Assignment 4

Assignment Objective

The objective of this assignment is to estimate the location of an object we will explore how to implement a simple version of a physics simulator with frictional contact interactions in the context of planar bouncing. Here, we are given a square object in the plane with known inertial and contact parameters. The objective is to predict the trajectory of the object given the initial state of the object.

Instructions

In this assignment, we will ask you to fill out missing pieces of code in a Python script. You will submit the completed code to Canvas. Your code is passed to an auto-grader that will verify the correctness of your work and assign you a score out of 100.

- Download the assignment scripts `assignment_4.py` and `assignment_4_aux.py` from the Files menu of Canvas.
- For each "Part" of the HW assignment, fill in the missing code as described in the problem statement.
- Submit the completed `assignment_4.py` to Canvas.
- The contribution of each part to your total score is noted in the subtitle.
- The assignment is due on November 7^{th}, 2020.
- You may need the `open3d` library to visualize the point clouds. Otherwise, you can save the point clouds to a file and use a software like MeshLab to visualize them.

Part 1 – Point Cloud Transformation (10 points)

3D point clouds are a collection of 3 coordinate values (x, y, z) and 3 color (r, g, b) values. We can represent this collection of points as a $N \times 6$ matrix, where N is the number of point in the point cloud:

$$\mathbf{P} = \begin{bmatrix} x_1 & y_1 & z_1 & r_1 & g_1 & b_1 \\ x_2 & y_2 & z_2 & r_2 & g_2 & b_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_N & y_N & z_N & r_N & g_N & b_N \end{bmatrix}$$

Our objective here is to write a function that transforms a point cloud from one reference frame to another. To this end, complete the function `transform_point_cloud` in `assignment_4.py`, which takes a point cloud $\mathbf{P}$, a rotation matrix $\mathbf{R} \in SO(3)$ and a translation vector $\boldsymbol{t} \in \mathbb{R}^3$ and returns the transformed point cloud $\hat{\mathbf{P}}$:

$$\hat{\mathbf{P}} = \text{transform\_point\_cloud}(\mathbf{P}, \boldsymbol{t}, \mathbf{R})$$

where $\mathbf{P}$ and $\hat{\mathbf{P}}$ are both `(N, 6)` numpy arrays, $\boldsymbol{t}$ is a `(3, )` numpy array, and $\mathbf{R}$ is a `(3, 3)` numpy array. Note that given a point $\boldsymbol{p} = \begin{bmatrix} x & y & z \end{bmatrix}^T$ we can find its transformed point as follows:

$$\hat{\boldsymbol{p}} = \mathbf{R}\boldsymbol{p} + \boldsymbol{t}$$

Also, note that the transformation does not effect the color component.

You can test your `transform_point_cloud` implementation by uncommenting `transform_point_cloud_example` in the `__main__` function and executing `assignment_4.py`. If your implementation is right, you should see how your transformed object (in red) matches the expected position (in green).

Part 2 – Point Cloud Registration (10 points)

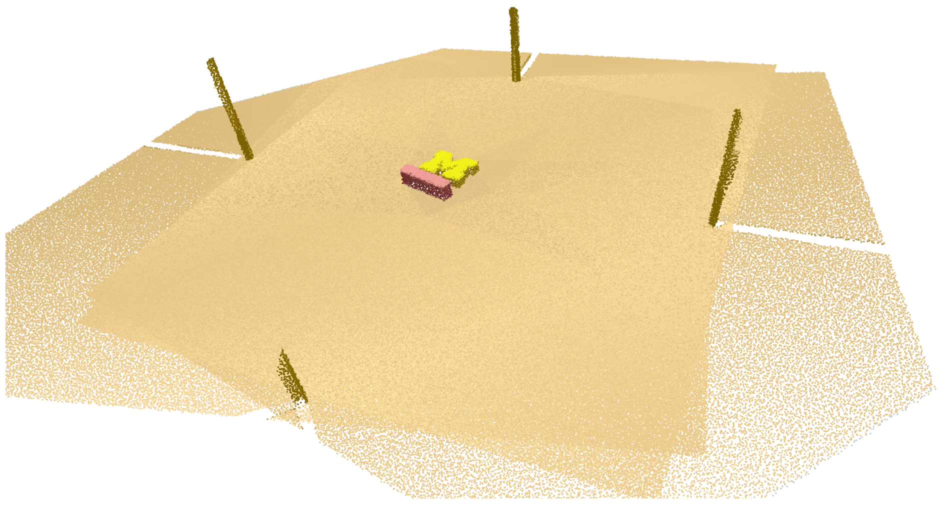
Typically, on a manipulation scene we will have multiple cameras at different view points. Multiple cameras reduce occlusions and noise effects, effects that as we will see hinder ICP performance. Assuming that the cameras have been calibrated, we can know their locations, and therefore merge their measurements into one scene.

