CS498 Intelligent Robotics, Spring 2020
Homework 5: Plane Extraction

Due date: 4/23/2020

Instructions: This assignment is to be completed *individually*. You are allowed to discuss the problems with your classmates, but all work must be your own. You are not permitted to copy answers, take written notes, or look up online solutions.

All coding will be done on the Klamp't Jupyter system. Setup is as follows:

1. Login and open up a terminal under the `Run` tab.
2. If this is your first time, run `git clone https://github.com/krishauser/cs498ir_s2020.git; cs498ir_s2020/setup.sh`. If this is not your first time, run `cs498ir_s2020/update.sh` to get the updated homework code. Go ahead and close the terminal tab, and return to the file browser. The homework code is in `cs498ir_s2020/HW5`.
3. When you run the notebooks, check the Kernel menu to make sure you are using the `cs498ir-virtualenv` kernel.
4. Make sure to periodically backup your code to your local machine. To do this, select the notebook in the file browser and choose `Download`. (We don't expect to have the server die, but Engineering IT won't make any guarantees!)

You may implement your calculations manually using Klampt, Numpy, or the elementary operations found in the Python math module, according to your personal preference.

To submit, upload your written answers to Gradescope by April 23 before 9:45am, and upload your programming answers in the form of Jupyter notebook files (*.ipynb) onto the class http://learn.illinois.edu site.

Overview

Like HW5, it is recommended to *start with programming* and move on to the written questions.

In the HW5 folder, you will find a folder "hannover1" which contains a subset of an outdoor laser scanning dataset (http://kos.informatik.uni-osnabrueck.de/3Dscans/hannover1/). Running the first few cells of the notebook you will be able to see a point cloud as the scanner has moved around some buildings.

Note: In this assignment, it is possible that you will start some long-running processes. To kill a long-running process, use the "Kernel > Interrupt" function. Please be conscious of the finite amount of memory available on the system, and avoid clobbering the server with massive models, long-running processes, and infinite loops. When you are done, don't just close your browser window, but *manually*

interrupt the kernel if it is running (dark dot in the upper right of your screen) before you log out of the system.

Programming Problem A: RANSAC

The first few cells of HW5 load a point cloud from the dataset. A point cloud is a Numpy array of size N x 3, and to learn how to work with Numpy arrays it will be helpful for you to look through the quick start guide here: https://docs.scipy.org/doc/numpy/user/quickstart.html.

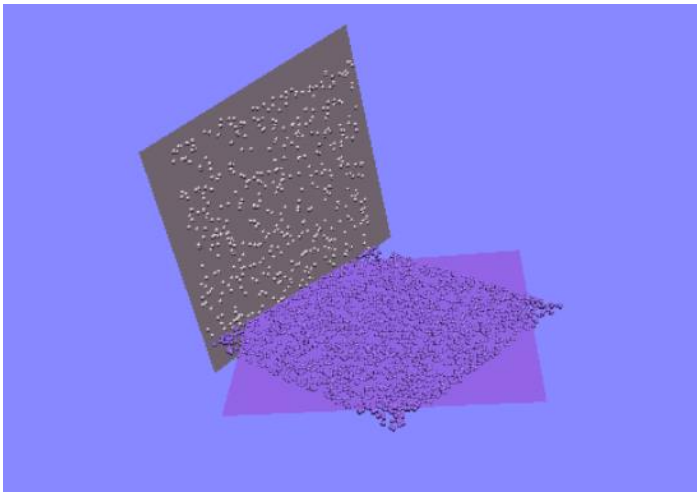
The `fit_planes_ransac` in the third-to-last cell uses the RANSAC algorithm to select a variable number of planes (*k*) from the point cloud. It also produces an *assignment* of each of the *N* points to planes, which is just a list of *N* plane indices, where each index is either -1 (indicating no plane assigned),or in the range 0…,*k*-1.

If you run all of the cells, you will see the results of RANSAC on a synthetic dataset. Scroll back up to the Klamp't visualization to examine the results. (If you run the last cell multiple times, you will see slightly different results each time).

Currently, the way the algorithm works is that it repeatedly: 1) Picks 3 random points. 2) Fits a plane through those points. 3) Inspects all of the other points. All other points within a given distance of the plane are considered inliers. 4) If the number of inliers exceeds a threshold, then the plane is kept.

