GEO-1015, Assignment-3: The best shape detection algorithm is...

Ming-Chieh Hu (6186416), Daan Schlosser (5726042)
Neelabh Singh (6052045) & Lars van Blokland (4667778)

January 17th, 2025

Contents

1 Introduction

1.1 Overview of the assignment and objectives

In this assignment, we evaluate three popular shape detection algorithms - RANSAC, Region Growing and Hough Transform - to determine the best approach for identifying planes in spatial datasets.

The primary objectives of this report are:

1. To understand the underlying principles and ideas of the three algorithms.
2. To implement and refine these algorithms using real-world datasets and to assess their performance.
3. To compare the algorithms based on metrics such as accuracy, execution time, and robustness.
4. To provide a final recommendation on the best algorithm for plane detection for AHN4 data.

1.2 Used dataset

In this assignment we use the given BK-City dataset (and a subset) which contains a 3d point cloud of TU Delft's Bouwkunde faculty building, where the ground and nearby vegetation has been removed. This building is part of the AHN4 dataset.

2 Methodology

2.1 RANSAC

We start by implementing the basic RANSAC algorithm in the terrain-book. The algorithm works but there are many improvements that can be made, which will be mentioned in this and the next sections.

2.1.1 Improved RANSAC

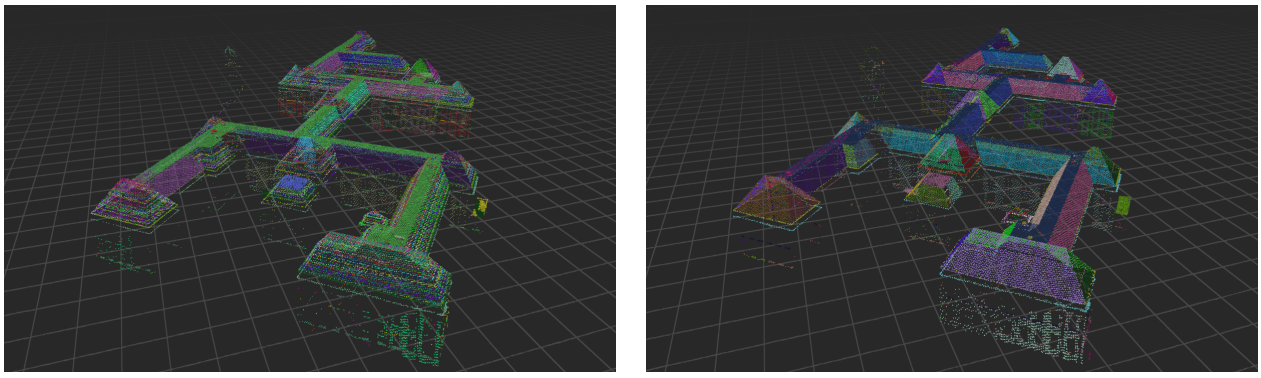
From the results of the BK city dataset, we observed some unexpected horizontal planes (Figure 1.a). Although these planes met the minimum inlier point threshold, they weren't the intended outcomes. To address this, we developed an improved selection process to better guide the function towards the desired results (Algorithm 1). The new selection process follows a two-step procedure. First, it randomly selects a single point. Then, it selects three additional points from the neighbors of the first point and use them as parameter to construct plane. By doing so, the selected 3 seed points are more likely to lie on the same plane, effectively reducing the occurrence of unwanted horizontal planes. Note that the search radius for this process is set to 5 meters, a moderate scale suitable for buildings.

Algorithm 1: Improved RANSAC algorithm using nearest neighbor

Input: An input point cloud P, a *mask* to determine whether a point has been categorized or not, the error threshold ϵ, and the number of iterations k

Output: The set of points C_{best}, the score s_{best}, the detected plane instance $\mathcal{I}_{best}$

```
s_best ← 0;
C_best ← 0;
I_best ← nil;
for i ← 0 ... k do
    p_0 ← 1 randomly selected point from P;
    M ← 3 randomly selected point from neighbors(p_0);
    I ← plane instance constructed from M;
    C ← ∅;
    forall p ∈ P do
        d ← distance(p, I);
        if d < ε then
            if p is not masked then
                Add p to C;
    s ← score(C);
    if s > s_best then
        s_best ← s;
        C_best ← C;
        I_best ← I;
return C_best, s_best, I_best
```



(a) Simple RANSAC (b) Improved RANSAC

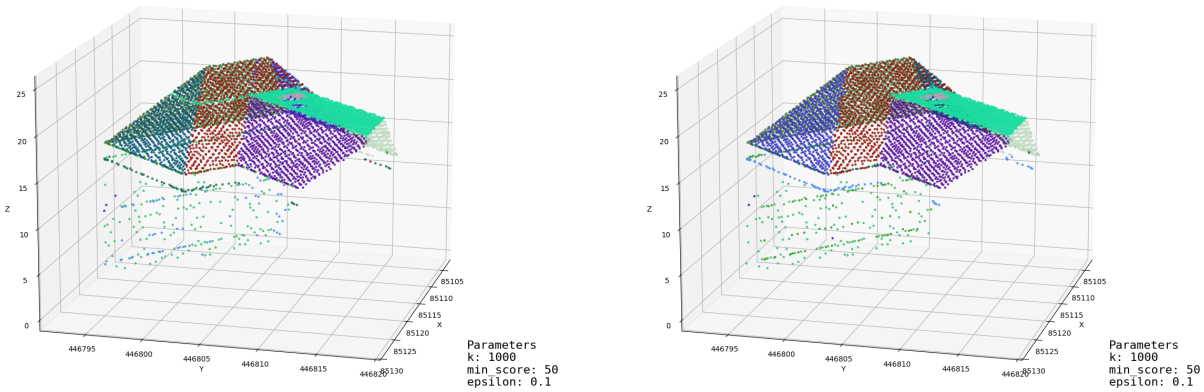
Figure 1: Different implementation of RANSAC outcomes (without using post-processing and clustering functions). In these 2 pictures we use parameters: $k = 1000$, $s_{min} = 300$, and $\epsilon = 0.1$.

2.1.2 Post-processing function

Up to this point, we have successfully eliminated the unwanted horizontal planes. However, the model still generates some unintended 'stripes' near the intersection of planes, which is a special case. This issue arises from the mask we designed; while it simplifies the classification process, it also limits our ability to handle this scenario. To address this, we propose a post-process (as Algorithm 2) that identifies points near plane intersections and re-classifies them based on the class of their nearest neighbors. It is clearly shown in Figure 2 that the 'stripes' are eliminated by the post-processing function. Note that the nearest neighbors algorithm here searches within a radius of $5 \times \epsilon$, the 5 here is a parameter *multiplier* we added and exposed to users.

Algorithm 2: Post-processing function

Input: An input point cloud P with id as its 4th dimension, list of detected plane instance $L_{\mathcal{I}}$, the error threshold ϵ
Output: Point cloud P with id as its 4th dimension
forall $\mathcal{I} \in L_{\mathcal{I}}$ **do**
  $collison \leftarrow \emptyset$;
  forall $p \in P$ **do**
    $d \leftarrow \text{distance}(p, \mathcal{I})$;
    if $d < \epsilon$ **then**
      if p *doesn't have same id as* $\mathcal{I}$ **then**
        Add p to $collison$;
  forall $p \in collison$ **do**
    $N \leftarrow \text{neighbors}(p, (5 \times \epsilon))$;
    $p[3] \leftarrow$ majority class found in N;
return P



Without post-process

With post-process

Figure 2: Outcomes before and after post-processing. (To better showcase the 'stripe' we use BK subset here.) In these 2 pictures we use parameters: $k = 1000$, $s_{min} = 50$, and $\epsilon = 0.1$.