Complete the function `merge_point_clouds` in `assignment_4.py` such that the inputs are a list of point clouds $\left[\mathbf{P}_1, \dots, \mathbf{P}_k\right]$ and a list containing their respective camera poses $\left[\boldsymbol{C}_{\boldsymbol{P}1}, \dots, \boldsymbol{C}_{\boldsymbol{P}k}\right]$. Each camera pose is a tuple composed of a translation vector and a rotation matrix $\boldsymbol{C}_{\boldsymbol{P}i} = (\boldsymbol{t}_i, \mathbf{R}_i)$. This function returns a point cloud $\mathbf{P}_{\text{merged}}$ which contains all the points from the input point clouds registered with respect to the world frame.

$$\mathbf{P}_{\text{merged}} = \text{merge\_point\_clouds}(\left[\mathbf{P}_1, \dots, \mathbf{P}_k\right], \left[\boldsymbol{C}_{\boldsymbol{P}1}, \dots, \boldsymbol{C}_{\boldsymbol{P}k}\right])$$

where $\mathbf{P}_{\text{merged}}$ is a `(N, 6)` numpy array where $N = \sum_{i=1}^{k} N_i$ being N_i the number of points in $\mathbf{P}_i$. You may find useful the function `transform_point_cloud` from Part 1.

You can test your `merge_point_clouds` implementation by uncommenting `reconstruct_scene` in the `__main__` function and executing `assignment_4.py`. If your implementation is correct, your code will merge 3 different view points of a scene composed by a yellow block M, a red box on top of a table and 4 support columns. We will use this scene again in Part 6.



Part 3 – Find Correspondences by Closest Points (15 points)

In this part, we're going to implement an important part of the ICP algorithm, whose purpose is to find the correspondences between two given point clouds $\mathbf{P}_A$ and $\mathbf{P}_B$. For each point in $\mathbf{P}_A$ we will find the closest point in $\mathbf{P}_B$. In other words, we will compute the correspondence vector $\boldsymbol{c}_{B,A} \in \mathbb{N}^{N_A}$ whose entries are defined as follows

$$[\boldsymbol{c}_{B,A}]_i = \underset{1 \leq j \leq N_B}{\arg\min} ||\boldsymbol{p}_{A,i} - \boldsymbol{p}_{B,j}||_2$$

Where $\boldsymbol{p}_{A,i} \in \mathbf{P}_A, \quad 1 \leq i \leq N_A$ and $\boldsymbol{p}_{B,j} \in \mathbf{P}_B, \quad 1 \leq j \leq N_B$

Note that both point clouds must be expressed with respect to the same reference.

Complete the function `find_closest_points` in `assignment_4.py` such that given two point clouds $\mathbf{P}_A, \mathbf{P}_B$ returns the correspondence vector $\boldsymbol{c}_{B,A}$ as described above.

$$\boldsymbol{c}_{B,A} = \text{find\_closest\_points}(\mathbf{P}_A, \mathbf{P}_B)$$

where $\mathbf{P}_A$ and $\mathbf{P}_B$ are numpy arrays of sizes `(N_A, 6)` and `(N_B, 6)` respectively, and $\boldsymbol{c}_{B,A}$ is a numpy array of size `(N_A,)`. You can assume that $\mathbf{P}_A$ and $\mathbf{P}_B$ are expressed with respect to the same reference.

Part 4 – Estimate Transformation (15 points)

This part is the core of IPC algorithm. Given two point clouds with correspondent points, we will estimate the transformation given by $\boldsymbol{t}, \mathbf{R}$ which represents the transformation from model coordinates to scene coordinates. This transformations enables us to find the pose of the model point cloud expressed in scene coordinates.

$$\boldsymbol{t}^*, \mathbf{R}^* = \min_{\boldsymbol{t}\in \mathbb{R}^3, \mathbf{R}\in\mathbb{R}^{3\times 3}} \sum_{i=1}^{N} ||\mathbf{R}\boldsymbol{p}_B + \boldsymbol{t} - \boldsymbol{p}_A||$$
$$\text{s.t } \mathbf{R}\mathbf{R}^T = \mathbf{I}$$

This optimization can be computed using singular value decomposition (SVD). The steps are:

1. Compute the point cloud centroids $\boldsymbol{\mu}_A$ and $\boldsymbol{\mu}_B$

$$\boldsymbol{\mu}_A = \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{p}_{A,i}, \quad \boldsymbol{\mu}_B = \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{p}_{B,i},$$

2. Compute matrix $\mathbf{W}$

$$\mathbf{W} = \sum_{i=1}^{N} (\boldsymbol{p}_{A,i} - \boldsymbol{\mu}_A)(\boldsymbol{p}_{B,i} - \boldsymbol{\mu}_B)^T$$

