Jesús Pestana <jesus.pestana@pro2future.at>
Friedrich Fraundorfer <fraundorfer@icg.tugraz.at>

Practical Part 3 – Smooth trajectory generation

ROS related theory

- Trajectory generation: polynomial interpolation to obtain a smooth trajectory, which the drone can traverse at higher speeds
- Path planning: cost-aware path simplification
- (Optional) Obstacle maps: check that the generated smooth trajectory is obstacle-free and take action if it happens to collide with obstacles
- Work on ROS:
    - ompl::geometric::PathSimplifier – cost-aware path simplification
    - mav_trajectory_generation::PolynomialOptimizationNonLinear – setting up an optimization problem whose solution is the desired smooth trajectory for the drone

Task Description

The goal of this task is to generate obstacle-free smooth trajectories based on the obstacle-free paths obtained in the practical 2. For a drone to be able to navigate a trajectory at high speed [R1], the trajectory needs to be continuous and several times differentiable. This practical deals with the generation of such smooth trajectories based on the sampled / rough trajectories provided by the software developed on the second practical.

The 2-step approach, which you are integrating and/or coding on the practicals, was first introduced by Richter et al. [R2]. In this practical, we will be using the implementation from Burri et al. [R3].

These 2 steps are:

1. (Practical 2) Calculation of an obstacle-free path using a traditional sampling-based path planner – in our case using the RRTStar or the PRMStar algorithm.

2. (Practical 3) Interpolation of a smooth enough polynomial over the obstacle-free path – meaning that the polynomial needs to be at least 4 times differentiable with a continuous 4^{th} derivative. In physics jargon this smoothness level corresponds to obtaining a trajectory with continuous snap. For context, the names of the successive derivatives of the trajectory position are speed/velocity, acceleration, jerk and snap.

Note: for a drone the acceleration is approximately set by gravity and the total propeller thrust (whose direction depends on the drone's body rotation). Knowing this, it is possible to imagine that the acceleration of the trajectory sets the body rotation over time, the jerk the body rotational velocity and the snap the rotational acceleration. A smooth trajectory ensures that the motion can be exerted by the propellers and electrical engines of the drone. As long as it is known, the heading of the drone is irrelevant, because the controller only needs to set the appropriate tilt to achieve the planned acceleration along the trajectory. These intuitions can be proofed mathematically, a good reference for this – due to its writing quality/style – is the work by Faessler et al. [R4].

Recommended steps

The coding task is sub-divided in 3 parts:

A) Cost-aware path simplification (or shortening) – see Fig. 1
B) Smooth trajectory generation – see Fig. 2
C) (Optional) Check that the smooth trajectory is obstacle-free (and modifying the planned path when it is not) – see Fig. 3

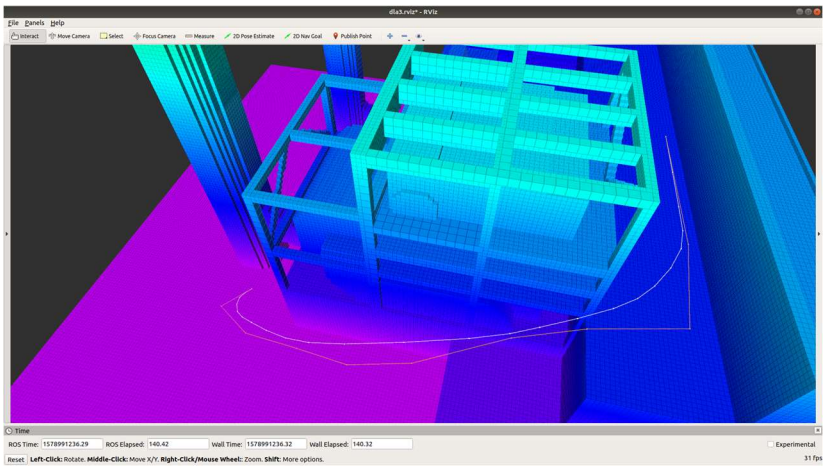

Fig. 1 Cost-aware path simplification. The orange path is the raw path planned by the PRMStar path planner and the white path is the result of the path simplification.