2.1.3 Cluster by distance

Some points classified by RANSAC may belong to the same large plane, even though they are not connected and are located far apart. In such cases, we aim to separate these points into distinct surfaces to more accurately represent the building's geometry. While DBSCAN is a powerful algorithm for this task, we observed significant variations in surface density due to the LiDAR sensor's shot angle. To address this issue more effectively, we propose a clustering function (Algorithm 3) that relies solely on the distances between clusters, rather than their density.

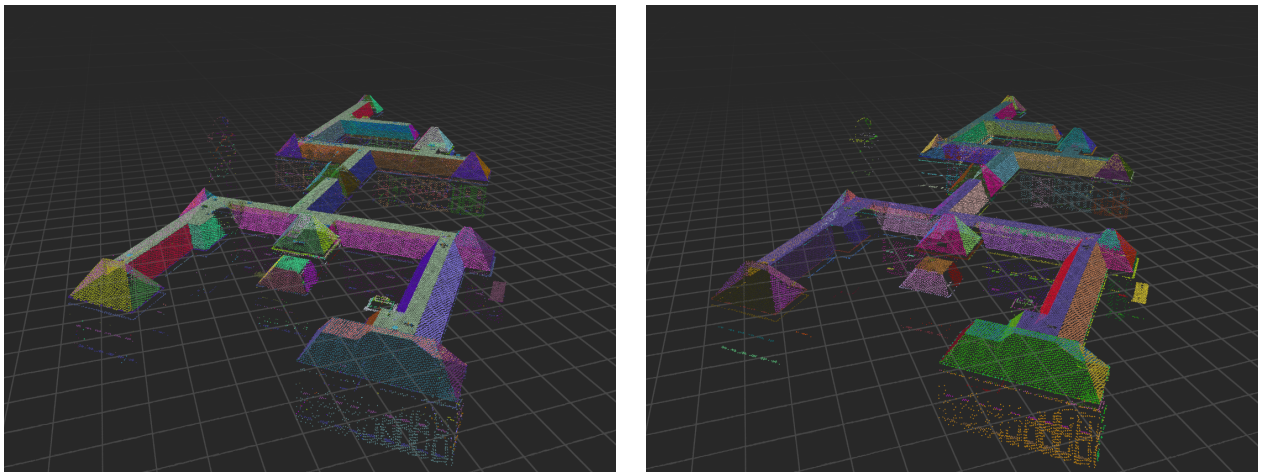
This function introduces 2 new parameters into our method: the distance threshold δ and n_{min}, the minimum score required for a cluster to be considered valid. Note that, when this function is applied, the outcome may

include clusters with surface inliers less than s_{min}.

Algorithm 3: Cluster by distance

Input: An subset of point cloud P_s in a specific plane (with id as its 4th dimension), a minimum score n_{min} for a surface to be considered valid, the distance δ defining the radius of a neighborhood
Output: Point cloud P_s with id as its 4th dimension

```
todo ← [];
id ← 1;
P_s[..., 3] ← 0;
forall p_s ∈ P_s do
    if p_s is not labeled as 0 then
        continue
    Append p_s to todo;
    while todo is not empty do
        current ← pop(todo);
        N ← neighbors(current, δ);
        forall p_s ∈ N do
            if p_s is labeled as 0 then
                size ← size + 1;
                Append p_s to todo;
                p_s[3] ← id;
    id ← id + 1;
    for i ← 0 ... id do
        P_i ← {all points p_s labeled as i};
        if size(P_i) < n_min then
            P_i[..., 3] ← 0;
return P_s
```



Without clustering | With clustering

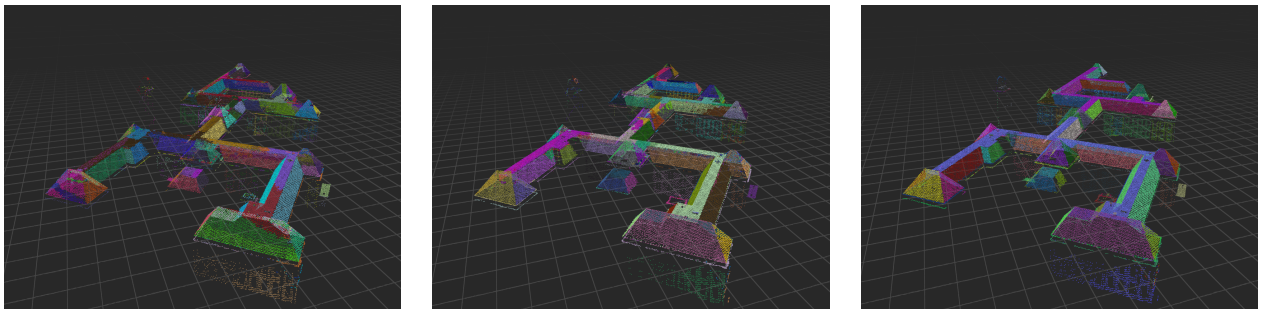
Figure 3: Outcomes before and after distance clustering. In these 2 pictures we use parameters: $k = 1000$, $s_{min} = 300$, $\epsilon = 0.1$, $multiplier = 5$. Clustering function uses $\delta = 4.0$, $n_{min} = 10$.

2.1.4 Parameters and their effects

In the simplest form of RANSAC, there are three key parameters: the number of iterations k, the minimum score s_{min} required for a plane to be considered valid, and the error threshold ϵ, which defines the boundary for determining whether a point belongs to a plane.

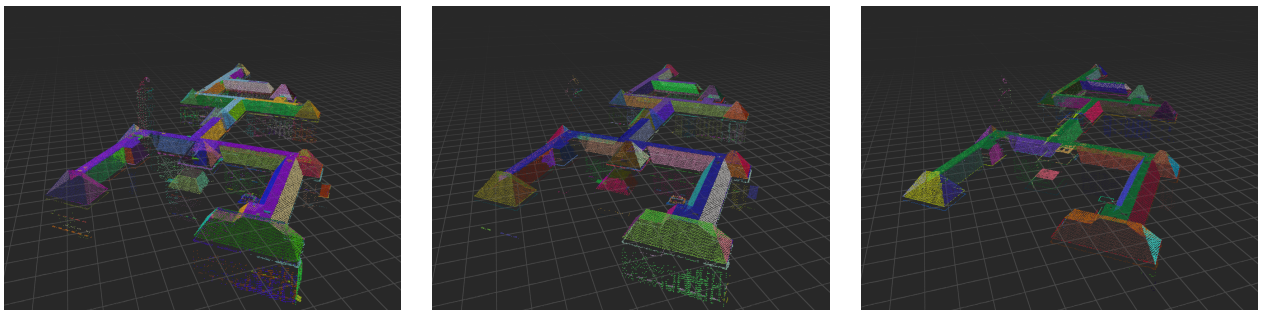
Original RANSAC parameters
As long as the computer has sufficient processing power, a larger value of k generally yields better results. However, if k is set too low, the number of iterations may be insufficient to identify the "best" plane, resulting in an outcome that resembles a collage of poorly segmented planes (Figure 4). Beyond a certain point, increasing

k further will reach a bottleneck where it no longer improves the results. s_{min} is a critical parameter that requires careful selection. A larger s_{min} will ignore smaller, trivial surfaces, while a smaller s_{min} will include more trivial surfaces (Figure 5). However, a smaller s_{min} may also increase processing time. Similar to s_{min}, ϵ must also be chosen carefully. A larger ϵ often results in cascade-like patterns, while a smaller ϵ may create a camouflage-like figure due to its low tolerance for variation (Figure 6).



