TSFS12 HAND-IN EXERCISE 2
Planning for Vehicles with Differential Motion Constraints

August 28, 2021

1 OBJECTIVE

The objective of this exercise is twofold. First, two different kinds of planning methods will be investigated: motion planning using graph search on a state lattice with motion primitives *and* sampling-based motion planning using rapidly-exploring random trees (RRTs). Second, for both of these methods, differential constraints on the vehicle motion will be considered and taken into account in the planning. For the lattice planning, optimality in terms of path length will also be considered. The planning using RRTs will be performed both for a particle model and for a simplified kinematic car model in an environment with static obstacles. The implementation of the graph-search methods from Hand-in Exercise 1 will be re-used in this exercise, as part of the implementation of the lattice-based motion planning for the kinematic car model. In the extra assignment for higher grades in Appendix A, optimal planning using RRT* is considered for a particle model.

2 PREPARATIONS BEFORE THE EXERCISE

Before performing this hand-in exercise, please make sure you have read and understood the material covered in:

- Lecture 4.
- Sections 5.5 and 14.4 in LaValle, Steven M: "*Planning Algorithms*", Cambridge University Press, 2006.
- Section 4.5 in Bergman, K: "*Exploiting Direct Optimal Control for Motion Planning in Unstructured Environments*", Ph.D. Thesis No. 2133, Div. Automatic Control, Linköping Univ., 2021.
- *For the extra assignment*: Section 3.3.3 in Karaman, S., & E. Frazzoli: "Sampling-based algorithms for optimal motion planning", *The International Journal of Robotics Research*, 30(7), 846–894, 2011, on the RRT* algorithm.

To get code skeletons to be used for the different exercises, download the latest files from `https://gitlab.liu.se/vehsys/tsfs12` and familiarize yourself with the provided code.

You are free to choose in which language to implement these exercises. Code skeletons are available in Matlab and Python, but if you prefer to make your own implementation that is also possible. In the student labs at campus, all Python packages needed are pre-installed in the virtual environment activated by

```
source /courses/tsfs12/env/bin/activate
```

3 REQUIREMENTS AND IMPLEMENTATION

The objective of the exercise is to implement different motion planners taking differential motion constraints into account, evaluate the properties of the different planners, and get an understanding for which planning tasks that they are suitable. Therefore, to pass this exercise you should implement baseline versions of the following motion planners:

- Lattice-based planning with motion primitives for a kinematic car model moving in a world with two translational and one orientational degrees-of-freedom. Here, both feasible and optimal (with respect to path length) motion planning is considered.
- Feasible motion planning using RRT for a particle moving in a 2D world.
- Feasible motion planning using RRT for a kinematic car model in a world with two translational and one orientational degrees-of-freedom.

The exercise is examined by submitting the following on the Lisam course page:

1. Runnable code. If your implementation consists of several files, submit a zip-archive.
2. A short document, which does *not* have to be formatted as a self-contained report, containing:
   - answers to the questions, including relevant plots, in Section 4, and other questions that you have encountered or reflections that you have made when solving this exercise.
   - a concluding discussion.

   Submit the document in PDF format.

It is allowed to discuss the exercises on a general level with other course participants. However, code sharing outside groups is not allowed. Moreover, both students in the group should be fully involved in performing all exercises, and thereby be prepared to answer questions on all subtasks.

See Appendix A for the extra assignment needed for higher grades. The extra assignment is done individually and is submitted by a document answering the questions, including suitable figures, plots, and tables. The document is to be submitted in PDF format. The code should also be submitted as a zip-archive. There is only pass/fail on the extra exercise and the document can not be revised or extended after first submission. It is not required to get everything correct to pass the assignment, we assess the whole submission. You are of course welcome to ask us if you have questions or want us to clarify the exercise.

Since these exercises are part of the examination of this course, we would like to ask you not to distribute or make your solutions publicly available, e.g., by putting them on github. (A private repository on gitlab@liu is of course fine).

Thanks for your assistance!

4 DISCUSSION TOPICS & EXERCISES

This hand-in exercise is divided into three different parts, with subtasks to be performed for each. The required implementation code for solving the different tasks is partially provided for the exercises in the skeleton files. In the directory with the code skeletons, there is a subdirectory called `Functions` for the Matlab version, where a number of auxiliary functions and classes are available that will be useful in the implementation. In Python, the corresponding implementation with auxiliary functions and classes is available in the files `motionprimitives.py` and `world.py`.