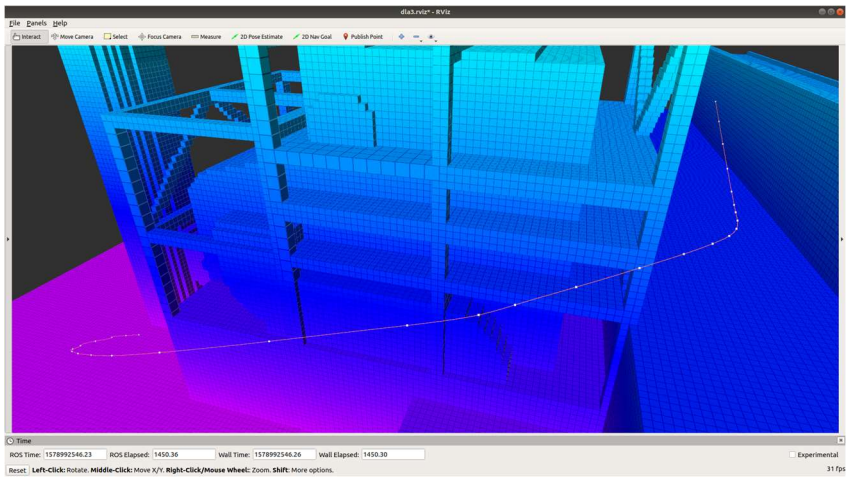

Fig. 2 Smooth trajectory generation. Using the simplified path (white points) as an input, a smooth trajectory (orange trajectory) is generated.

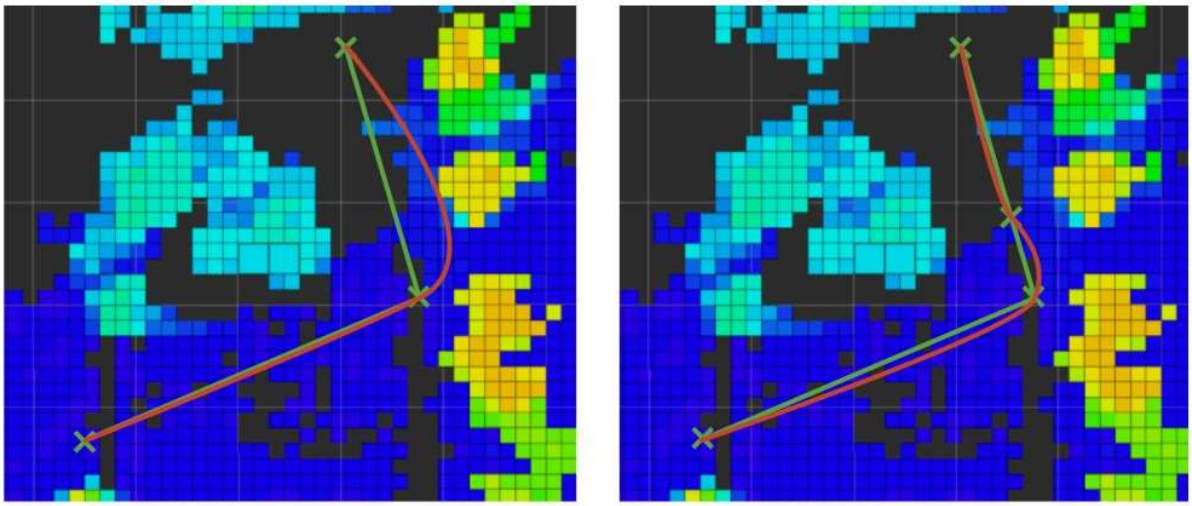

Fig. 3: Handling of collisions on the polynomial path: we project the collision onto the straight-line path and add a vertex. This pulls the polynomial towards the straight line path and slows down the trajectory.

Fig. 3: Check that the generated smooth trajectory is obstacle-free (and modifying the simplified path when it is not). Figure and caption from the paper by Burri et al. [R3].

Part A: Cost-aware path simplification

The paths calculated by the path planner algorithms RRTStar or PRMStar are made out of randomly sampled points. As a consequence, the checkpoints on these paths are only rarely aligned along straight lines, and the paths themselves take unnecessary alternating movements horizontally and vertically. Both of these characteristics would have a negative effect on the smooth trajectory generated in part B, which is the reason to perform the path simplification (or shortening). The result of the simplification should be a path that is rounded on the corners and otherwise straighter than the input raw path from the path planner algorithm.

