ARO Homework 6: ICP with fast nearest-neighbor search

Karel Zimmermann, Vladimir Kubelka

2017

The basic implementation which was finished during the lab searches for nearest neighbors by iterating over all points. This approach is sufficient for demonstration on small point-clouds, but would take too much time with real-life point-clouds counting hundreds of thousands of points. Your task is to:

1. Learn what a *k-d tree* is and what is it good for.
2. Look up documentation of the SciPy implementation of the *k-d tree*.
3. Replace the *brute-force* approach to the nearest-neighbor search (used during the lab) by the *k-d tree* approach.
4. Test it on a larger point-cloud, stored in `P0_large.npy` and `Q0_large.npy`. You might want to plot only a subset of the points to make the process faster or to plot only the final result. Compare the *brute-force* and *k-d tree* approach computation time (we recommend to omit plotting for the comparison).
5. **Bonus:** The point clouds provided so far are not identical, but they were created by sampling from a single original point cloud. In practice, you often receive only partially overlapping point clouds (you have moved your sensor between the scans, so you see some new points and some are not visible anymore). You also cannot tell exactly how much you have moved. We provide sections from the large point cloud that overlap: `P0_large_secA.npy` and `Q0_large_secB.npy`. Their mis-alignment is actually not that severe, so the algorithm should work fine and converge nicely... or should it? Try and see if you can make it work. (A tip: if the algorithm does not work as fine as expected, have a look at the correspondences *idxp* and *idxq*. The *k-d tree* gives you distances for free as a return value. And you don't have to use all the correspondences...)