4.1 Motion Planning Using a State Lattice

In this exercise, we will use motion primitives to do motion planning on a state lattice, i.e., a discretization of the state space in which the motion is taking place. The model adopted is a kinematic single-track model (see Lecture 3), where the wheels on each axle have been lumped together to simplify the modeling. The car states are defined by the x and y coordinates of the rear axle, and the orientation θ in a fixed world coordinate frame. The state-space model in continuous time can be written as

$$\dot{x} = v\cos(\theta), \tag{1}$$
$$\dot{y} = v\sin(\theta), \tag{2}$$
$$\dot{\theta} = \frac{v}{L}\tan(\delta), \tag{3}$$

where v is the velocity, δ is the steering angle of the front wheel and L is the length of the car wheelbase. In this exercise, we assume that the longitudinal speed v is constant and thus let the steering angle δ be the only control input. Moreover, forward and reverse motions are allowed in this exercise, which means that the car is either moving forward or backward with a given speed. For the graph-search in the state lattice, the implementations of graph-search strategies from Hand-in Exercise 1 will be re-used.

In the provided code for Matlab, the main file is `run_lattice_planning.m`, which will be completed in this task. For the corresponding implementation in Python, open the jupyter notebook `lattice_planning.ipynb`. The skeleton file for Python assumes that you have collected all your planners in a file `planners.py`, but you can also paste your solutions from Hand-in Exercise 1 directly into the notebook if you prefer.

The motion primitives encode possible optimal (with respect to minimum path length) movements from an initial state to another neighboring state that must be located on the state lattice. When computing the motion primitives, two fundamental properties of the model are employed. The first is that the motion given a defined initial orientation is translationally invariant—i.e., a motion from an initial state with orientation θ_0 that is applicable (in the sense that the car stays on the defined state lattice) at one state (x_0, y_0) is applicable also for another arbitrary translated state (x_0', y_0') with the same initial orientation of the car. The second property is that for all motion primitives that define forward motion, a corresponding reverse motion could be obtained with the same path, but reversed initial and final state. Consequently, we only need to compute forward motion primitives, and then utilize this property in the online planning to obtain the reverse motion of the car.

Exercise 4.1. The motion primitives are computed using the optimization software package CasADi for a grid of the orientation state θ according to

$$\theta \in \left\{-\frac{3\pi}{4}, -\frac{\pi}{2}, -\frac{\pi}{4}, 0, \frac{\pi}{4}, \frac{\pi}{2}, \frac{3\pi}{4}, \pi\right\}. \tag{4}$$

The computations required for determining the motion primitives are in Matlab performed by the code provided in the file `MotionPrimitives.m` in the subdirectory

`Functions`. In Python, the functionality is implemented in `motionprimitives.py`. The motion segments are computed using numerical optimization with CasADi.

- To run the computation of the motion primitives, you need CasADi installed. This is pre-installed in the student labs at campus, you only need to add the installation directory to the Matlab path. You do this by including the line below in your scripts

  ```
  addpath /courses/tsfs12/casadi
  ```

  To run on your own computer, download and install CasADi according to the instructions on

  `https://web.casadi.org/get/`

  If you are using Python, CasADi is pre-installed in the student labs at campus, just activate the virtual environment as described in Section 3. To install the CasADi package on your private computer, run

  ```
  pip install casadi
  ```

  after you activated your virtual environment.

- Read through the code and available methods for the class `MotionPrimitives` to get an overview of what is done in the different parts of the implementation. In particular, study the function `compute_mprims` in which the actual motion primitives are computed using optimization. The function has the following API:

  ```
  compute_mprims(theta_init, lattice, L, v, u_max)
  ```

  where `theta_init` is the vector with allowed initial orientations, `lattice` defines the initial and goal configurations for the desired motion primitives, `L` is the length of the car wheelbase, `v` is the velocity, and `u_max` is the maximum steering angle.

In the computations, symmetry between the different motion primitives is used to avoid some computations (thereby reflections and rotations of the already computed motion primitives are employed successively). The computed motion primitives are stored in the file `mprims.mat` (Matlab) or `mprims.pickle` (Python) and the implementation checks if the file exists to avoid re-computing them, if they already have been computed once.