The usual approach to obtain a smoother path for navigation is to simplify the raw path through simple operations that are calculated using only points contained in the current version of the path. Such operations may for instance take randomly 2 points of the path, and try to shortcut through them rather than taking the rest of the intervening path. Before accepting any simplification operation, the resulting path is still checked for collisions (by using the obstacle map). In addition to the collision check, a second heuristic is utilized in order to decide whether a simplification operation is accepted: the length (or the cost) of the trajectory is compared before and after the operation, and a shortening operation is accepted when it leads to a shorter (or less costly) path. In the OMPL libabry some simplification operations are “cost-aware”, meaning that they are only applied when they lead to a path cost reduction – remember, in our case this is a combination of higher clearance, shorter path length and less variation on the altitude direction. Other simplification operations in the OMPL library are accepted based only on the collision check and the path length, which tends to result in simplified

paths that approach obstacles too much around the corners of obstacles (therefore, ignoring the usage of the clearance as part of the path planners cost function – to be clear, this is something that we want to avoid).

For the cost-aware path simplification, the following steps are advised:

1. Read the code of the class ompl::geometric::Pathsimplifier, in particular try to identify which of the implemented simplification operations are cost-aware. The code for this class is implemented on the following files:

   ompl/src/ompl/geometric/PathSimplifier.h
   ompl/src/ompl/geometric/src/PathSimplifier.cpp

2. Look for example code for the class ompl::geometric::Pathsimplifier inside the OMPL library. Read the code carefully, particularly regarding the simplification operations and assess which operations are cost-aware and which are not. The example code that we recommend to use is on the file:

   ompl/tests/geometric/2d/2dpath_simplifying.cpp

3. By modifying the path planner implemented on the practical 2, include the path simplification steps as follows. The functions that will be used to simplify the raw planned trajectory (obtained in practical 2) are:

   - (collision-check + cost-aware) perturbPath, shortcutPath
   - (collision-check only) smoothBSpline.

   In our baseline solution for this task, we are using:

   - Perform 20 consecutive passes of the perturbPath and shortcutPath one after the other (in the documentation, they are advised to be used one after the other, when the cost includes the obstacle clearance).

     ```
     Do 20 passes:
         snapToVertex = 0.025; // 0.05 (less points) // 0.025 (quite good) // 0.01
     (more points) // original value: 0.005
         simplifier.perturbPath(*path, 2.0, 100, 100, snapToVertex);
         simplifier.shortcutPath(*path, 100, 100, 0.33, snapToVertex);
     ```

   - Perform 1 pass of the BSpline smoother. The Bspline smoother is utilized to provide a path with smaller inner angles (that is smaller inner angles between successive segments of the [sampled] shortened trajectory).

     Note: each pass of this smoother roughly duplicates the number of vertices, which strongly affects the runtime of the trajectory smoothing / generation routine which will be utilized in the next steps of this task. Performing only 1 pass of the BSpline smoother will provide a good trade-off between smoothness and runtime for the next part.

     ```
     simplifier.smoothBSpline(*path, 1);  // 1-2 passes
     ```

In order to visualize the simplified path together with the raw path a new version of the TrajectoryVisualization class is provided. The new implementation allows to set different colours to the markers visualized in RViz, as well as the input and output topics. Instead of changing the name of the topics inside the code, ROS has the capability of setting different topic names for publishing and for subscriptions using the remap directive on the node's launchfile. Therefore, you will publish the raw planned path and the simplified path and you will need one TrajectoryVisualization node per trajectory that is to be visualized in RViz.

