IASD 2021/22 Project Assignment #1: A Simple Method for Aligning Point Clouds

Pedro Miraldo and Rodrigo Ventura
Instituto Superior Técnico, University of Lisbon

(Version 1.3, October 3, 2021)

Introduction

A good perception sensor is of most importance for any intelligent agent. More and more, autonomous agent rely of 3D sensors, such as 3D LiDAR or RGB-D cameras. (See examples in Figs. 1 and 2.) While a conventional camera gives us color data, which is important for detection and classification tasks, 3D sensors have been widely used for 3D computer computer vision problems, such as agent's localization and mapping of the environment. A proof of the efficiency and robustness's of these kinds of sensors is that most of the in development autonomous driving cars rely significantly on 3D sensing for navigation and pedestrian detection and avoidance. See an example in Fig. 3. While 2D cameras give us a dense representation of the environment, making it easy to get point to point correspondences, 3D sensors such as LiDAR gave us depth information. This simplifies significantly the localization problem making it the preferable sensor for autonomous vehicles for this task.

This year's IASD project consist in three deliverables. This assignment is related to the first one, which aims at giving a local solution to the localization problem.

1 Problem Statement and Solution

In this course, students will implement an algorithm for computing the localization using 3D sensors. Two 3D scans were obtained from an agent perceiving the environment at two distinct positions and orientations. We formalized this problem as follow:

Figure 1: 3D LiDAR from Velodyne. Model Alpha puck. Image from velodynelidar.com.

Figure 2: 3D RGB-D camera from Intel. Model Intel RealSense D435. Image from www.intelrealsense.com.

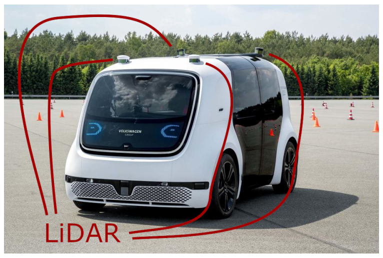

Figure 3: Picture of the VW autonomous vehicle prototype. Image from www.autocar.co.uk.

Problem 1 (Localization using 3D data) *Consider two point clouds with N points each, $\mathcal{S}_1 = \{\mathbf{p}_1, \ldots, \mathbf{p}_N\}$ and $\mathcal{S}_2 = \{\mathbf{q}_i, \ldots, \mathbf{q}_N\}$, where $\mathbf{p}_i, \mathbf{q}_i \in \mathbb{R}^3$ are points in the first and second scans, respectively. The localization (aka transformation between the first and second scans' frames) is given by finding the transformation (rotation matrix $R \in \mathcal{SE}(3)$ and translation vector $\mathbf{t} \in \mathbb{R}^3$) that aligns the points in the two scans.*

For a given set of 3D point correspondences (denoted as $\{\mathbf{p}_i \leftrightarrow \mathbf{q}_i\}$), the transformation can be computed by solving:

$$R^*, \mathbf{t}^* = \underset{R,\mathbf{t}}{\arg\min} \sum_{\mathbf{p}_i \leftrightarrow \mathbf{q}_i} \text{dist}(R\mathbf{p}_i + \mathbf{t}_i, \mathbf{q}_i), \tag{1}$$

where dist(.,.) computes the geometric distance between a pair of points. However, in addition to the sparsity of the data, point clouds suffer from a lack of ordered data. As a result, getting the point correspondences is hard and usually not directly computed in these kinds of problems. Instead, methods consider the raw (unordered sequence of points) and alternative between having a way of getting potential matches (by considering closest points between scans) and computing the transformation using Eq. 1. The computation of the closest point is given by $\mathbf{p}_i \leftrightarrow \widetilde{\mathbf{q}}_i$ ($O(N^2)$ complexity $\underline{\text{in the worse case scenario}}$), where, for a given $\mathbf{p}_i$, $\widetilde{\mathbf{q}}_i$ is the closest point in $\mathcal{S}_2$ to $\mathbf{p}_i$.