3. Decompose $\mathbf{W}$ using SVD.

$$\mathbf{W} = \mathbf{U}\mathbf{D}\mathbf{V}^T$$

4. Compute $\mathbf{R}^*$

$$\mathbf{R}^* = \mathbf{U}\mathbf{V}^T$$

5. Compute $\boldsymbol{t}^*$ given $\mathbf{R}^*$

$$\boldsymbol{t}^* = \boldsymbol{\mu}_A - \mathbf{R}^*\boldsymbol{\mu}_B$$

Complete the function `find_best_transform` in `assignment_4.py` such that given two point clouds $\mathbf{P}_A, \mathbf{P}_B$ representing the scene and the model respectively, returns the estimated transformation vector $\boldsymbol{t}^*, \mathbf{R}^*$.

$$\boldsymbol{t}^*, \mathbf{R}^* = \text{find\_best\_transform}(\mathbf{P}_A, \mathbf{P}_B)$$

where $\mathbf{P}_A$ and $\mathbf{P}_B$ are numpy arrays of sizes `(N, 6)` , and $\boldsymbol{t}^*$ and $\mathbf{R}^*$ are numpy array of size `(3,)`, and `(3,3)` respectively.

Part 5 – Perfect Model ICP (30 points)

Now that we can compute correspondences and estimate a transformation between 2 point clouds, it is time to put it all together. In this part, we will run ICP to estimate the pose of a perfect, fully-observable, noiseless Michigan's block M point cloud.

1. Complete the function `icp_step` in `assignment_4.py` that takes as input a scene point cloud $\mathbf{P}_A$, a model point cloud $\mathbf{P}_B$, and the initial proposed transformation given by $(\boldsymbol{t}_{\text{init}}, \mathbf{R}_{\text{init}})$ and returns 3 elements:

   - $\boldsymbol{t}$: Estimated position of the model point cloud $\mathbf{P}_B$ with respect to the scene $\mathbf{P}_A$
   - $\mathbf{R}$: Estimated rotation of the model point cloud $\mathbf{P}_B$ with respect to the scene $\mathbf{P}_A$.
   - $\boldsymbol{c}_{B,A}$: Vector of correspondences between point cloud $\mathbf{P}_A$ and $\mathbf{P}_B$

   $$\boldsymbol{t}, \mathbf{R}, \boldsymbol{c}_{B,A} = \text{icp\_step}(\mathbf{P}_A, \mathbf{P}_B, \boldsymbol{t}_{\text{init}}, \mathbf{R}_{\text{init}})$$

   where $\boldsymbol{t}$ is a numpy arrays of sizes `(3,)`, $\mathbf{R}$ is a numpy arrays of size `(3,3)`, and $\boldsymbol{c}_{B,A}$ is a numpy arrays of size `(N_A,)`, where N_A is the number of point clouds in the scene point cloud $\mathbf{P}_A$.

2. Complete the function `icp` in `assignment_4.py` so the ICP algorithm computes $\boldsymbol{t}$, $\mathbf{R}$, and $\boldsymbol{c}_{B,A}$.

3. Test your code using `perfect_model_icp` in `assignment_4.py`. If everything is implemented right, you should be able to recover the position of the M block. You may need to uncomment the call for this function in the `__main__` function.

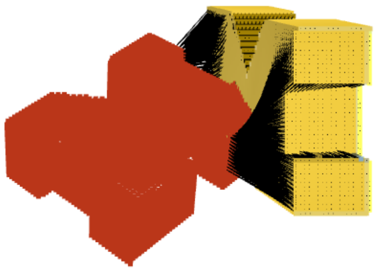


Figure 2. Perfect model ICP step visualization. In red the model point used to estimate the position of the M in scene (yellow). In black we visualize the estimated correspondences.

Part 6 – Real ICP (20 points)

In Part 5, we applied ICP to a perfect model, and hopefully we were able to recover the pose of an object. However, full observations are very hard to get in real life due to occlusions and noise. In this last part, we will face a more real problem setup, where we are given a scene point cloud as the one seen in Part 2. Again, our mission is to estimate the position of the yellow block M, but this time we will need to come up with ideas to remove noise points. To this end:

1. Implement `filter_points` in `assignment_4.py`. Given a scene point cloud, filter out points not belonging to the object. Implement your own heuristics. You can filter by color or positions. You can assume that the object (x, y) coordinates will always be within the square $[-0.6, -0.6] \times [0.6, 0.6]$. The better the filtering, the better results will ICP obtain.

2. Modify as needed `custom_icp` in `assignment_4.py`. You can use functions implemented in previous parts or implement new ones. For example, you can implement new functionality such as RANSAC or other algorithms to get rid of outliers in the transformation estimation. You are free to choose the strategy to follow. Only two constrains:

   - Do not use external libraries.
   - Do not change your already implemented functions from previous parts, as those will be graded separately.

To test you implementation, you can uncomment `real_model_icp` in the `__main__` function.