Update the corresponding files on your source code (use a difftool - like "meld" - to compare the new implementation with the old one), the affected files are (contained in "dla3.zip"):

dla2_path_planner/include/trajectory_visualization/trajectory_visualization.h
dla2_path_planner/src/trajectory_visualization/trajectory_visualization.cpp
dla2_path_planner/src/trajectory_visualization_ros_node.cpp
dla2_path_planner/launch/trajectory_visualization.launch

In order to start the trajectory visualization node, execute:

roslaunch dla2_path_planner trajectory_visualization.launch marker_color:="O"

Part B: Smooth trajectory generation

This part of the practical deals with the generation of a smooth trajectory using the simplified path as an input. For this purpose, we will be using polynomial interpolation. In our case this entails the calculation of a polynomial which: (1) traverses all the waypoints of the simplified path; and (2) constrains the initial and final velocity and other derivatives to zero (drone in hovering state / flying in place).

We will be using the method (and implementation) from Burri et al. [R3], which is hosted in the github repository [SW2]. In this method, the polynomial (a piecewise high-order spline) is calculated through the non-linear optimization of a cost function that includes separate terms for:

- the minimization of the norm of the snap along the trajectory,
- the minimization of the trajectory traversal time, and
- the setting of soft constraints to enforce a maximum velocity and a maximum acceleration over the calculated trajectory. This means that there is a guarantee that the configured maximum velocity and accelerations will not be surpassed – which is a good safety measure.

For the smooth trajectory generation, the following steps are advised:

1. Read the main README.md of the repository [SW2] (note: this file is nicely displayed on the main page of the github repository). In order to better interpret the results of the trajectory generation routine, it is advisable to also read section V "Path Planning" of the paper [R3].
2. Look for example the code inside the repository that generates a 3D trajectory as discussed in the introduction of the practical. Note: the repository is organized as a ROS metapackage, where all the examples are inside the mav_trajectory_generation_example package. Another possibility is checking the test-cases of the package mav_trajectory_generation (see the file: *test_polynomial_optimization.cpp*).

The recommended example code is implemented on these files:

```
mav_trajectory_generation_example/include/mav_trajectory_generation_example/ex
ample_planner.h
mav_trajectory_generation_example/src/example_planner.cc
mav_trajectory_generation_example/src/example_planner_node.cc
mav_trajectory_generation/test/test_polynomial_optimization.cpp
```

3. Read the code carefully, paying special attention to the part of the code that deals with defining the constraints for the interpolating polynomial. Also, think about how you will be providing the simplified path as input to this part of the code.

4. Since 2 executables on this project have a header in common (example_planner.h), then: (1) make a copy of the 3 example files, (2) add them to the CMakeLists.txt as a separate executable and (3) modify them until you can compile the new executable properly (at the very least, you will need to modify the #include directives, in order to make sure that the appropriate header file is used during compilation). Name the new executable "dla3_trajectory_generator".
   Note: for step 4, do everything inside the existing "mav_trajectory_generator_example" folder of your cloned "mav_trajectory_generation" repository. Modify the "CmakeLists.txt" that is inside the "mav_trajectory_generator_example" folder and add your source and header files inside its "src" and "include" folders respectively.

5. Declare and assign a ROS subscriber that subscribes to the simplified path topic (result from part A of the practical). The trajectory generation should occur inside the callback function of this subscriber.

6. Code the trajectory generation: based on your understanding of the README.md file, the test-cases (file: *test_polynomial_optimization.cpp*) and example code (file: *example_planner.cpp*). Add the following constraints to the calculation of the interpolating polynomial:

   - Traversal of all the waypoints of the simplified path (position constraints). Note: ignore the UAV initial pose / "uav_pose" topic
   - Set the initial and final velocity and other derivatives to zero (drone in hovering state)
   - Set the maximum velocity, acceleration, jerk and snap soft constraints. In the example, the values for these constraints are provided through a parameter file; in order to use this, see the provided ROS launchfile (dla3.zip):

```
mav_trajectory_generation_example/launch/dla3_trajectory_generator.launch
```

   - The non-linear optimization parameters values that worked best for us are the following:

```
mav_trajectory_generation::NonlinearOptimizationParameters parameters;
parameters.algorithm = nlopt::LN_SBPLX;
parameters.f_rel = 0.00005;
parameters.time_alloc_method = mav_trajectory_generation::NonlinearOptimiza-
tionParameters::kSquaredTime;
```

7. In the provided extra materials (dla3.zip), you can find the code for a node that subscribes to the generated trajectory. Upon receiving a trajectory, this nodes starts evaluating the trajectory over time and publishing tf-transforms that can be visualized in ROS Rviz (see video).

```
video: Assignment3_video2.mp4
files: dla3_trajectory_sampler.h, dla3_trajectory_sampler.cc,
       dla3_trajectory_sampler_node.cc
```

To visualize the generated smooth trajectory, inside ROS RViz add a "display" of type "MarkerArray" subscribed to the topic "/trajectory_generator/trajectory_markers". The visualization will look like in Fig. 4. The visualization shown in Fig. 3 can be obtained by changing the configuration of the "MarkerArray" display to only show the "path".

