowest cost
    Q = remove(Q, v) ;            // remove selected vertex from frontier
    if v = goal then
        break while loop;         // goal is reached, start backtracking
    end
    foreach neighboring vertex v' ∈ N_G(v) do
        cost' = C[v] + g(v, v') ;                  // cost for new vertex v'
        if cost' < C[v'] then
            update C[v'] = cost' ;
            update backpointer B[v'] = v ;
            Q = append(Q, v') ;                    // add v' to frontier
        end
    end
end

// Backtracking
P = [goal], v = goal ;                                          // initialize
while v ≠ start do
    v = B[v] ;
    P = append(P, v)
end
P = reverse(P) ;                          // go start → goal instead of goal → start
```

Algorithm 1: Pseudo-code for a best-first graph search algorithm to find shortest path.

B The A-star search algorithm

The $A^\star$ algorithm extends the simple best-first search from Appendix A by using an heuristic function h to improve the way that the best-first search selects vertices from the frontier. We can think of the estimate $h(v, goal)$ as optimistic estimate of the expected remaining cost from vertex v towards the $goal$ vertex. Therefore, we consider score $S[v] = C(v) + h(v, goal)$ as an optimistic estimate of the total cost of the shortest path from $start$ to $goal$ going through v. As can be seen in the implementation in Algorithm 2, $A^\star$ therefore selects the vertex with the lowest score from the frontier to expend next, and is as a result a more efficient best-first search algorithm. Finally note that when a vertex is added to the frontier, its new score should

be computed too using its updated cost. Otherwise, the algorithm is the same as Algorithm 1.

Input: Start vertex $start$, goal vertex $goal$, cost function g, graph G
Output: Shortest path P

```
// Initialize backpointers B, costs C, scores S, and frontier Q
B = zeros(1, V) ;
C = inf(1, V) ;
C[start] = 0 ;                  // initial cost for path to start is zero
S = zeros(1, V) ;                                                    // (!)
S[start] = h(start, end) ;      // initial heuristic for start vertex (!)
Q = [start] ;                   // initial frontier has start vertex only

// Explore graph with best-first search till goal is found
while true do
    v = argmin_{v∈Q} S[v] ;              // get vertex with lowest score (!)
    Q = remove(Q, v) ;                 // remove selected vertex from frontier
    if v = goal then
        break while loop;               // goal is reached, start backtracking
    end
    foreach neighboring vertex v' ∈ N_G(v) do
        cost' = C[v] + g(v, v') ;                    // cost for new vertex v'
        if cost' < C[v'] then
            update C[v'] = cost' ;
            update S[v'] = cost' + h(v', goal) ;                         // (!)
            update backpointer B[v'] = v ;
            Q = append(Q, v') ;                          // add v' to frontier
        end
    end
end

// Backtracking
P = [goal], v = goal ;                                          // initialize
while v ≠ start do
    v = B[v] ;
    P = append(P, v)
end
P = reverse(P) ;                        // go start → goal instead of goal → start
```

Algorithm 2: Pseudo-code for a A-star graph search algorithm to find shortest path. Note that the lines marked by **(!)** are different from the Algorithm 1.

C Planning in configuration space with cell decomposition

The *configuration space* of a vehicle or robot is the d-dimensional space which contains all possible d-dimensional state vectors $\mathbf{x}_t$. For instance, if the state is a 4d vector $\mathbf{x} = [x, y, \theta, \omega]^\top$ then we can think of it as a point in a 4d configuration space. The motion or kinematic model of the vehicle or robot then describes how the state can move through this space. To plan motion from one state to another, we need to find a continuous path from the initial state to the goal state. It turns out that we can use graph search to get a good (approximate) solution.

One popular approach is *cell decomposition*, which discretizes the configuration space into a grid of discrete states. In case of the 4D space, this would therefore result in a 4D grid. Each

cell in this grid thus corresponds to a state, and the motion model defines which cells can be reached from which other cells. This 4D grid can also be seen as a graph, where the cells are the vertices, and the motion model defines the edges. To plan a movement from initial to goal state thus reduces to the approximately same problem of finding an optimal path through this graph from the start vertex to the goal vertex, for which the algorithm of Appendix A can be used. The complete approach of planning in the configuration space can be summarized as

1. Discretize the configuration space into a finite set of vertices for the graph.
2. Define edges between the vertices. The edges correspond to the possible movements that the vehicle can make.
3. Determine which states are unobtainable, and remove any edges to unreachable states.
4. Map the goal and target state to the best matching vertices in the graph.
5. Run the path-planning algorithm
6. Convert the path of vertices back to a sequence of vehicle states.

Since the discretized space has lost resolution, the positions on the path might express to many rough changes in various configuration space dimensions. Also, there could be other constraints or dynamics that have been omitted from the configuration space to keep planning fast, such as the vehicle momentum which would favor few velocity changes. The final path should therefore checked and optimized to a smooth continuous path with few discontinuities before being send to the controller to be executed.