2 Objective

A simple algorithm for aligning 3D point clouds is described in Algorithm 1. Students are not implementing the complete algorithm. This assignment is only focused on developing some of its tools. A set of classes with specific modules are available for helping in the Fenix section page of the project. Specifically `registration`, `get_pointcloud`, and `visualization_vtk`:

Algorithm 1 A simple algorithm for 3D point cloud alignment

```
Input: $\mathcal{S}_1 = \{\mathbf{p}_1, \ldots, \mathbf{p}_N\}$ and $\mathcal{S}_2 = \{\mathbf{q}_1, \ldots, \mathbf{q}_N\}$
Output: $R_{\text{out}}$ and $\mathbf{t}_{\text{out}}$
1: $R_{\text{out}} \leftarrow I$                                        ▷ Identity Matrix
2: $\mathbf{t}_{\text{out}} \leftarrow \mathbf{0}$
3: while some stoping criterion isn't met do
4:     Compute $\{\mathbf{p}_i \leftrightarrow \widetilde{\mathbf{q}}_i\}$;
5:     Compute $\{R, \mathbf{t}\}$ that aligns $\mathcal{S}_1$ and $\mathcal{S}_2$ using Eq. 1 with matches $\{\mathbf{p}_i \leftrightarrow \widetilde{\mathbf{q}}_i\}$
6:     $\mathbf{p}_i \leftarrow R\,\mathbf{p}_i + \mathbf{t}, \quad \forall i$               ▷ Updates $\mathcal{S}_1$
7:     $R_{\text{out}} \leftarrow R\,R_{\text{out}}$                          ▷ Updates output rotation
8:     $\mathbf{t}_{\text{out}} \leftarrow R\,\mathbf{t}_{\text{out}} + \mathbf{t}$                     ▷ Updates translation
```

registration.py Has a registration class that gets two point clouds (NumPy arrays of dimension `(N,3)`), and computes the 3D alignment using the algorithm described above. Two modules are not implemented, the computation of the nearest neighbour correspondences $\{\mathbf{p}_i \leftrightarrow \widetilde{\mathbf{q}}_i\}$ (step number 4 of the Algorithm 1) and the computation of R and $\mathbf{t}$ that aligns the $\mathcal{S}_1$ and $\mathcal{S}_2$ with correspondences $\{\mathbf{p}_i \leftrightarrow \widetilde{\mathbf{q}}_i\}$.

get_pointcloud.py Contains a class to deal with the loading and storing of the point clouds. There is a module missing, `load_point_cloud`, which will load the point cloud from a `.ply` file.

visualization_vtk.py Contains a class for visualization purposes. No changes are necessary.

3 Assignments

Specifically, the students have to implement three modules from two different classes:

`load_point_cloud(.)` from the `point_cloud_data` class. This module get as input the file name (a `.ply`) and import the point cloud into the class' attribute `self.data`, which is a dictionary. Keys are the number of the added point, and the values are NumPy arrays of dimension (3,), namely `[x, y, z]`, representing the coordinates of the respective 3D point in the world. A guiding example on a `ply` file format and how the students should read it is shown in Sec. 4. If the data in a `ply` file is corrupted, the function should return `False`. Otherwise, return `True`.

`find_closest_points()` from the `registration` class. This module will compute the correspondences between points in scan 1 and scan 2, by for every point in scan 1 finding the closest point in scan 2. In the `registration` class, the point clouds are stored in NumPy arrays, with dimension `(N,3)`, attributes `self.scan_1` and `self.scan_2`. Correspondences must be stored in a dictionary, with keys being an integer identifying the correspondence id. Values are another dictionary, with keys `'point_in_pc_1'` and `'point_in_pc_2'`, and `'dist2'`, and values are NumPy array of dimension `(3,)` representing the 3D coordinates of a point in scan 1, 3D coordinates of a point in scan 2, and the geometric distance between these two points (i.e., `sqrt((x_1-x_2)**2 + (y_1-y_2)**2 + (z_1-z_2)**2)`), respectively. The function should return the correspondences' dictionary.

