Laboratory class #6: Trajectory generation for drone racing

In this lab we will continue with the setup presented in Lab 3, extending the capabilities of the drone to execute smooth trajectories.

1. Drone race

In the lab you are going to simulate a single drone race. Given a list of gates, the objective of the drone is to go through all these gates in the least possible time (Figure 1).

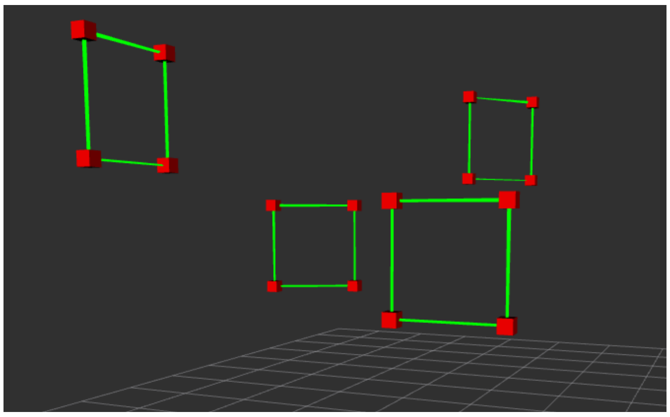
Figure 1. Example of the gates the drone needs to go through.

To do that, you already know how to simulate a drone in Gazebo (using hector_quadrotor) and how to send low-level commands to move it, either pose goals or velocity commands. You studied and implemented a global planner in a different lab, although in this case the plan is given by the position and orientation of the list of gates. In Lab 3, you also saw that direct connection between the high-level goals and the low-level inputs did not provide good results. This is specifically relevant in the context of aggressive navigation, as in a drone race. Therefore, an intermediate layer that generates suitable trajectories, functions parameterized by the time between these two modules is needed for this problem.

2. Trajectory generation software

We will rely on the already available software for trajectory generation mav_trajectory_generation[1]. The first step of the lab is to install this software and its dependencies. In a terminal execute the following orders (it will take a while to finish):

```
cd ~/catkin_ws/src
mkdir traj_gen
cd traj_gen
git clone https://github.com/ethz-asl/mav_trajectory_generation.git
git clone https://github.com/catkin/catkin_simple.git
git clone https://github.com/ethz-asl/eigen_catkin.git
git clone https://github.com/ethz-asl/eigen_checks.git
git clone https://github.com/ethz-asl/glog_catkin.git
git clone https://github.com/ethz-asl/mav_comm.git
git clone https://github.com/ethz-asl/nlopt.git
cd ~/catkin_ws
catkin build
```

[1] https://github.com/ethz-asl/mav_trajectory_generation

The mav_trajectory_generation package contains the necessary data structures and functions to generate polynomial trajectories multiple times differentiable and capable of satisfying different constraints. In the LAB, we will focus on the linear optimization problem that is described in the paper "Polynomial trajectory planning for aggressive quadrotor flight in dense indoor environments"[2].

3. Code structure

We provide you with a ROS package, "arob_lab6", that contains the starting code for the lab. The package contains a single node, called "drone_race" that you will need to complete to fulfill the tasks.

Exercise 1.

Include the package in your workspace and compile it using catkin build. To execute it, run `roscore` in one tab of the terminal. Run an instance of rviz in a second tab,

```
rosrun rviz rviz
```

and open the visualization configuration that we have provided in the package, File -> Open Config -> arob6.rviz. Finally run the drone_race node in a third console tab.

```
rosrun arob_lab6 drone_race
```

You should see something like Figure 1.

4. Visualization tools in Rviz

In the Displays panel of rviz you should see different topics published. One is the drone odometry, which at the moment is not showing anything because you have not run yet Gazebo and Hector. There are also several MarkerArray topics. As explained in the previous lab, Markers[3] are a visualization tool of rviz that helps you include visual information of different nature, from basic shapes to very complex 3D meshes. Although all the markers could be published in the same topic, we have splitted them to differentiate what they represent. In the default visualization only the topic "gate_markers" is enabled to visualize the gates.

Exercise 2.

Open the file drone_race.cpp and localize the function draw_gate_markers. Observe how to specify in the code the properties of the markers and understand how the gate is created. Change the **left** corners of the gate to make them yellow spheres with a radius of 25 centimeters.

5. Trajectory Generation code

The node we provide also contains a very simple example to illustrate how to compute a trajectory. You can see this code in the function `generate_trajectory_example`.

The trajectory requires a list of vertices, each one specifying one or more constraints. You need to specify a position constraint for all the vertices, but the value of the derivatives can either be free or fixed. In the example, the first and the last vertices have all the derivatives forced to zero, using the function `makeStartOrEnd`. However, the middle vertex only has constraints on the position and velocity.

The next step is to assign an initial time for each vertex. You can do this automatically using the function `estimateSegmentTimes` depending on the maximum velocity and acceleration of the drone,

[2] C. Richter, A. Bry, and N. Roy, "Polynomial trajectory planning for aggressive quadrotor flight in dense indoor environments," in International Journal of Robotics Research, Springer, 2016. *https://groups.csail.mit.edu/rrg/papers/Richter_ISRR13.pdf*

[3] https://wiki.ros.org/rviz/DisplayTypes/Marker

or you can set them manually. In this second case, remember that you need to specify as many times as the number of vertices minus 1.

Then, the solver computes the optimal trajectory, solving the optimization problem described in the paper. The value of N must be an even number at least twice as big as the order of the polynomial you want to compute. The example computes a minimum SNAP trajectory (polynomial of order 5), which is why N is equal to 10.

The trajectory software also includes functions to sample and evaluate the obtained trajectory at different time intervals. Although this function does not use this code, we include it because it will be helpful in the exercises you will need to do.

Finally, the node offers two marker visualizations for the trajectory: the default visualization provided in the original software, published in the topic `trajectory_vectors`, and a simplified visualization with just the positions, published in the topic `trajectory_markers`.

Exercise 3.

Toggle on the topics `trajectory_vectors` and `trajectory_markers` in rviz. Run again the drone_race node and observe the trajectory computed in the example. Observe how the trajectory changes if you comment the velocity constraint of the middle vertex. Observe also how the acceleration changes if you reduce the segment times.

Now you are ready to dive into the function `generate_trajectory`, which contains part of the necessary code to compute a trajectory given the list of gates.

Exercise 4.

Complete all the necessary code to make Hector follow a trajectory that goes through all the gates. Hector must start and end the race at the origin of coordinates. Your solution has to work with the two race tracks that we provide (gates.txt and gates_hard.txt).

You will submit the source code of your solution to Moodle (drone_race_PXX.cpp), where XX is your lab-pair number and a 1 page pdf file where you describe the main elements of your solution. You can include plots that visualize your trajectories.