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

September 7, 2020

1 OBJECTIVE

The objective of this exercise is twofold. First, two different kinds of planning methods will be investigated: sampling-based motion planning using rapidly-exploring random trees (RRTs) and motion planning using graph search on a state lattice using motion primitives. 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. In the extra assignment for higher grades in Appendix A, optimal planning using RRT* is considered. The planning will be executed both for a particle model and for a simplified kinematic car model in an environment where obstacles are located. 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.

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 2.3 in Bergman, K: "*On Motion Planning Using Numerical Optimal Control*", Licentiate Thesis, Div. Automatic Control, Linköping Univ., 2019.
- *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 implementations that is also possible. This document is written with reference to the Matlab files, but pointers towards corresponding Python equivalents are provided in the text. In the student labs, all 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 contraints into account, evaluate their properties, and get an understanding for which planning tasks that they are appropriate. Therefore, to pass this exercise you should implement baseline versions of the following motion planners:

- RRT for a particle moving in a 2D world.
- RRT for a kinematic car model in a world with two translational and one orientational degrees-of-freedom.
- 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.

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 during solving this exercise.
   - a concluding discussion.

   Submit the document in PDF format.

See Appendix A for the extra assignment needed for higher grades. The extra assignment is performed individually and is submitted by a document answering the questions stated, including suitable figures and plots. There is only pass/fail on the extra assignment and the document can not be revised or extended after the first submission.

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`, where a number of auxiliary functions and classes are available that will be useful in the implementation.

4.1 RRT with Particle Model

In the first 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, 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 up the jupyter notebook `rrt_particle.ipynb`.

Exercise 4.1. The file `rrt_particle.m` gives the fundament 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`), and `opts` contains different options for the planner.

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 δ m in the direction of the sampled state in the `Steer` function. If the distance to the sampled state is lower than δ, 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 (column number in `nodes`) for the parent of each node.

The tasks are now the following:

- Read through the code skeleton in `rrt_particle.m` 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 `ObstacleFree` 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 struct `opts`:
  - `opts.beta`: probability for selecting goal state as target state (i.e., the bias towards the goal state),
  - `opts.delta`: 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 4.2. Now we will use the implementation of the RRT planner in `rrt_particle.m` to solve motion-planning problems for the particle model. The so-

lution is to be implemented in the file `run_rrt_particle.m`. The provided code skeleton defines an example world 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` 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 the file `rrt_particle.m`.
- Plot the constructed tree and planned path from start to goal in the diagram with the obstacles. Hint: When determining the computed path, it is easiest to start with the goal node and then backtrack through the tree to the initial node 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`).

Exercise 4.3.

- 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.delta` in the `Steer` function. How does the tree structure change?
- 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?

4.2 RRT with Motion Model for a Simple 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 simple car with differential motion constraints. The model adopted is a kinematic single-track model, where the wheels on each axle have been lumped together to simplify the modeling. The car simulation model is implemented in the file `sim_car.m`. The car states are defined by the x and y coordinates, and the orientation θ. The state-space model in continuous time can be written as

$$\dot{x} = v\cos(\theta), \tag{1}$$

$$\dot{y} = v\sin(\theta), \tag{2}$$

$$\dot{\theta} = \frac{v}{L}\tan(\varphi), \tag{3}$$

where v is the velocity, φ is the steering angle of the front wheel and L is the length of the car wheelbase. In this exercise, we assume that the velocity v is constant and thus let the steering angle φ be the only control input u. 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 step in the simulation in `sim_car.m`.