The returning output must be, therefore, something like:

```
output = {key1:value1, key2:value2, ....},
```

where `key1`, `key2`, ... are integers identifying the correspondence number and values are of the type: (example for `value1` corresponding to `key1`):

```
value1 = {'point_in_pc_1' :  numpy_array_point_scan_1,
      'point_in_pc_1' :  numpy_array_point_scan_2,
           'dist2' :  dist_between_points}.
```

`compute_pose(.)` from the `registration` class. which gets as input a variable named correspondences, which is a dictionary with the correspondence obtained from the previous equation, and outputs is a tuple with R and $\mathbf{t}$ (NumPy arrays of dimension `(3,3)` and `(3,)`, respectively) that align the point clouds according to Eq. 1. The specific method for solving Eq. 1 is described in Algorithm 2. **No alternative to Algorithm 2 is allowed.**

A submission template is presented in Sec. 7 and can be downloaded with the remaining files. **Other than the NumPy, only standard libraries from Python are allowed.**

Algorithm 2 Register 3D point clouds (solving Eq. 1)

```
Input: $\mathcal{S}_1 = \{\mathbf{p}_1, \dots, \mathbf{p}_N\}$ and $\mathcal{S}_2 = \{\mathbf{q}_1, \dots, \mathbf{q}_N\}$
Output: $R_{\text{out}}$ and $\mathbf{t}_{\text{out}}$
 1: $\overline{\mathbf{p}} \leftarrow \frac{1}{N}\sum_{i\in\mathcal{S}_1} \mathbf{p}_i$                ▷ Get the center point of $\mathcal{S}_1$
 2: $\overline{\mathbf{q}} \leftarrow \frac{1}{N}\sum_{i\in\mathcal{S}_2} \mathbf{q}_i$                ▷ Get the center point of $\mathcal{S}_2$
 3: $\widetilde{\mathbf{p}}_i = \mathbf{p}_i - \overline{\mathbf{p}}, \quad \forall i$                ▷ Frame aligned with the center of $\mathcal{S}_1$
 4: $\widetilde{\mathbf{q}}_i = \mathbf{q}_i - \overline{\mathbf{q}}, \quad \forall i$                ▷ Frame aligned with the center of $\mathcal{S}_2$
 5: $P \in \mathbb{R}^{N\times 3} = \text{stack}(\widetilde{\mathbf{p}}_i)$                ▷ Stack all $\widetilde{\mathbf{p}}_i$ in a matrix of dimension $N \times 3$
 6: $Q \in \mathbb{R}^{N\times 3} = \text{stack}(\widetilde{\mathbf{q}}_i)$                ▷ Stack all $\widetilde{\mathbf{q}}_i$ in a matrix of dimension $N \times 3$
 7: $A = Q^T P$                ▷ Get matrix $\mathbf{A}$ of dimension $3 \times 3$
 8: $U, \Sigma, V^T = \text{svd}(A)$                ▷ $U$, $\Sigma$, and $V$ using SVD, all of dimension $3 \times 3$
 9: $R_{\text{out}} = U\,\text{diag}(1, 1, |UV^T|)\,V^T$                ▷ Get the output rotation
10: $\mathbf{t}_{\text{out}} = \overline{\mathbf{q}} - R_{\text{out}}\,\overline{\mathbf{p}}$                ▷ Get the output translation
```

NOTES: svd(A) denotes the Singular Value Decomposition of a matrix and diag(x, y, z) denotes a diagonal matrix with entries x, y, and z. Check the NumPy library!