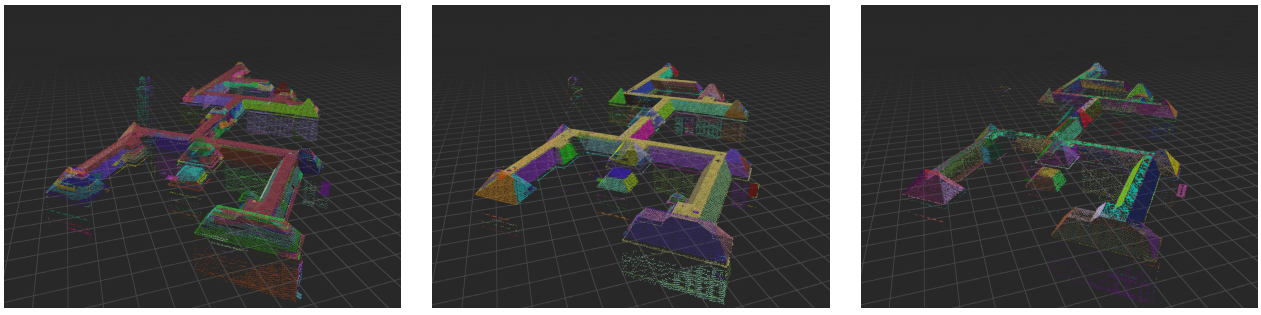
$k = 20$, 312 surfaces found | $k = 100$, 226 surfaces found | $k = 500$, 246 surfaces found

Figure 4: RANSAC with different k values. In these 3 pictures, s_{min} is set to 300 and ϵ is set to 0.1.



$s_{min} = 100$, 396 surfaces found | $s_{min} = 200$, 225 surfaces found | $s_{min} = 400$, 142 surfaces found

Figure 5: RANSAC with different s_{min} values. In these 3 pictures, k is set to 500 and ϵ is set to 0.1.



$\epsilon = 0.4$, 459 surfaces found | $\epsilon = 0.2$, 344 surfaces found | $\epsilon = 0.05$, 277 surfaces found

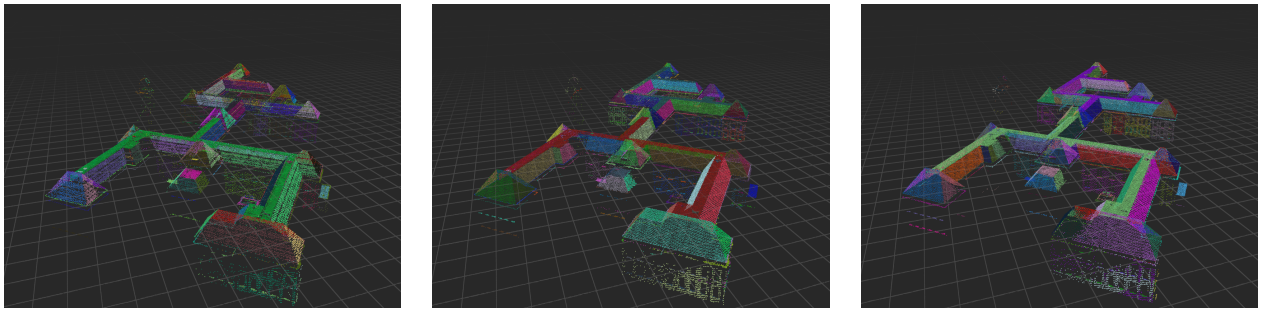
Figure 6: RANSAC with different ϵ values. In these 3 pictures, k is set to 500 and s_{min} is set to 300.

New parameters used in post-processing and clustering

The *multiplier* parameter is used to define the search radius for our post-processing function. The radius is calculated as *multiplier* $\times \epsilon$ to perform the nearest neighbors query. Choosing a reasonable value for the multiplier is straightforward. For example, in the BK city dataset, we set $\epsilon = 1$ and *multiplier* $= 5$, resulting in a 0.5-meter search radius, which works well with the point density and distribution of the dataset. Choosing an excessively large *multiplier* can lead to false classifications of points, while selecting a value that is too small may prevent some points from finding their neighbors, causing them to be excluded from the surface (Figure 7).

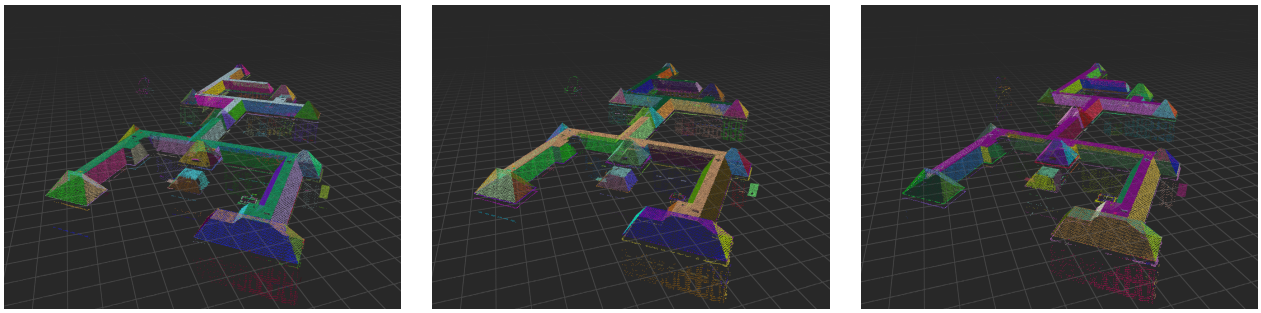
The parameters δ and n_{min} are integral to the distance-based clustering function. A large δ may prevent the program from properly separating planes into distinct surfaces (Figure 8), while a small δ can result in the

generation of numerous trivial surfaces. Similarly, n_{min} significantly affects the resulting surfaces. A larger n_{min} eliminates more small clusters, whereas a smaller n_{min} preserves these trivial clusters (Figure 9).



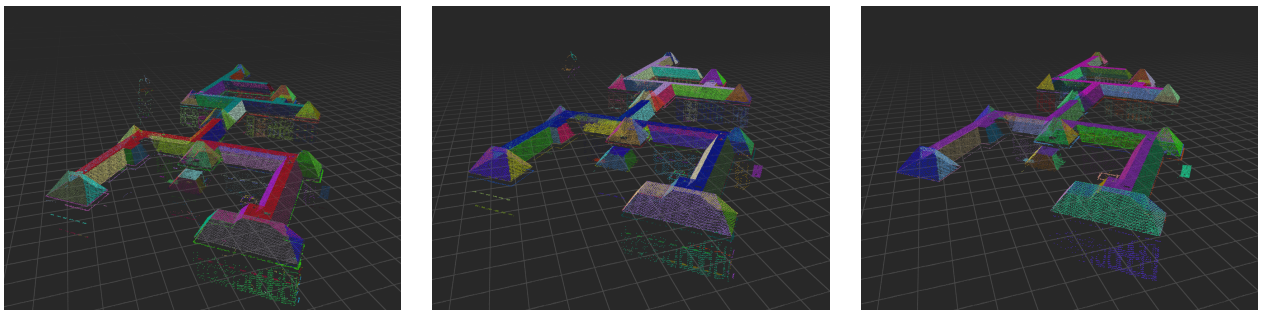
multiplier = 1, 266 surfaces found | *multiplier* = 5, 253 surfaces found | *multiplier* = 20, 266 surfaces found

Figure 7: RANSAC with different *multiplier* values. With $k = 500$, $s_{min} = 300$, $\epsilon = 0.1$, $\delta = 4.0$, $n_{min} = 10$.



$\delta = 2$, 266 surfaces found | $\delta = 4$, 213 surfaces found | $\delta = 8$, 208 surfaces found

Figure 8: RANSAC with different δ values. With $k = 500$, $s_{min} = 300$, $\epsilon = 0.1$, *multiplier* = 5, $n_{min} = 10$.



$n_{min} = 5$, 364 surfaces found | $n_{min} = 25$, 167 surfaces found | $n_{min} = 125$, 106 surfaces found

Figure 9: RANSAC with different n_{min} values. With $k = 500$, $s_{min} = 300$, $\epsilon = 0.1$, *multiplier*, $\delta = 4.0$.

2.1.5 Time Complexity

We estimated the time complexity of algorithm 1, 2, 3 with $n =$ the number of points in point cloud P.