D Trajectory optimization

The vehicle state at time t is expressed by the vector $\mathbf{x}_t = [x_t, y_t, \theta_t, \kappa_t, v_t]$, consisting of position x_t, y_t, orientation θ_t, and the velocities κ_t, v_t, where κ_t is the turning rate (in rad/s), and v_t the forward velocity (in m/s). Given an initial vehicle state $\mathbf{x}_0$ and a goal vehicle state $\mathbf{x}_G$ for some time T in the near future, the trajectory optimization problem is to find the control parameters $\mathbf{u}_t$ for $0 \leq t \leq T$ that bring the vehicle in the desired future state.

Instead of optimizing all $\mathbf{u}_t$ directly, we shall see that it is more convenient to use a smaller set of parameters $\mathbf{p}$ to describe the control inputs over time. Which parameters are exactly used will be explained in D.1, but is not very important for understanding this overview first.

Another piece of the puzzle is to determine expected state $\mathbf{x}_T$ of the vehicle given the initial state $\mathbf{x}_0$, parameters $\mathbf{p}$, and the vehicle dynamics. We regard the resulting state as a non-linear function of $\mathbf{p}$ and $\mathbf{x}_o$, i.e. $\mathbf{x}_T(\mathbf{x}_0, \mathbf{p})$, which will be explained in D.2.

The trajectory optimization can now be formulated as finding those parameters $\mathbf{p}$ that ensure that $\mathbf{x}_T(\mathbf{x}_0, \mathbf{p})$ is very similar to $\mathbf{x}_G$. This problem can be stated in terms of an error vector C,

$$C(\mathbf{p}) = \begin{bmatrix} x_T - x_G \\ y_T - y_G \\ \theta_T - \theta_G \end{bmatrix} \qquad \text{where} \quad \mathbf{x}_T(\mathbf{x}_0, \mathbf{p}) = [x_T, y_T, \theta_T, \kappa_T, v_T]. \tag{1}$$

We wish to minimize the squared norm $||C(\mathbf{p})||^2$, i.e. all components of the error vector should go to 0. Since $\mathbf{x}_T(\mathbf{x}_0, \mathbf{p})$ is highly non-linear, there is not a closed-form solution to this optimization problem. However, we can use general non-linear optimization methods, which are

typically based on gradient descent. From a current estimate of $\mathbf{p}$, gradient descent methods determine the gradient $\delta C(x,p)/\delta \mathbf{p}$ which expresses how the error C changes for a small change in parameters $\delta \mathbf{p}$. We can then make a small adjustment to the parameters along the gradient such that the error becomes smaller.

Matlab provides a simple interface for such non-linear least-squares optimization method using `lsqnonlin`. We just need to provide it the error vector $C(\mathbf{p})$ as a function of the parameters $\mathbf{p}$ to optimize, and it uses iterative gradient descent method to find optimal parameters $\mathbf{p}'$,

$$\mathbf{p}' = \underset{\mathbf{p}}{\operatorname{argmin}} ||C(\mathbf{p})||^2 \tag{2}$$

In conclusion, we can determine a (near) optimal set of steering parameters $\mathbf{p}$ that ensure that the vehicle reaches the goal state $\mathbf{x}_G$ by constructing a vector of errors to jointly minimize, and using a generic gradient descent method.

D.1 Control parameterization

The parameters $\mathbf{p}$ describe the control inputs $\mathbf{u}_t$ for future time period $0 \leq t \leq T$. One approach is to let $\mathbf{p} = (\mathbf{u}_0, \cdots, \mathbf{u}_T)$ explicitly be the control inputs at many time steps. However, too many parameters are difficult to optimize efficiently. Instead, it is better to describe the control inputs as *functions* with only a few parameters $\mathbf{p}$ which are easier to optimize, namely $\mathbf{u}_t = (a(t,\mathbf{p}), \omega(s,\mathbf{p}))$. We will now explain the symbols in these terms.