Exercise 4.2. Now we will use the computed motion primitives to solve motion-planning problems for the kinematic car model. A code skeleton for this purpose is available in the file `run_lattice_planning.m` (Matlab) or `lattice_planning.ipynb` (Python). In the first part of the code, the `MotionPrimitives` object is defined, the motion primitives for forward motion are computed, and the different motion primitives are plotted. Then a world model is defined, using the `BoxWorld` class (provided in the subdirectory `Functions` in Matlab and in `motionprimitives.py` in Python), with which box-shaped obstacles can be placed in a defined 2D world. A number of missions are pre-defined in the code that you are welcome to explore, but please also experiment yourselves with other interesting missions.

We will now reuse the implementations of the graph-search algorithms from Hand-in Exercise 1. Recall that the API for the planners is defined as

```
Planner(number_of_nodes, mission, f, heuristic, num_controls)
```

The state-update function defined by `f` in `run_lattice_planning.m` (Matlab) or `lattice_planning.ipynb` (Python) uses the available motion primitives in the struct/dictionary `mp.mprims` to determine possible next states from the current state.

- Carefully study the content of the file `next_state.m` (Matlab) or the implementation of `next_state` in `lattice_planning.ipynb` (Python), where the possible state transitions from a certain initial state are computed. Note that there is a flag specifying if reverse driving should be allowed or not (true by default).
- In the provided skeleton file, there are some lines of code that allow you to investigate the output from `next_state` for the initial state. Describe the output, is it what you expect?

Similarly to the implementation in Hand-in Exercise 1, you need to define a heuristic function for use in the graph-search strategies `best_first` and `astar`. This is defined by the function `cost_to_go`. A first attempt of defining this function is available in the provided code skeleton, which only considers the translational degrees-of-freedom.

When defining the respective planners, we also would like to keep track of the control input that is associated with each motion segment. Therefore, in this exercise we set `num_controls` to 3 and store three indices in a matrix that keep track of the motion primitive used for the different motion segments. The first index is specifying the set of motion primitives corresponding to the current orientation and the second index is the particular motion primitive used in that set. The third index is either 1 or -1, depending on if the car is moving forward or reverse, respectively, in that motion segment.

- Implement a lattice planner that computes a motion plan from an initial state to another desired final state (both in the defined state lattice) based on the code skeleton provided in the file `run_lattice_planning.m` (Matlab) or in the file `lattice_planning.ipynb` (Python). Apply each of the methods from Hand-in Exercise 1, `depth_first`, `breadth_first`, `dijkstra`, `best_first`, and `astar`, for performing a lattice-based computation of a motion plan.
- Plot the resulting motion plans for the respective graph-search strategy (for improved visibility, it could be advantageous to plot the different plans in different plots). For determining the path corresponding to a particular sequence of motion primitives (specified by the indices in `plan.control`, where `plan` is the output from one of the graph-search algorithms) the function `plan_to_path` in the `MotionPrimitive` class is useful. To determine the coordinates (x, y, θ) corresponding to a certain node number, the property `st_sp` in the object `world` is useful.
- Analyze the results for the different planners. Do all planners provide the same plan from start to goal? In case they differ, is this an expected result? Do any of the planners provide the optimal path (with respect to path length)?

Exercise 4.3.

- Experiment with different types of planning missions, i.e., modify the initial and goal states, and define different world models. Note that the start and goal states need to be *on the state lattice*.
- Reflect upon the results with respect to length of the paths and the time required to compute the plan. To illustrate the results
  - plot the plan lengths vs. the planning times for the different planners and
  - plot the planning time vs. the number of visited nodes during the search

  for at least two suitable planning missions.

- Experiment with different choices of the heuristics used in the specified function `cost_to_go`. Can you find alternative functions, providing higher performance of the planner (in terms of computation time and number of visited nodes)?

4.2 RRT with Particle Model

In this part of the exercise, the task is to find a path for a particle moving in a plane (2D world) using RRT. There are no motion restrictions for the particle in this plane, except the obstacles. In the provided code for Matlab, there are two files named `rrt_particle.m` and `run_rrt_particle.m` that will be completed in this task. For implementation in Python, open the jupyter notebook `rrt_particle.ipynb`.