Extract from the generated smooth trajectory: expected traversal time and the maximum speed, acceleration jerk and snap. Check that the soft constraints are working. Reflect on how changing the soft constraints for speed, acceleration, jerk and snap produces trajectories where the drone accelerates at the beginning and decelerates at the end of the trajectories for longer periods of time. Record a video showing the expected flight of the drone over time of the generated trajectories.

It is worthwhile to study how the input path affects the generated smooth trajectories, by changing the path simplification parameters from part A. For instance, aside from the current settings, try the following:

- Input raw trajectory from the path planner to the trajectory generation routine
- Input simplified trajectory with no BSpline smoother step. Are the generated trajectories better or worse than when utilizing the BSpline smoother with 1 and 2 passes?
- (Preferred solution) Input simplified trajectory with no BSpline smoother step and no PerturbPath step (use only some passes of the shortcutPath path-shortening function).

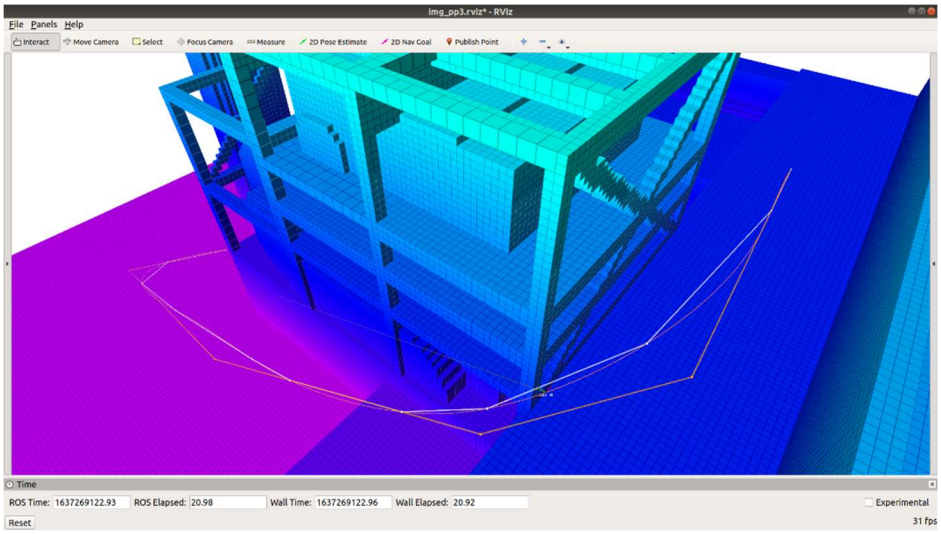

Fig. 4 Smooth trajectory generation. Using the simplified path (white path) as an input, the smooth trajectory (orange curve) is generated. The (orange path) is the raw path from the utilized path planner (RRTStar or PRMStar).

(Optional) Part C: Check that smooth trajectory is obstacle-free

As you may have noticed, the interpolating polynomial does not follow exactly the simplified path. It may well be possible that the resulting smooth trajectory collides with obstacles.

For this reason, it is necessary to check the generated smooth trajectory for collisions. This requires both executables – the path planner with the path simplification functionality and the trajectory generator – to be implemented together, so that the obstacle map is accessible after the trajectory generation step. In the case of a collision, the simplified path should be modified as described in Fig. 3 (or using some other heuristic) and then use the modified simplified path to rerun the trajectory generation. This process should be repeated until the smooth trajectory is obstacle-free, or otherwise give up after a fixed number of tries.