An issue with this implementation is that each point can be an inlier to multiple planes. Indeed, it will produce dozens of planes even though the dataset only consists of two planes (see the print-out on the last cell).

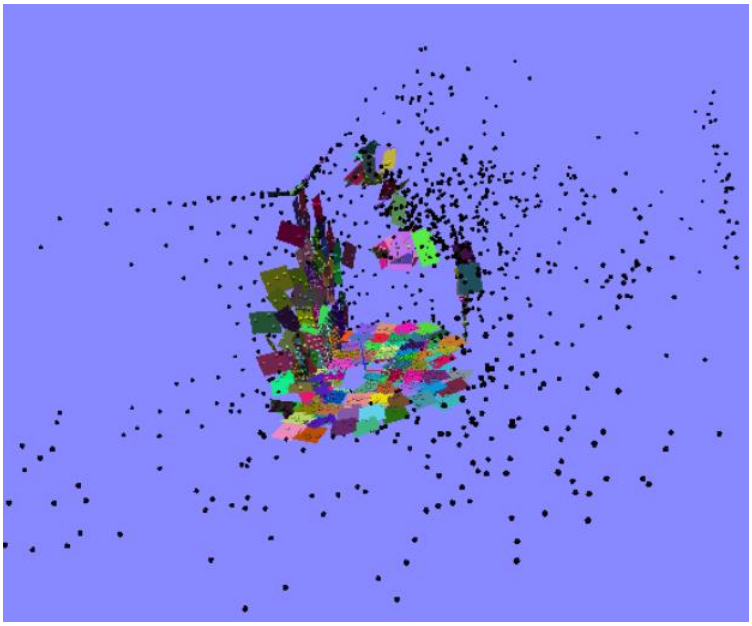
Modify the algorithm so that it never considers a point to be an inlier of more than 1 plane. Be sure to disregard prior inliers when sampling the 3 random points. Your result should only give 2 or 3 planes when running on this synthetic dataset, like the image shown below.



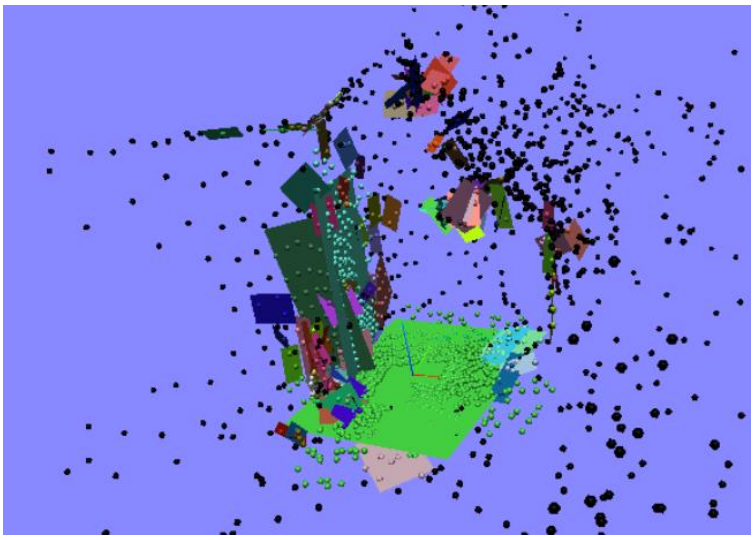
Programming Problem B: Region Growing

[Aside: If you wish, you can try RANSAC on the Hannover dataset point cloud to observe the planes that are extracted. It should be clear that these leave a lot to be desired.]