Exercise 4.4. The file `rrt_particle.m` (Matlab) or `rrt_particle.ipynb` (Python) gives the foundation for implementation of an RRT motion planner for the particle model. The API for the function is

```
rrt_particle(start, goal, world, opts)
```

where `start` is the initial state, `goal` is the desired goal state, `world` is a description of the map of the world using the class `BoxWorld` (provided in the subdirectory `Functions` in Matlab and in `world.py` in Python), and `opts` contains different options for the planner (in Python, `opts` is a dictionary and the elements are accessed with their corresponding key, e.g., `opts['beta']`).

The options that should be considered are:

- A bias in the sampling towards the goal state in the tree expansion, governed by the parameter β.
- The tree is expanded at maximum K steps.
- In each step the particle is moving (at most) λ m in the direction of the sampled state in the `steer` function. If the distance to the sampled state is lower than λ, the new state becomes the sampled state.
- A threshold ε should also be included, which if positive terminates the tree expansion when a node in the tree is closer than ε to the goal state.

Instead of having a complete class or object for representation of the graph $\mathcal{G}$ describing the constructed tree, we keep track of the states in the matrix `nodes` and have a vector `parents` that stores the node number (corresponding to column number in `nodes`) for the parent of each node. This creates a more light-weight representation of the constructed tree.

The tasks are now the following:

- Read through the code skeleton in the file `rrt_particle.m` (Matlab) or in the file `rrt_particle.ipynb` (Python) carefully and make sure you understand the intention of the different available auxiliary functions (these will be useful later when completing the RRT implementation). Note that the function `nearest` returns the index of the nearest node, not the node itself.
- Study also the implementation and available methods for the class `BoxWorld`; the method `obstacle_free` will be convenient in the implementation of the RRT planner. Note that it will be necessary to *check several points along the path between the nearest node and the new node to avoid collisions with the obstacles for intermediate points.*
- Complete the implementation of the RRT planner by filling in the missing lines of codes in the function `rrt_particle`. In particular, the implementation should consider the options discussed previously by using the data structure `opts`:

- `opts.beta`: probability for selecting goal state as target state (i.e., the bias towards the goal state),
- `opts.lambda`: step size for the tree expansion in the `steer` function,
- `opts.eps`: threshold for stopping the search (negative for full search),
- `opts.K`: maximum number of iterations.

Exercise 4.5. Now we will use the implementation of the RRT planner to solve motion-planning problems for the particle model. The solution is to be implemented in the file `run_rrt_particle.m` (Matlab) or `rrt_particle.ipynb` (Python). The provided code skeleton defines an example world (one of the example worlds also used in the previous part of the exercise on lattice planning) and draws it and thereafter defines the start and the goal state, and default values for the options needed for the RRT planner.

- Complete the code in `run_rrt_particle.m` (Matlab) or `rrt_particle.ipynb` (Python) such that a path is planned for the particle to move in the defined world from an initial state to another desired goal state (as close as possible). Use the RRT planner implemented in `rrt_particle`.
- Plot the complete constructed tree *as well as* the planned path from start to goal in the diagram with the obstacles. *Hint: When determining the computed path, it is easier to start with the goal node and then backtrack through the tree to the initial node by using the variables* `nodes` *and* `parents`.
- Try different combinations of obstacles (defined by the `world` object), start state (variable `start`), and goal state (variable `goal`). Analyze the results and the resulting paths and discuss the results.

Exercise 4.6.

- Extend the implementation by adding code for computing the total number of nodes in the tree, the number of nodes along the path from the found path from the initial state to the goal state, and the length of the path from the initial state to the goal state.
- Vary the step length `opts.lambda` in the `steer` function. How does the tree structure change? Discuss the results in terms of the quantitative criteria implemented in the previous item.
- Consider now the case that a positive `opts.eps` is used. Vary the parameter `opts.beta` that determines the bias towards the goal state in the sampling of the states. Also, vary the threshold parameter `opts.eps` and study the results (in particular with respect to the total number of nodes in the tree before the search is terminated). Can you see any differences in the obtained tree or planned path?

4.3 RRT with Motion Model for a Simplified Car

The task in this part of the exercise is to extend the implementation from the previous task, such that the planner computes a feasible path for a simplified car model with differential motion constraints. The model is the same as the one used in the lattice planning in Exercises 4.1–4.3, namely a car modeled with kinematic motion equations, moving in a world with three degrees-of-freedom (two translational and one orientational). Also in this exercise, only forward motion with a constant velocity v is allowed and thus the only control input u is the steering angle δ.