4 Input formats (ply file)

The input files containing the point clouds are of type `.ply`. Each contains a single point cloud, and is structured as follows:

```
ply
format ascii 1.0           { ascii/binary, format version number }
comment made by anonymous  { comments are keyword specified }
comment this file is a cube
element vertex 8           { define "vertex" element, 8 in file }
property float x           { vertex contains float "x" coordinate }
property float y           { y coordinate is also a vertex property }
property float z           { z coordinate, too }
element face 6             { there are 6 "face" elements in the file }
property list uint8 int32 vertex_index {"vertex_indices" is a list of ints }
end_header                 { delimits the end of the header }
0 0 0                      { start of vertex list }
0 0 1
0 1 1
0 1 0
1 0 0
1 0 1
1 1 1
1 1 0
4 0 1 2 3                  { start of face list }
4 7 6 5 4
4 0 4 5 1
4 1 5 6 2
4 2 6 7 3
4 3 7 4 0
```

These files have three parts. First, they start with a header, which contains how the data is stored in the file. Depending on the files and point cloud characteristics, the header can include more or fewer lines. Then it comes to the vertex list and the face list. The points with coordinates `[x, y, z]` are the items in the `ply` files denoted as vertexes. So, from the header, we have first to get the number of vertexes and check the order and number of items in the vertex line. After that, we will extract the relevant information (aka `x`, `y`, and `z` coordinates) from each line. Notice the vertices can have more information than the geometric position of the point (for example, color), but we are only interested in the `x`, `y`, and `z`. **All the information in the vertices are labeled with `property float`. The remaining properties (like `element face`, `property list`), and all elements after `start of face list` should be ignored!**

5 Evaluation

The deliverable for this assignment is shown through DEEC Moodle, with the submission of a single python file, called `solution.py`, implementing the modules mentioned above. Instructions for this platform are available on the course webpage. Finally, the grade is computed in the following way:

- 50% from the public tests;
- 50% from the private tests; and
- -10% from the code structure and readability.

Deadline: **8-October-2020**. Projects submitted after the deadline will not be considered for evaluation.

6 Run an example

We provide the students with a script to run the public tests (available at Fenix, project section) for at-home tests and evaluations, with a visualization tool for validation. Students need to have installed `NumPy` and `vtk` libraries. The file `run_example.py` contains a simple routine to use the previously mentioned classes and the `solution.py` to load the point cloud and visualize the alignment.

There are 8 public tests available. To run the scripts, in a terminal call:

```
$ python run_example key
```

where key is a string: `PUB1`, `PUB2`, ..., `PUB8` calling for the respective test.

7 Submision template:

```
from registration import registration
from get_pointcloud import point_cloud_data

import NumPy as np
from typing import Tuple


class registration_iasd(registration):

    def __init__(self, scan_1: np.array((..., 3)), scan_2: np.array((..., 3))) -> None:

        # inherit all the methods and properties from registration
        super().__init__(scan_1, scan_2)

        return

    def compute_pose(self,correspondences: dict) -> Tuple[np.array, np.array]:
        """compute the transformation that aligns two
        scans given a set of correspondences

        :param correspondences: set of correspondences
        :type correspondences: dict
        :return: rotation and translation that align the correspondences
        :rtype: Tuple[np.array, np.array]
        """

        pass

    def find_closest_points(self) -> dict:
        """Computes the closest points in the two scans.
        There are many strategies. We are taking all
        the points in the first scan
        and search for the closes in the second.
        This means that we can have > than 1 points in scan
        1 corresponding to the same point in scan 2.
        All points in scan 1 will have correspondence.
        Points in scan 2 do not have necessarily a correspondence.

        :param search_alg: choose the searching option
        :type search_alg: str, optional
        :return: a dictionary with the correspondences.
            Keys are numbers identifying the id of the correspondence.
            Values are a dictionaries with 'point_in_pc_1',
            'point_in_pc_2' identifying the pair of points
            in the correspondence.
        :rtype: dict
        """

        pass

class point_cloud_data_iasd(point_cloud_data):

    def __init__(self, fileName: str) -> None:
        # inherit all the methods and properties from registration
        super().__init__(fileName)

        return


    def load_point_cloud(self,file: str) -> bool:
        """Loads a point cloud from a ply file

        :param file: source file
        :type file: str
        :return: returns true or false, depending on whether the method
        the ply was OK or not
        :rtype: bool
        """

        pass
```