In the provided code, there are two files named `rrt_diff.m` and `run_rrt_diff.m` that will be completed in this task. For implementation in Python, open up 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. These trajectory segments are stored in the struct `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 `DistanceFcn` in `rrt_diff.m`. 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 δ s (i.e., a certain time step). Note that `opts.delta` 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 4.4. The file `rrt_diff.m` gives the fundament for implementation of an RRT motion planner for the car with a kinematic motion model. The API for the function is

```
1 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`), and `opts` contains the different options for the solver.

- Read through the code skeleton in `rrt_diff.m` 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 `SteerCandidates` 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`.

Exercise 4.5. Now we will use the implementation of the RRT planner in `rrt_diff.m` to solve motion-planning problems for the kinematic car model. The solution is to be implemented in the file `run_rrt_diff.m`. 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` 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). Use the planner implemented in the file `rrt_diff.m`.

- Plot the constructed tree and the planned path from start to goal in the diagram with the obstacles. Also, plot the orientation of the vehicle in a separate diagram. Verify that it is consistent with the path in the world map. Notice that all state trajectories, including the orientation, are available in the struct `state_trajectories`.

Exercise 4.6.

- 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.delta`. How are the resulting tree and paths affected?

Exercise 4.7.

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

Exercise 4.8.

- 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 tree structure when varying this set?

4.3 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 motion model and state space is the same as in Exercises 4.4–4.8, 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, constant longitudinal speed is assumed during the motion. Moreover, only forward motion is allowed. For the search in the resulting state lattice, the implementation of graph-search strategies from Hand-in Exercise 1 will be re-used.

In the provided code, the main file is `run_lattice_planning.m`, which will be completed in this task. For implementation in Python, open up the jupyter notebook `lattice_planning.ipynb`.

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, we employ that the motion given a defined initial orientation is translational 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.

Exercise 4.9. 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 establishing the motion primitives are performed by the code provided in the file `MotionPrimitives.m` in the subdirectory `Functions`. The motion segments are computed using numerical optimization with CasADi.

- To run this module you need CasADi installed. This is pre-installed in the student labs, 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/`

  In Python, you install the CasADi package by running

```
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` and the implementation checks if the file exists to avoid re-computing them, if they already have been computed once.

Exercise 4.10. Now we will use 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`. In the first part of the code, the `MotionPrimitives` object is defined, the motion primitives are computed, and the different motion primitives are plotted. Then a world model is defined, using the same `BoxWorld` class as used in the previous tasks in this hand-in exercise.

We will now reuse the implementation 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_next, heuristic, number_of_controls)
```

The state-update function defined by `f` in `run_lattice_planning.m` uses the available motion primitives in the struct `mp.mprims`.

- Carefully study the file `next_state.m`, where the possible state transitions from a certain initial state are computed.

Similarly to the implementation in Hand-in Exercise 1, you need to define a heuristic function for use in the graph-search strategies `BestFirst` and `Astar`. This is defined in 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 `number_of_controls` to 2 and store the index for the set of motion primitives corresponding to the current orientation and the index for the particular motion primitive used in that set.

- Implement a lattice planner that computes a motion plan from an initial state to another desired final state (in the state lattice) based on the code skeleton provided in the file `run_lattice_planning.m`. Apply each of the methods from Hand-in Exercise 1, `DepthFirst`, `BreadthFirst`, `Dijkstra`, `BestFirst`, and `Astar`, for performing a lattice-based computation of a motion plan.
- Plot the resulting motion plans for the respective graph-search strategy. For determining the path corresponding to a particular sequence of motion primitives (specified by the indices in `plan.control`, where `plan` is the output struct from one of the graph-search algorithms) the function `control_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.

Exercise 4.11.

- Experiment with different types of planning tasks, i.e., modify the initial and goal states, and define different world models. Notice 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 one suitable planning task.
- Experiment with different choices of the heuristics used in `cost_to_go`. Can you find alternative functions, providing higher performance of the planner (in terms of computation time and number of visited nodes)?

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 code, there are two files available named `rrt_star_particle.m` and `run_rrt_star_particle.m` that will be completed in this task. For implementation in Python, open up the jupyter notebook `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 δ m. If the distance to the sampled state is lower than δ, 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` gives the fundament for implementation of an RRT* motion planner for the particle model. The API for the function is

```
1 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`), and `opts` are the different options for the solver.

- Read through the code skeleton in `rrt_star_particle.m` 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, 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 `ConnectMinCost`), and in addition there is a re-wiring step for neighboring nodes (see function `RewireNeighborhood`).

- 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.m` to solve motion-planning problems for the particle model. The solution is to be implemented in the file `run_rrt_star_particle.m`. 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 `run_rrt_star_particle.m` 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.m`.

- 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?