Here $a(t,\mathbf{p})$ is the *acceleration profile*, as function of t. In this assignment, we will consider only constant velocity dynamics, meaning that the acceleration function $a(t,\mathbf{p}) = 0$ is simply zero for any t and parameters $\mathbf{p}$.

Likewise, $\omega(s,\mathbf{p})$ is the *steering profile*. Note that instead of expressing this profile as a function of time t, it is expressed as a function of s, the traveled distance (in meter). Using s instead of t ensures that we can more or less determine ω independent of the acceleration profile: it tells is *where* to steer instead of *when* to steer.

The steering profile function will be expressed using a small set of *control points* k_0, k_1, and k_2, which fix the steering control at the beginning of, half-way, and end of the trajectory. Intermediate steering angles can then be obtained by interpolation. Let s_f be the total length (in meter) of the trajectory at future time T, then we know that the control function should satisfy $\omega(0,\mathbf{p}) = k_0$, $\omega(s_f/2,\mathbf{p}) = k_1$, and $\omega(s_f,\mathbf{p}) = k_2$.

A good choice is for interpolation is to model the function as the cubic *spline*, which ensures that it moves through the points k_0, k_1 and k_2 at $s/s_f = 0$, $s/s_f = .5$, and $s/s_f = 1$ respectively, see Figure 6. Finally, note that initial steering angle control k_0 is already determined by the initial vehicle state. Therefore, the vector $\mathbf{p}$ of parameters to optimize is $\mathbf{p} = [k_1, k_2, s_f]$.

D.2 Dynamics and Euler integration

Similar to the self-localization assignment earlier, the non-linear vehicle motion model $\mathbf{x}_{t+\Delta t} = f(\mathbf{x}_t, \mathbf{u}_t, \Delta_t)$ expresses how the state $\mathbf{x}_t$ changes to state $\mathbf{x}_{t+\Delta t}$ over a time interval Δt. Control

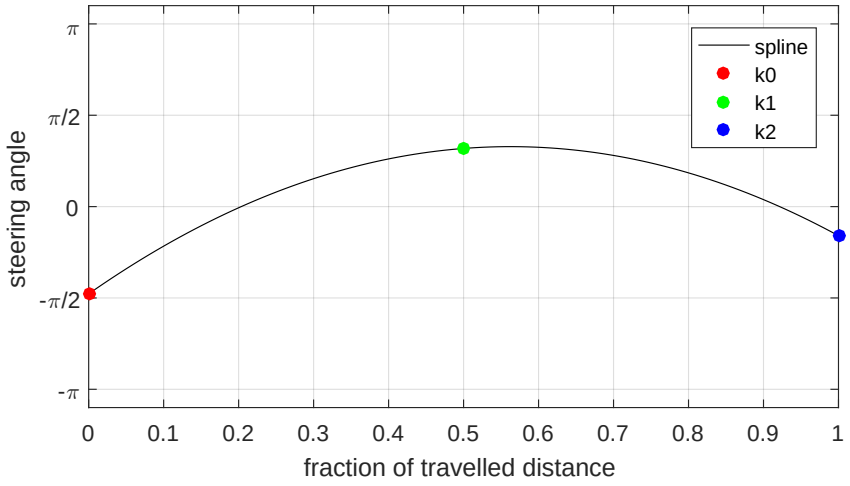

Figure 6: Defining steering control ω using a spline with only three control points k_0, k_1 and k_2. The control points define the the output at the fraction of traveled distance s/s_f for $s/s_f = 0\%$, $s/s_f = 50\%$, and $s/s_f = 100\%$. The spline is a smooth function without abrupt changes through the control points, thus defining the control ω for any intermediate location along the trajectory.

inputs $\mathbf{u}_t = (a_t, \omega_t)$ define acceleration a_t and steering angle ω_t, hence f is defined as

$$x_{t+\Delta t} = x_t + v_t \sin(\theta_t)\Delta_t \qquad y_{t+\Delta t} = y_t + v_t \cos(\theta_t)\Delta_t \tag{3}$$

$$\theta_{t+\Delta t} = \theta_t + v_t \kappa_t \Delta_t \qquad v_{t+\Delta t} = v_t + a_t \Delta_t \tag{4}$$

$$\kappa_{t+\Delta t} = \omega_t \tag{5}$$

Note that this model is *not* stochastic, i.e. it does not include any process noise.