The time complexity of our RANSAC is $O(|L_{\mathcal{I}}| \times (kn + n \log n))$, where k is the iteration number and $|L_{\mathcal{I}}|$ is the number of 'plane' instances before distance clustering ($|L_{\mathcal{I}}| \ll n$). The time complexity of post-process is $O(|L_{\mathcal{I}}| \times (n + c(\log n + m)))$, Where c is the number of points in *collision* ($c < n$) and m is the number of neighbors within search radius. The time complexity of distance clustering is $O(n \log n + nm)$. Where m is the number of neighbors within δ.

Since we have $m \ll n$, $c < n$, combining them sequentially yields: $O(|L_{\mathcal{I}}| \times (kn + n \log n))$.

2.2 Region Growing

2.2.1 Algorithm explanation

The Region Growing algorithm is a method for extracting regions from a point cloud by iteratively expanding regions from seed points based on a similarity criterion. In our implementation, we focus on 3D point clouds and use normal vectors as the primary criterion for region growing. A region R grows by examining neighboring points of its members and adding candidate points c that satisfy a similarity criterion. For plane detection, this criterion involves checking if the angle between the normal vector of c and the region's normal is below a threshold. If the angle is small, c is added to R; otherwise, it is ignored. The process continues until no more compatible points are found, at which point the algorithm moves to the next seed point to grow a new region.

Algorithm 4: Region Growing Algorithm

Input: An input point cloud P, a list of seed points L_S, a function to find the neighbours of a point *neighbours*()
Output: A list with detected regions L_R

```
L_R ← [];
for each s in L_S do
    S ← {s};
    R ← ∅;
    while S is not empty do
        p ← pop(S);
        for each candidate point c ∈ neighbours(p) do
            if c was not previously assigned to any region then
                if c fits with R then
                    Add c to S;
                    Add c to R;
    Append R to L_R;
return L_R
```

2.2.2 Seed point selection

The selection of seed points is a critical operation within the Region Growing algorithm that identifies the starting points for region expansion. Numerous methods have been proposed for selecting seed points, but our implementation commences with normal and planarity computation for each point in the point cloud P. From the original point, the algorithm looks for k-nearest neighbors using a KDTree, centers this neighborhood, and computes the covariance matrix. Then, Principal Component Analysis (PCA) extracts the normal vector (the eigenvector associated with the smallest eigenvalue) and computes planarity as $\frac{\lambda_2 - \lambda_3}{\lambda_1}$, where λ_1, λ_2, and λ_3 are the eigenvalues in descending order. The normals are oriented upwards, and the planarity values are stored. In the second step, points are sorted by planarity in descending order, and the top N points (where $N = \max(\text{min\_seed}, 0.02 \times \text{total\_points})$, with min_seed being a new parameter for the minimum number of seed points) are selected as seed points. These seed points will begin the region growing process, ensuring that regions start with the most reliable planar points.

Algorithm 5: Seed Point Selection for Region Growing

Input: - A point cloud P from a LAZ file;
- Parameters: **begin**
  - k: Number of nearest neighbors;
  - max_angle: Angle threshold in degrees;
  - min_seed: Minimum number of seed points to select;

Output: - A list of seed points for region growing;

Step 1: Compute Normals and Planarity begin
  for *each point in points* **do**
    Find k nearest neighbors using KDTree;
    Center the neighborhood by subtracting the centroid;
    Compute covariance matrix of the neighborhood;
    Perform PCA to get eigenvalues and eigenvectors;
    Extract normal vector (eigenvector of smallest eigenvalue);
    Ensure normal points upward (positive z);
    Compute planarity $= (\lambda_2 - \lambda_3)/\lambda_1$;
    Store normal and planarity for each point;

Step 2: Select Seed Points begin
  ]

Sort points by planarity in descending order;
Select top N seed points ($N = \max(min\_seed, 2\%$ of total points);

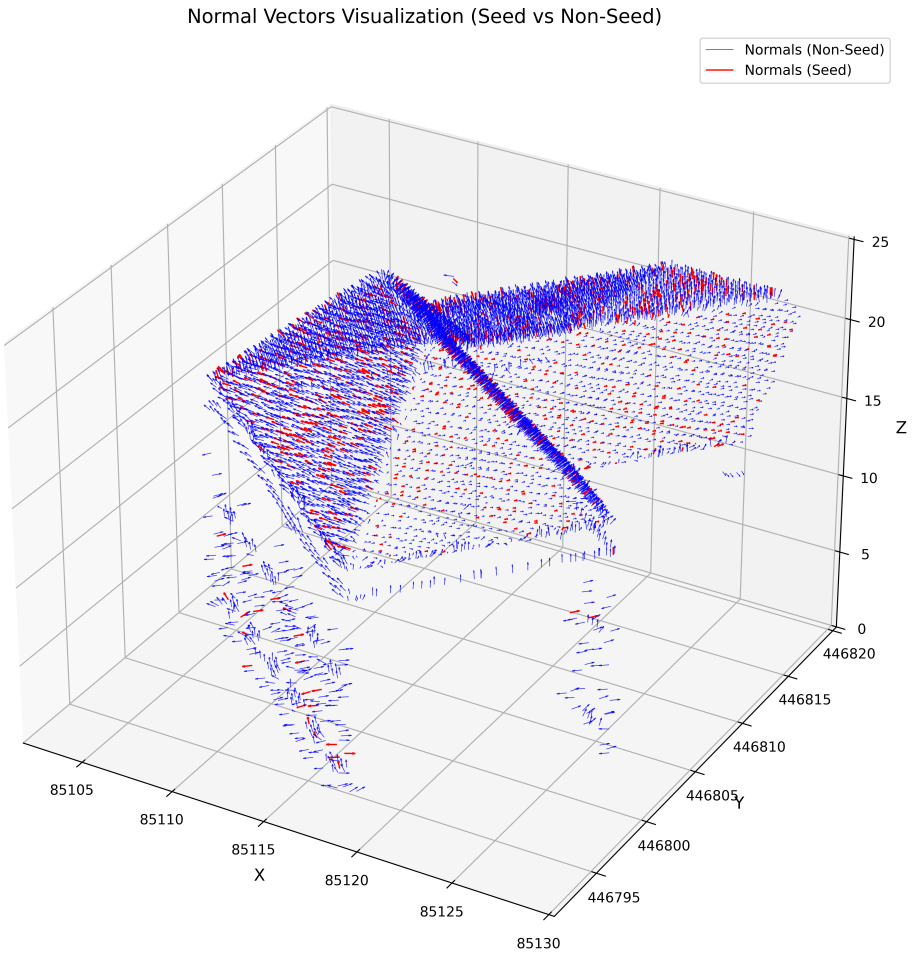


Figure 10: Normal vectors of points and selected Seed points

2.2.3 Implementation and modifications

Our method retains the core idea of Region Growing while introducing further improvements. This base algorithm uses normal vectors and angle-based similarity criteria to determine the membership of region, while KDTree data structure is implemented for efficient nearest neighbor searching in large point clouds. The key modifications we applied are as follows:
1. Dynamic normal vector updates: Apart from using static normals, we will update the average normal vector of the region every time a point is added to the region. This adopts the least-coupled approach to ensure a good coherence of the regions detected and improves the plane detection.
2. Region size filtering: The minimum size threshold for a region will be set to eliminate very small sections of the detected region from processing due to noise. Although this introduces yet another parameter, this measure very effectively increased the quality of the planes detected by reducing the number of discarded planes.