The car simulation model is implemented in the file `sim_car.m` in Matlab and directly in the jupyter notebook `rrt_diff.ipynb` in Python. In the provided function `sim_car`, the car model is simulated forward with a specified time step, using a forward-Euler discretization with step size h. When choosing the control input u in the planning, there is also a need to select a discrete set of control actions, $\mathcal{U}$, to evaluate in each step for expansion of the tree. In this exercise, this set is by default chosen as a uniform grid with a specified number of points in the interval $[-\frac{\pi}{4}, \frac{\pi}{4}]$. The set $\mathcal{U}$ is then used when building the tree in the RRT planner. Thereby, the control input (i.e., the steering angle) is kept constant during the whole forward step in the simulation in `sim_car`.

In the provided code for Matlab, there are two files that are named `rrt_diff.m` and `run_rrt_diff.m` that will be completed in this task. For implementation in Python, open the jupyter notebook `rrt_diff.ipynb`.

Each edge in the tree will be associated with a certain control input and a certain state trajectory from the parent node to its child node. *Note that in contrast to the RRT planner for the particle model in the previous exercise, here the edges between nodes are not necessarily straight lines.* These trajectory segments are intended to be stored in the data structure `state_trajectories`. Another choice that has to be made in the implementation is how closeness should be evaluated, when finding the node closest to the new sampled state or terminating the search because we are close to the goal state. Taking the Euclidean norm of the difference of the state vectors directly, can lead to undesirable behavior in the search since translational and orientational degrees-of-freedom are then compared equally. As a remedy for this, alternative distance measures for the orientation should be considered. This distance measure is to be implemented in the function `distance_fcn` implemented in `rrt_diff.m` (Matlab) or `rrt_diff.ipynb` (Python). In the provided code skeleton, this function by default treats all states equally, which might not be the best choice in terms of performance.

The same options in `opts` as in the previous implementation of an RRT for the particle model should be supported in this implementation. A bias towards the goal state is introduced in the tree expansion, governed by the parameter β. The tree is expanded at maximum K steps, where in each step the particle is moving for λ s (i.e., a certain time step). *Note that* `opts.lambda` *in this implementation corresponds to a time step rather than a step length in path.* A threshold ε is also included, which if positive terminates the tree expansion when a node in the tree is closer than ε to the goal state.

Exercise 4.7. The file `rrt_diff.m` (Matlab) or `rrt_diff.ipynb` (Python) gives the foundation for implementation of an RRT motion planner for the car with a kinematic motion model. The API for the function is

```
rrt_diff(start, goal, u_c, sim, world, opts)
```

where `start` is the initial state, `goal` is the desired goal state, `u_c` is a vector with the possible control actions (i.e., steering angles) in each step, `world` is a description of the map of the world using the class `BoxWorld` (provided in the subdirectory `Functions` in Matlab and in the file `world.py` in Python), and `opts` contains the different options for the solver.

- Read through the code skeleton in `rrt_diff.m` (Matlab) or `rrt_diff.ipynb` (Python) carefully and make sure you understand the intention of the different available auxiliary functions (these will be useful later when completing the RRT implementation). Note that the function `steer_candidates` returns empty variables in case none of the tested inputs result in a collision-free path.
- Complete the implementation of the RRT planner by filling in the missing lines of codes in the function `rrt_diff`. Note the variable `state_trajectories` in

which all motion segments (i.e., edges in the tree) are intended to be stored. This data structure will be useful later when plotting the resulting trees.

Exercise 4.8. Now we will use the implementation of the RRT planner in `rrt_diff` to solve motion-planning problems for the kinematic car model. The solution is to be implemented in the file `run_rrt_diff.m` (Matlab) or `rrt_diff.ipynb` (Python). The provided code skeleton defines an example world and draws it, and thereafter defines the start and goal state, the possible control inputs `u_c`, and the options needed for the RRT planner.

- Complete the code in `run_rrt_diff.m` (Matlab) or `rrt_diff.ipynb` (Python) such that a motion plan is computed for the kinematic car model to move in the defined world from an initial state to another desired goal state (as close as possible). In Matlab, use the planner implemented in the file `rrt_diff.m`, whereas in Python the RRT implementation is made directly in the jupyter notebook.
- Plot the constructed tree *as well as* the planned path from start to goal in the diagram with the obstacles. The connections between nodes should be visualized in the plot by its corresponding actual path, not a straight line. Notice that all state trajectories, including the orientation, should in the implementation be available in the data structure `state_trajectories`. Also here it is advantageous to start at the goal node and backtrack through the tree when plotting the solution path.