If we would know the control input $\mathbf{u}_t$ for a future time period $0 \leq t \leq T$ up to time T, the model can be used to determine the expected final state $\mathbf{x}_T$. This can be done by continuously integrating the all the changes in the state over the whole period, though this is infeasible in practice. Instead, we will use Euler integration which is a numerical approximation to do the integration. In effect, the Euler integration estimates the state sequence $\mathbf{x}_0$, $\mathbf{x}_{\Delta t}$, $\mathbf{x}_{2\Delta t}$ up to $\mathbf{x}_T$ by simulating the dynamics through iteratively applying f with small time intervals Δt,

$$\mathbf{x}_{\Delta t} = f(\mathbf{x}_0, \mathbf{u}_0, \Delta_t) \tag{6}$$

$$\mathbf{x}_{2\Delta t} = f(\mathbf{x}_{\Delta t}, \mathbf{u}_{\Delta t}, \Delta_t) \tag{7}$$

$$\dots$$

$$\mathbf{x}_T = f(\mathbf{x}_{T-\Delta t}, \mathbf{u}_{T-\Delta t}, \Delta_t). \tag{8}$$

The smaller Δt, the more iterations are required to reach time T, but also the more accurate the approximation. Our objective is thus to determine the control inputs such that $\mathbf{x}_T$ is similar to our goal state $\mathbf{x}_G$.

E Trajectory generation and evaluation

Appendix D has explained how to optimize a trajectory to bring the vehicle from an initial state to a goal state $\mathbf{x}_G$. So how do we use this to plan a path that avoids collisions with moving obstacles?

First, we use the trajectory optimization approach to generate several *candidate* trajectories, each resulting from a slightly different goal state. These goal states are close to the ideal goal state, but with a small or large amount of lateral offset l with respect to lane center, see Figure 3. For instance, $l = -1$ would result in a trajectory 1 meter to the left of the lane center, and $l = 2$ in a trajectory 2 meter to the right of the center. Once candidate trajectories have been generated, each is evaluated by comparing the future vehicle positions to the future positions of the other (moving) obstacles. The "badness" of a trajectory is represented by its cost $\widehat{c}$, hence we should pick the candidate trajectory with lowest cost, see Figure 3.

Consider a candidate trajectory generated for lateral offset l, for which we obtain predicted vehicle states $\mathbf{x}_t$ at future time t. Also, let $\mathbf{x}'_t$ be the predicted state of an another vehicle, and $d(\mathbf{x}_t, \mathbf{x}'_t)$ the distance in meters between the vehicles. If the vehicle moves to close to the obstacles, this should result in a higher cost. We can compute a per-time step cost $\widehat{c}_t(\mathbf{x}_t, \mathbf{x}'_t)$. One possible cost function could be $\widehat{c}_t^{dist}(\mathbf{x}_t, \mathbf{x}'_t) = e^{-d(\mathbf{x}_t, \mathbf{x}'_t)}$, which gradually becomes smaller as the distance between the objects increases. Alternatively, the cost function could only yield non-zero cost if the distance is below a threshold τ, e.g. $\widehat{c}_t^{dist}(\mathbf{x}_t, \mathbf{x}'_t) = max(\tau - d(\mathbf{x}_t, \mathbf{x}'_t), 0)$. The per-time step cost should be combined into a single trajectory cost $\widehat{c}^{dist}$, for instance by maximizing $\widehat{c}^{dist} = \max_t[\widehat{c}_t^{dist}(\mathbf{x}_t, \mathbf{x}'_t)]$, or averaging $\widehat{c}^{dist} = \frac{1}{T}\sum_t[\widehat{c}_t^{dist}(\mathbf{x}_t, \mathbf{x}'_t)]$, or something else.

Likewise, larger deviations from the ideal lane center should also increase the cost. For instance, we could just take the absolute lateral deviation as a cost term $\widehat{c}^{lat} = |l|$. The final trajectory cost $\widehat{c}$ can be a combination of different cost terms,

$$\widehat{c} = \alpha \cdot \widehat{c}^{dist} + \widehat{c}^{lat}, \tag{9}$$

where α determines how the different terms are relatively weighted.