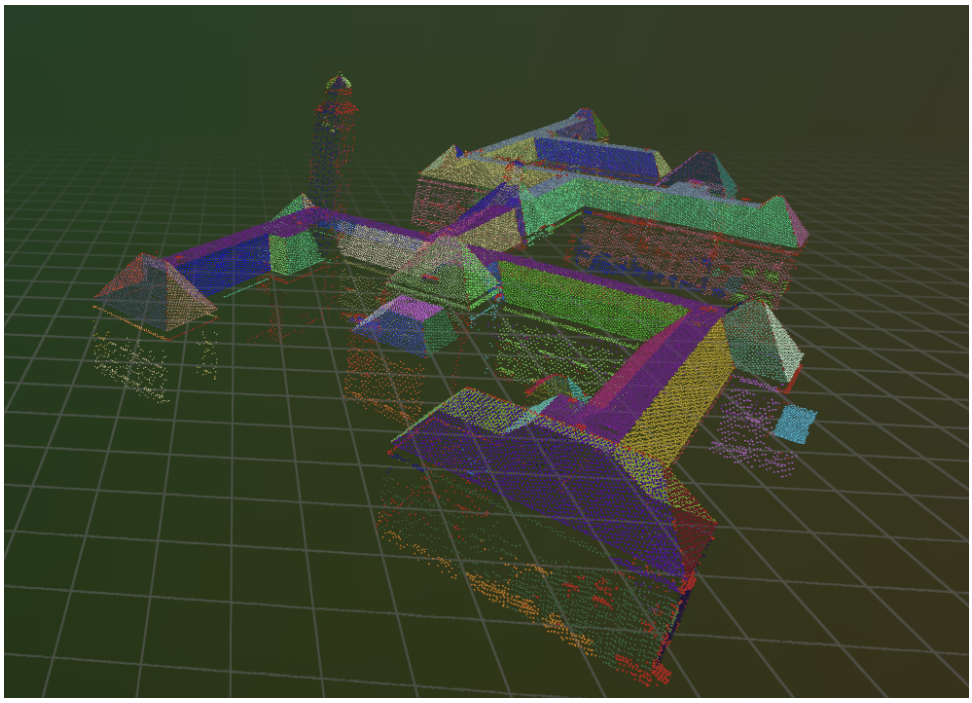


Figure 11: Outcome for BK city having 165 planes. Above we use parameters: $k = 25, max\_angle = 30, min\_seeds = 20000, min\_region\_size = 10$.

Algorithm 6: Region Growing using Normal Similarity

Input: - *points*: A NumPy array of shape $N \times 3$ containing point coordinates (x, y, z);
- *normals*: A NumPy array of shape $N \times 3$ containing normal vectors for each point;
- k: Number of nearest neighbors to consider;
- *max_angle*: Maximum allowed angle (in degrees) between normals for region growing;
- *tree*: A KDTree built from the points for efficient nearest neighbor search;
- *seed_points*: A list of seed points to start region growing;
- *min_region_size*: Minimum number of points in a region

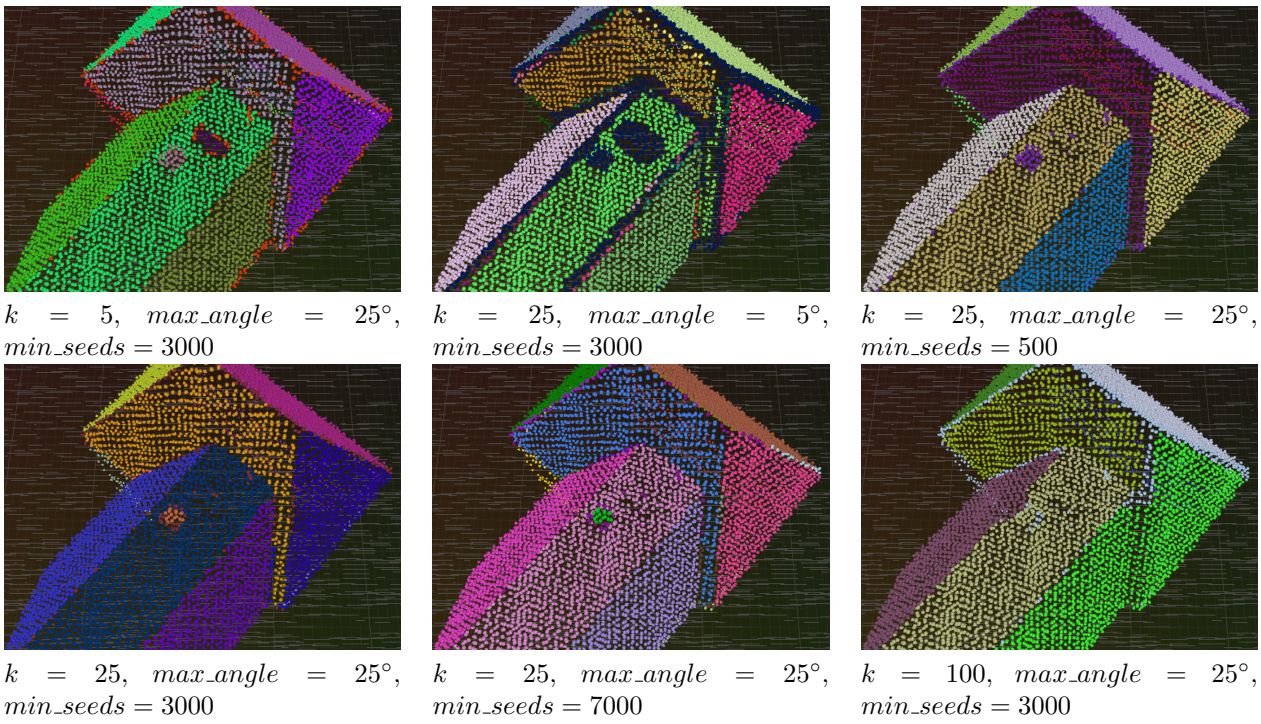
Output: - A NumPy array *segment_ids* of shape N containing segment IDs for each point;

Region Growing begin
- Convert *max_angle* to radians: *max_angle_rad* = np.deg2rad(*max_angle*);
- Initialize: **begin**
  - *processed* = array of False (size = number of points);
  - *regions* = [] (to store detected regions);
  - *min_region_size* = 10 (minimum points for a valid region);
- **for** *each seed point seed in seed_points* **do**
  - **if** *processed*[*seed*] *is True* **then**
    - Skip to the next seed point;
  - Initialize: **begin**
    - $S = \{seed\}$ (stack of points to process);
    - $R = \emptyset$ (current region);
    - *region_normals* = [] (normals of points in the region);
  - **while** S *is not empty* **do**
    - Pop a point p from S;
    - Find $k + 1$ nearest neighbors of p using *tree*;
    - **for** *each neighbor c in neighbors (excluding p)* **do**
      - **if** *processed*[*c*] *is False* **then**
        - Compute the region normal: **begin**
          - **if** R *is not empty* **then**
            - *region_normal* = mean(*region_normals*);
            - Normalize *region_normal*;
          - **else**
            - *region_normal* = *normals*[*seed*];
        - Compute the angle between *region_normal* and *normals*[*c*];
        - **if** *angle* < *max_angle_rad* **then**
          - Add c to S;
          - Add c to R;
          - Mark c as processed;
          - Add *normals*[*c*] to *region_normals*;
  - **if** *size of* $R \geq$ *min_region_size* **then**
    - Add R to *regions*;
    - Print region size for debugging;

Assign Segment IDs begin
- Initialize *segment_ids* = array of zeros (size = number of points);
- **for** *each region in regions* **do**
  - Assign a unique segment ID to all points in the region;
- **return** *segment_ids* ; `// Return the final segment IDs`

2.2.4 Parameters and their effects

In the simplest form of Region growing, there are two key parameters: the number of number of nearest neighbours k, the $minimum_angle$ difference required for a normal to be considered for determining whether a point belongs to a plane.



$k = 5$, $max\_angle = 25°$, $min\_seeds = 3000$ | $k = 25$, $max\_angle = 5°$, $min\_seeds = 3000$ | $k = 25$, $max\_angle = 25°$, $min\_seeds = 500$

$k = 25$, $max\_angle = 25°$, $min\_seeds = 3000$ | $k = 25$, $max\_angle = 25°$, $min\_seeds = 7000$ | $k = 100$, $max\_angle = 25°$, $min\_seeds = 3000$

Figure 12: Region Growing segmentation with various k, max_angle, and min_seeds values.