Exercise 4.9.

- Try different combinations of obstacles (defined by the `world` object), initial state (variable `start`), and goal state (variable `goal`).
- Experiment with different time-step lengths using the variable `opts.lambda`. How are the resulting tree and paths affected?

Exercise 4.10.

- Experiment with different choices of the distance measure for the state vector in the function `distance_fcn`. How does it affect the results? Investigate in particular the case when a positive `opts.eps` is used.

Exercise 4.11.

- Try different control sets $\mathcal{U}$, i.e., experiment with different degrees of discretization of the control input u. Can you see any trends in the resulting structure of the tree when varying this set?

A EXTRA ASSIGNMENT

RRT* with Particle Model

In the previous tasks based on RRT in this hand-in exercise, motion plans that are feasible with respect to two different motion models have been computed. However, it could be noted that the resulting paths are typically not straight and there is no measure in the algorithms to try to achieve as short path as possible from start to goal. In this extra exercise for higher grades, we will therefore extend the first RRT task that used a particle model to the more complex problem of planning a minimum-length path using RRT*.

In the provided skeleton code for Matlab, there are two files available that are named `rrt_star_particle.m` and `run_rrt_star_particle.m` that will be completed in this task. For implementation of the exercise in Python, open the jupyter notebook named `rrt_star_particle.ipynb`.

The same options in `opts` as in the previous implementation of an RRT for the particle model should be supported in this implementation. A bias towards the goal state is introduced in the tree expansion, governed by the parameter β. The tree is expanded at maximum K steps, where in each step the particle is moving (at most) λ m. If the distance to the sampled state is lower than λ, the new state becomes the sampled state. A threshold ε is also included, which if positive terminates the tree expansion when a node in the tree is closer than ε to the goal state. In addition, the neighborhood zone defined by the radius r should be considered in the planner in the variable `opts.r_neighbor`.

Exercise A.1. The file `rrt_star_particle.m` (Matlab) or the corresponding file `rrt_star_particle.ipynb` (Python) gives the foundation for implementation of an RRT* motion planner for the particle model. The API for the function is

```
rrt_star_particle(start, goal, world, opts)
```

where `start` is the initial state, `goal` is the desired goal state, `world` is a description of the map of the world using the class `BoxWorld` (provided in the subdirectory `Functions` in Matlab and in the file `world.py` in Python), and `opts` are the different options for the solver.

- Read through the code skeleton for this exercise that is available in the file `rrt_star_particle.m` (Matlab) or `rrt_star_particle.ipynb` (Python) carefully and make sure you understand the intention of the different available auxiliary functions (these will be useful later when completing the RRT* implementation).

In the implementation, you need to keep track of the cost for reaching the respective node (from the initial state, i.e., the tree root), also referred to as the cost-to-come. Also, there are more steps to be done each time a new node is inserted in the tree, since the new node should be connected with an edge to an existing node along the shortest possible path within a neighborhood (see function `connect_min_cost`), and in addition there is a re-wiring step for neighboring nodes (function `rewire_neighborhood`).

- Complete the implementation of the RRT* planner by filling in the missing lines of codes in the function `rrt_star_particle`.

Exercise A.2. Now we will use the implementation of the RRT* planner in `rrt_star_particle` to solve motion-planning problems for the particle model. The solution is to be implemented in the file `run_rrt_star_particle.m` (Matlab) or directly in the jupyter notebook `rrt_star_particle.ipynb` (Python). The provided code skeleton defines an example world and draws it, and thereafter defines the start and goal state, and the options needed for the RRT* planner.

- Complete the code in the file `run_rrt_star_particle.m` (Matlab) or in the file `rrt_star_particle.ipynb` (Python) such that a path is planned for the particle model to move in the defined world from an initial state to another desired goal state (as close as possible), where the aim is to minimize the path length. Use the planner implemented in the file `rrt_star_particle`.
- Plot the constructed tree and the planned path from start to goal in a diagram with the obstacles. Comment on differences compared to the results obtained with the RRT planner for the particle model.

Exercise A.3.

- Execute the RRT* tree construction for different number of iterations before it is stopped (varying the parameter `opts.K`, for the case when `opts.eps` is negative). Can you see any difference in the straightness of the paths in the tree, depending on the number of iterations performed?