Although this step is necessary for a real application, it falls out of the scope of the practical due to time constraints. If you are interested on implementing this part of the task you can contact us with questions – this would be good for us, since then we could update this document for the next year based on your questions / suggestions.

Tips

When coding a ROS Node it is important to divide the work in smaller tasks that can be implemented and tested incrementally, so that the code reaches maturity at the last step, but whose parts have by then already been tested. This is the criteria that was used to design the list of steps listed above.

Software libraries (for instance the OMPL or the trajectory generation algorithm) often include toy examples to describe how the functions of the library are to be used. It is important to read them and look for relevant examples. In this case your implementation of this task can be based on the examples stated above.

The visualization in RViz and the print-out of data are very useful to be able to evaluate the quality of the obtained results.

Take screenshots of the visualization and write a small report for yourself, for the case that you need this code in the future (other than for assignment 4).

Grading Scheme, assignment 3 score: 15 out of 60 points

The submission needs to include a small report – PowerPoint or PDF slides – about your work on this assignment, along with the source-code you have generated (or submit your catkin workspace source-code folder, the “$HOME/catkin_ws/src” folder of your virtual machine).

Most of the score obtained for this assignment (75%) is decided from the following content:

- Image(s) showing successful path shortening of a trajectory generated by your trajectory planner from assignment 2.
- Image(s) showing successful generation of smooth path, from a trajectory generated by your trajectory planner from assignment 2.
- Image(s) showing successful generation of smooth path, from shortened trajectories.
- Record a video (screen recording) of the playback of one of your smooth trajectories (for instance, by using the provided trajectory sampler).
- Extract important smooth trajectory parameters, such as: trajectory traversal time, trajectory length, max. velocity, max. acceleration, max. jerk and max. snap.

Choose any 2 points to work on from the following, which will decide the rest of your score (25%):

- Image(s) showing successful path shortening (of a trajectory generated by your trajectory planner from assignment 2) and showing that the path shortening is cost-aware.
- Test the capacity of the mav_trajectory_generation library to replay a trajectory setting new values for the max. velocity and max. acceleration.
- Considering that path shortening parameters affect the shortened trajectory, benchmark the performance of the trajectory smoothing depending on either: certain path shortening parameters, certain types of shortened trajectories or certain characteristics of the shortened path. If you report on 2 different such benchmarks, they will be counted as different optional works (and are enough to achieve the rest 25% of the score).
- You can optionally discuss in detail issues you have encountered, or any other work which you have done for the assignment.

Software packages

[SW1] Open Motion Planning Library (OMPL) [R5]: https://github.com/ompl/ompl

[SW2] ethz-asl/mav_trajectory_generation repository from [R3].
Link: https://github.com/ethz-asl/mav_trajectory_generation

Videos

References

[R1] Mellinger, D., & Kumar, V. (2011, May). “Minimum snap trajectory generation and control for quadrotors.” In 2011 IEEE International Conference on Robotics and Automation (ICRA2011).

[R2] C. Richter, A. Bry, and N. Roy, “Polynomial trajectory planning for aggressive quadrotor flight in dense indoor environments,” in International Journal of Robotics Research (IJRR2016), Springer, 2016.

[R3] Michael Burri, Helen Oleynikova, Markus Achtelik, and Roland Siegwart, “Real-Time Visual-Inertial Mapping, Re-localization and Planning Onboard MAVs in Previously Unknown Environments”. In IEEE Int. Conf. on Intelligent Robots and Systems (IROS2015), September 2015.
In github: https://github.com/ethz-asl/mav_trajectory_generation

[R4] Faessler, Matthias, Antonio Franchi, and Davide Scaramuzza. "Differential flatness of quadrotor dynamics subject to rotor drag for accurate tracking of high-speed trajectories." IEEE Robotics and Automation Letters (RAL2017).

[R5] OMPL - Open Motion Planning Library (OMPL)
https://ompl.kavrakilab.org/
https://github.com/ompl/ompl
https://github.com/ompl/ompl"/blob/master/demos/OptimalPlanning.cpp
http://wiki.ros.org/ompl

Paper: Sucan, I. A., Moll, M., & Kavraki, L. E. (2012). The Open Motion Planning Library. IEEE Robotics & Automation Magazine (RAM2012).