Core Parameters The neighborhood size, k, controls the number of neighbors considered for normal estimation. This parameter significantly impacts the smoothness of the computed normals and, consequently, the quality of the segmentation. We observed that a range of 5 to 30 neighbors generally produces optimal results. Values of k smaller than 5 resulted in noisy normals, while values exceeding 45 caused over-smoothing.
The angular threshold, max_angle, defines the maximum permissible angle between the normals of two adjacent points for them to be part of the same region. Testing revealed that lower values, such as 1° to 3°, led to excessive over-segmentation, where even minor variations in plane orientation caused unnecessary splitting.

Additional Parameters The min_region_size parameter filters out regions with fewer than 10 points, effectively eliminating noise and small, irrelevant segments, this can be exposed to user and be free to change based on the users knowledge of dataset. Meanwhile, min_seeds determines the minimum number of seed points required for initiating region growth. This parameter is crucial in avoiding premature or excessive segmentation. By default, it is set to the (where $N = \max(\text{min\_seed}, 0.02 \times \text{total\_points})$, where n is the total number of points in the dataset.

Conclusion The optimal parameter configuration for Region Growing segmentation is highly dependent on the dataset's characteristics and the specific requirements of the analysis. Based on our findings, values of $k = 25$, $max\_angle = 30°$, and $min\_seeds = 3000$ yielded the best results for typical architectural datasets.

2.2.5 Time Complexity

The three main components of time complexity of our region growing algorithm are as follows. The first one is $O(n \times k \log n)$ for the normal estimation, where n and k are as usual the number of points in the point cloud and the number of nearest neighbors used for the PCA respectively. The seed point selection process requires $O(n \log n)$, and it arranges the points on the basis of the planarity. The region growing complexity is $O(|LS| \times |R| \times k \times (\log n + |R|))$, where $|LS|$ changes, but is usually contained within (min_seed) or 2% of the total points., and $|R|$ is the average region size. As $|R| \ll n$ and k is constant, we can take an average-case complexity in sequence, which is $O(n \log n)$.

2.2.6 Optimization impact on time complexity

In our implementation, which deviates from the terrainbook's implementation, we precompute all neighbors in the k-d tree. This results in only having to query the k-d tree once before the loop, instead of querying the k-d tree each time it loops. This improves the total runtime significantly, resulting in an overall runtime of around 220x faster.

2.2.7 Worst-case Time Complexity

In the worst case, the point cloud forms a flat wall, or when the angle of the wall is lenient, to make matters worse, $|R|$ cannot be tightly restrictive and may grow to become as big as n with many seed points ($|LS| \approx n$). It eventually leads to $O(n^2 \log n)$ for input size n.

2.3 Hough Transform

2.3.1 Algorithm explanation

The Hough Transform detects planes in point clouds by generating a large number of candidate planes, which are then voted on by points depending on whether or not a point lies on or near a plane. Candidate planes with more votes than others are assumed to be a better description of the true planes.

2.3.2 Implementation

Plane generation Planes are initially generated in spherical coordinates using the parameters Theta, Phi and Rho. Theta and Phi are both in degrees and signify the angle interval between each plane. Rho is in meters and signifies the distance between each plane. Combining all values in the parameter intervals results in an array of triplets, where each triplet describes a single plane. This array is converted into two new arrays: The first contains points in Cartesian space, which are the points on the planes closest to the origin. The Second array contains the normal vectors for each of these planes. These arrays are respectively `plane_points` and `plane_normals.`

Accumulator voting Our accumulator consists of an array, with at every index a sub-array for each plane. For every point the algorithm computes the number of planes that lie within Epsilon distance of the current point. To compute this distance, the point is subtracted from the `plane_points` array, which results in an array of vectors that go from the current point to the plane. Then we compute the dot product of this new array and the `plane_normals` array. The result is an array of values, where every value indicates the distance between the current point and each of the planes. Planes that have a distance smaller than epsilon have a vote added in the accumulator.

Plane consolidation After voting has concluded it is necessary to extract the final planes from the accumulator. The underlying assumption is that planes with more votes are more likely to be correct. The consolidation step works by iterating over the planes in the accumulator, starting with the plane that got the most votes. All points found by this plane are removed from all other planes. Planes that now have less votes than the parameter `Alpha` are removed from the accumulator. Then the accumulator is resorted and the process repeats with the now second largest plane. When this loop terminates, each point will only be contained by a single plane.

2.3.3 Modifications

Point cloud chunking One of the main issues encountered when processing the entire BK dataset was a "layer cake" (see 18) like affect caused by horizontal planes dissecting the entire point cloud. Since these planes would find large numbers of points, they would be prioritized over legitimate planes. Our solution was to split the point cloud up into separate chunks, so that each chunk could be processed individually and then merged. This ensured that the horizontal planes would be outvoted by more desirable ones. As an additional bonus, processing the point cloud in chunks also results in a significant speedup.

Bounding box based plane removal To ensure that every point has the potential to be discovered by one of the planes, we generate many more planes than are necessary. While the voting process will eventually eliminate any unnecessary planes, a large number of these planes will never acquire a single vote. Many of these zero-vote planes can be detected by doing an intersection test between the planes and the bounding box of the point cloud. If a plane doesn't intersect with the bounding box of a point cloud, it will never intersect with any of its points.

Intermediate vote based plane removal Assuming that the point cloud was shuffled before voting, planes acquire votes fairly uniformly during the voting period. This means that voting trends can be used to determine which planes will eventually be relevant and which won't. The `Acceleration_factor` parameter is used by the algorithm to determine when to pause during the voting process. The median number of votes of the most popular planes is then used to cull a large number of unpopular planes. This vastly speeds up the algorithm, as significantly less planes need to be considered for the remainder of the points.

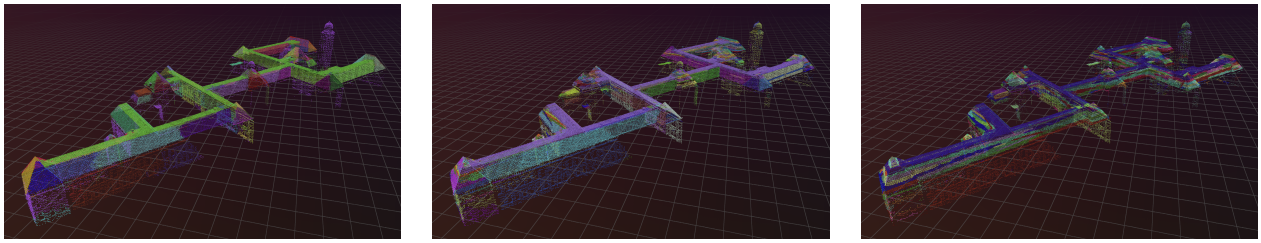
Plane re-processing After acquiring planes for each chunk, these planes need to be merged together. Although each chunk uses the same set of planes, different chunks find subtly different planes for sections of points that should be defined by a single plane. This results in a "patchy" classification (see 19). To fix this, the entire algorithm is performed again using only the planes found by processing the chunks. Due to the lowered number of planes this step is much faster than the initial pass. The resulting classification has less total planes and isn't as "patchy".

Plane cleaning Since the generated planes are infinitely large, they may "cut" through sections they aren't supposed to. During the consolidation process larger planes take points from smaller ones, meaning that a large plane could steal points from a smaller one (see 20). To fix this, we perform a final cleaning step. A KD-Tree is made using the classified points, which is used to find the closest neighbors for each point. Every point the counts the number of neighbors belonging to a different plane. If enough neighbors belong to a different plane, the current point is reclassified.