The `fit_planes_grid` function implements for you the start of a region growing technique. In the current implementation, it divides the point cloud into cells, and uses the RANSAC algorithm to generate planes in each cell. Uncomment the second or third lines in the last cell, and observe the output. You should see several small planes generated, mostly around where the point cloud has the highest density of points:



Your task is to grow these planes into a smaller number of aggregated planes using a region growing technique. The code where you should do this is marked for you, and make sure to read the comments carefully. For each plane, you should examine planes in the neighboring cells. If any of those planes is "sufficiently close", then you should use the `merge_planes(i,j)` function to merge them together. This process should be repeated until there are no remaining planes to be merged. The output will look something like this:

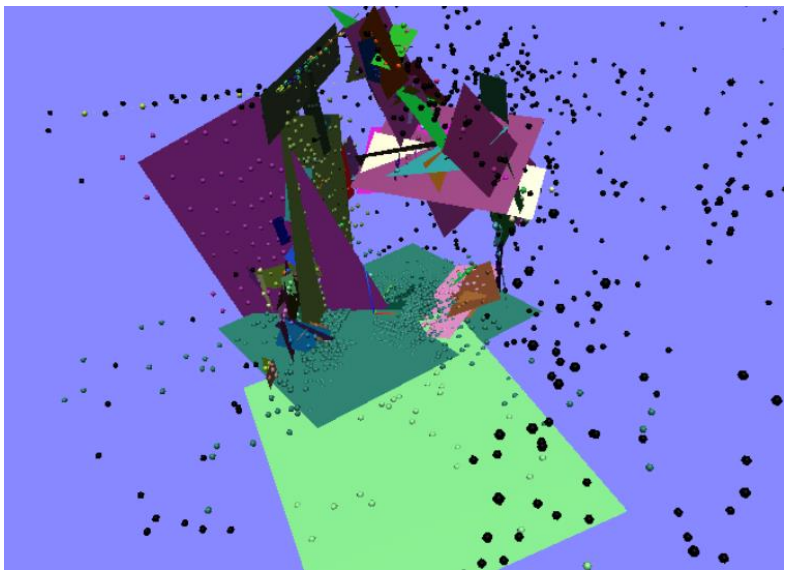


Programming Problem C. Region Growing Toward Outliers

One of the limitations of region growing just between planes is that it misses any cells that don't contain sufficient numbers of points for RANSAC to detect.

Expand upon your implementation in Problem B such that the region growing method examines neighboring cells to each plane, and adds *unassigned* points if they are sufficiently close to the plane. The `PlaneFitter.add_point` method should be used here. If you expand a plane to include outliers in a cell, continue to add points from neighboring cells.

The result may look something like this. Although the rectangles don't do a great job capturing the shape of each flat surface, the color of points shows the groupings that are produced by the algorithm. It shows that the two main walls of the building and the ground plane are properly identified, as well as some other significant features.



Written Problem 1: Online Plane Fitting

The `PlaneFitter` class performs "online" plane fitting, which can dynamically add new points and merge with existing other point clouds. In both cases, it updates the best-fit plane in O(1) time.

A. Explain how the centroid $\hat{p} = 1/N \sum_i p_i$ can be updated when a new point p_{N+1} is added. The result should be the centroid of all N+1 points.
B. Explain how the centroid $\hat{p} = 1/N \sum_i p_i$ can be updated when a group of M points with centroid $\hat{q} = 1/M \sum_j q_j$ is added. The result should be the centroid of all N+M points.
C. In the lecture slides, we showed that the best-fit plane to a set of points $p_1, \ldots, p_N$ could be computed using a singular value decomposition $A = UWV^T$ with A the Nx3 matrix with the i'th row being p_i minus the centroid. But instead, the `PlaneFitter` class computes the 3x3 covariance matrix of the points $Cov[p] = \sum_i (p_i - \hat{p})(p_i - p)^T$, where the sum is over the outer products. It then computes the eigendecomposition of this matrix, $Cov[p] = Q\lambda Q^T$. Use the properties of the singular value decomposition to show how these are related. In particular, how is Cov[p] related to A? How is Q related to V?

Written Problem 2: Complexity of RANSAC

A. To analyze the performance of the RANSAC method we may need to consider several parameters, including the number of points N, the number of iterations M, the threshold ϵ for considering a point to be an inlier, and the number of inliers needed to keep a point K. For the basic RANSAC algorithm (without your enhancements in programming problem A), what is its running time?
B. With your enhancements from programming problem A, including a point in at most one plane, subsequent iterations of RANSAC may take fewer iterations. Perform an output-sensitive analysis of the running time of your algorithm. Specifically, suppose that RANSAC produces H planes in total. How many point-to-plane distance checks and plane fits will it use? What other overhead does it need to perform? What conditions are necessary for the asymptotic complexity in this case to be different from the result in part A?