2.3.4 Parameters and their effects

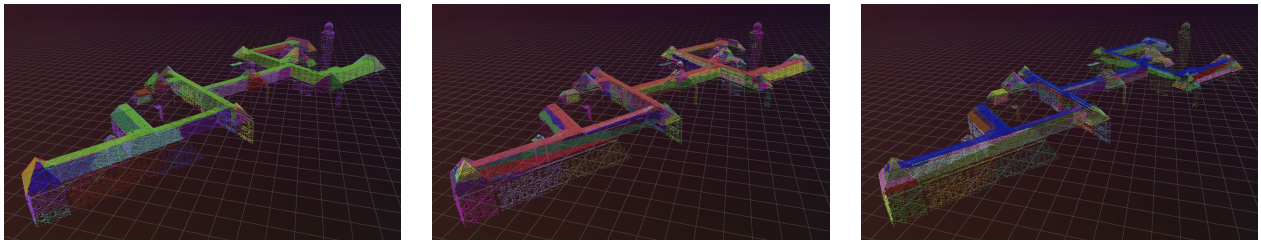
Hough Transform makes use of the following parameters:

- `Theta`, `Phi` and `Rho` which describe the number of planes and their distribution across the space.
- `Alpha` which describes the minimum number of points a plane needs to be a plane
- `Epsilon` which describes how far away a point is allowed to be from a plane before it is no longer a part of it.
- `Chunk size` which describes how large each chunk of points is in the x and y directions.
- `Acceleration factor` which describes at which point in the voting process the accumulator is culled of useless planes.
- `Reprocessing` whether or not the planes are reprocessed after merging the chunks together.
- `Cleaning distance` which describes the maximum distance a neighbor can have when performing plane cleaning.
- `Cleaning neighbors` which describers how many neighbors are evaluated when performing plane cleaning.



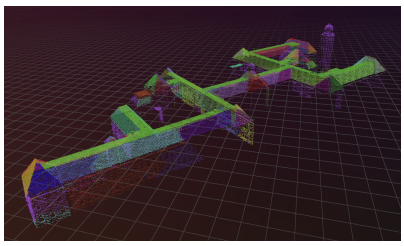
`Epsilon` is 0.15 `Epsilon` is 0.30 `Epsilon` is 0.45

Figure 13: Hough Transform with different `Epsilon` values.

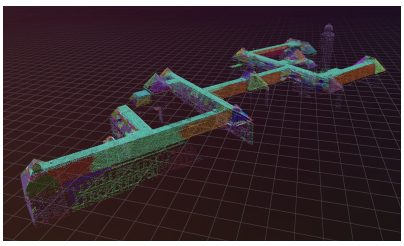


`Theta` is 10 `Theta` is 20 `Theta` is 30

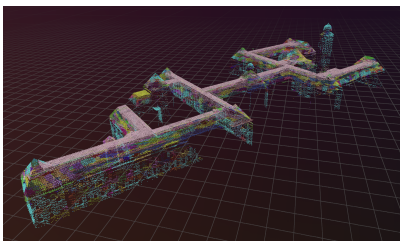
Figure 14: Hough Transform with different `Theta` intervals.



`Phi` is 10

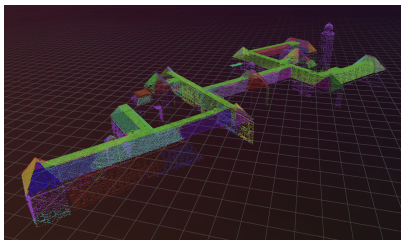


`Phi` is 20

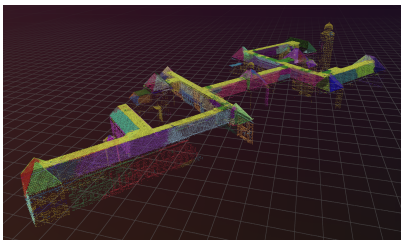


`Phi` is 30

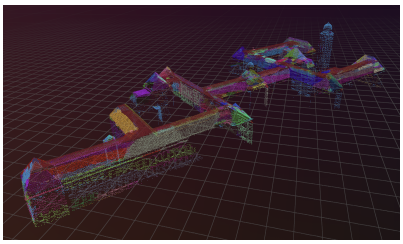
Figure 15: Hough Transform with different `Phi` intervals.



`Rho` is 10

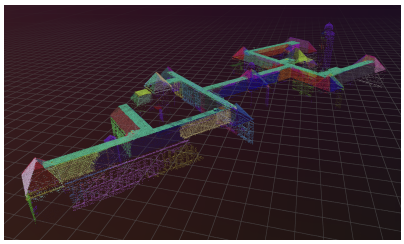


`Rho` is 20

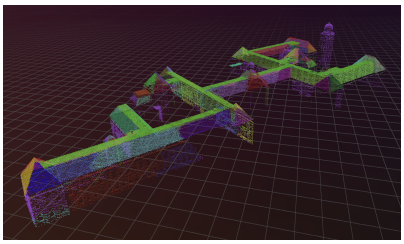


`Rho` is 30

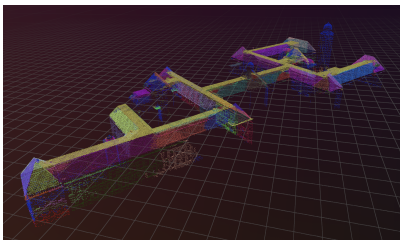
Figure 16: Hough Transform with different `Rho` intervals.



Chunk size is 10

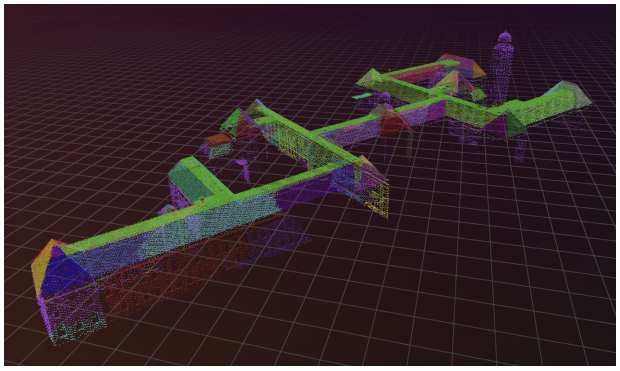


Chunk size is 25

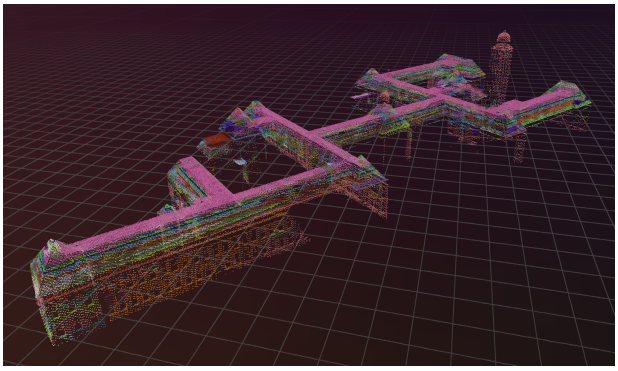


Chunk size is 40

Figure 17: Hough Transform with different chunk sizes.

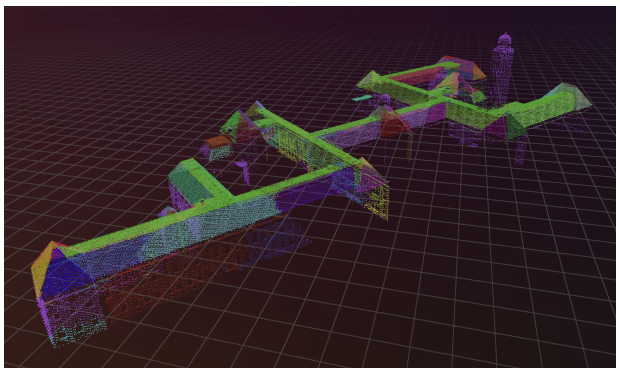


Chunking is enabled

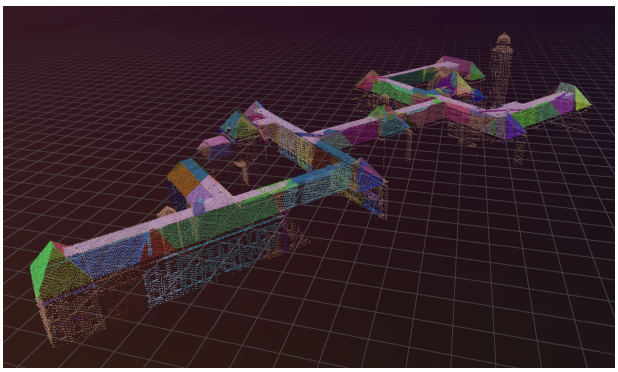


Chunking is disabled

Figure 18: Hough Transform when chunking of points is enabled and disabled.

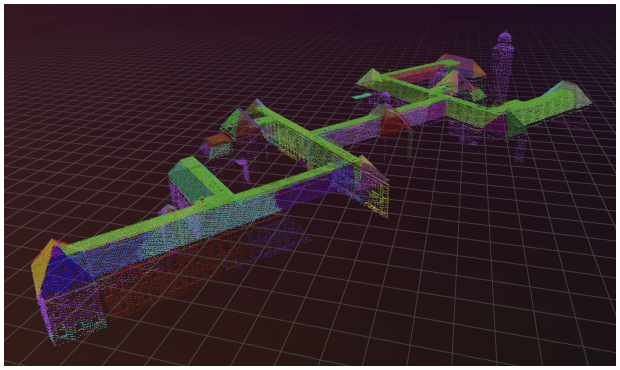


Reprocessing is enabled

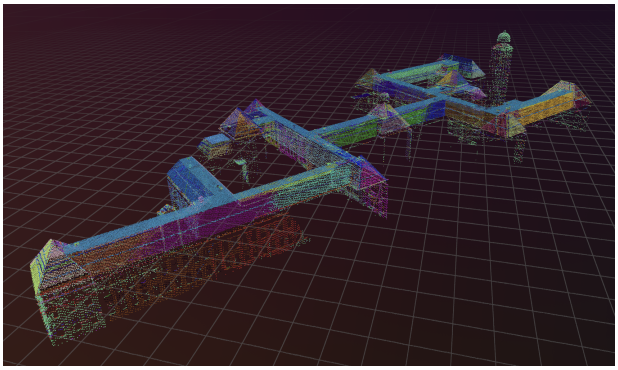


Reprocessing is disabled

Figure 19: Hough Transform when reprocessing of the planes is enabled and disabled.



Cleaning is enabled



Cleaning is disabled

Figure 20: Hough Transform when cleaning of the planes is enabled and disabled.

3 Results and comparison

3.1 Parameter Optimization

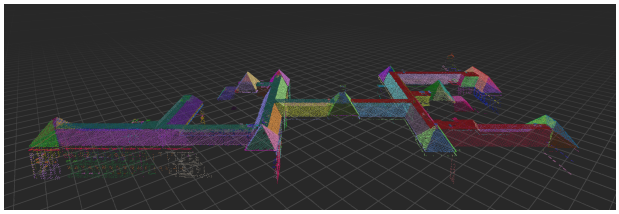
We fine-tuned our parameters with the following objectives, without any specific order or hierarchy:

- Enhancing the sharpness of detected surfaces.
- Minimizing the number of unclassified points.
- Reducing the number of falsely classified points.
- Maximizing time efficiency.

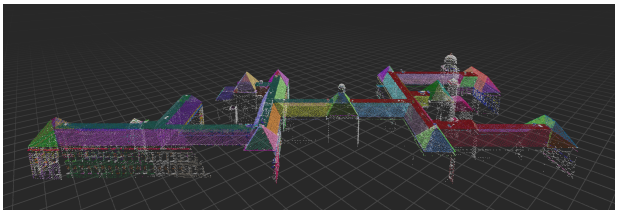
The parameters we ultimately selected are:

- RANSAC: $k = 1000$, $s_{min} = 200$, $\epsilon = 0.1$, $multiplier = 5$, $\delta = 3.5$, $n_{min} = 20$.
- Region Growing: $k = 25$, $max\_angle = 30$, $min\_seeds = 20000$, $min\_region\_size = 30$.
- Hough Transform: $Alpha = 50$, $Epsilon = 0.15$ $Theta = 10$, $Phi = 10$, $Rho = 0.1$, $ChunkSize = 25$, $Accelerationfactor = 10$, $Reprocessing = true$, $Cleaningdistance = 2$, $Cleaningneighbors = 80$

3.2 Visualizations and Visual Analysis for each Algorithm

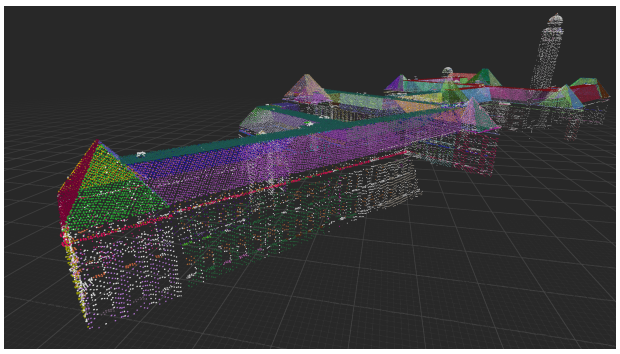


RANSAC result: classified points.

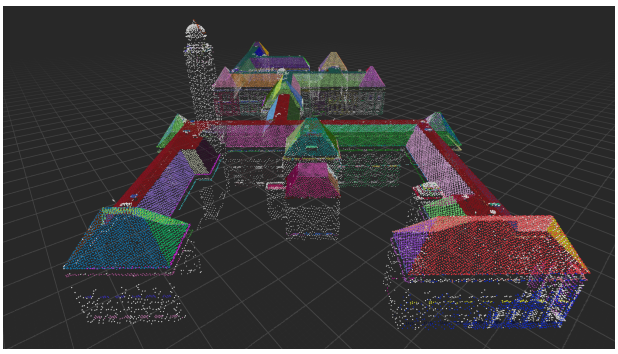


RANSAC result: all points.

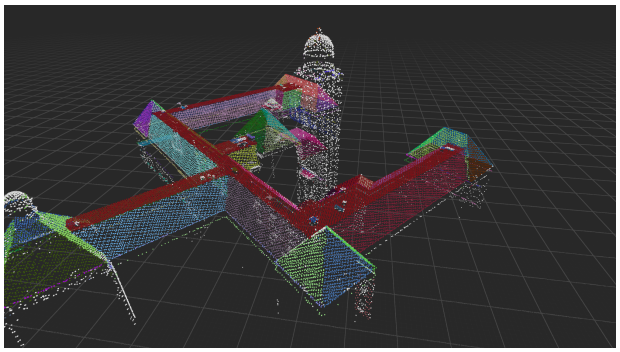
Figure 21: Classified and non-classified points from RANSAC outcome, with $k = 1000$, $s_{min} = 200$, $\epsilon = 0.1$, $multiplier = 5$, $\delta = 3.5$, $n_{min} = 20$.



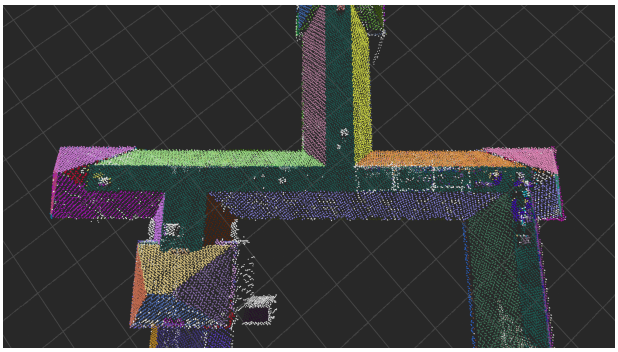
Some of the walls and even windows are extracted.



Some of the walls are being neglected.

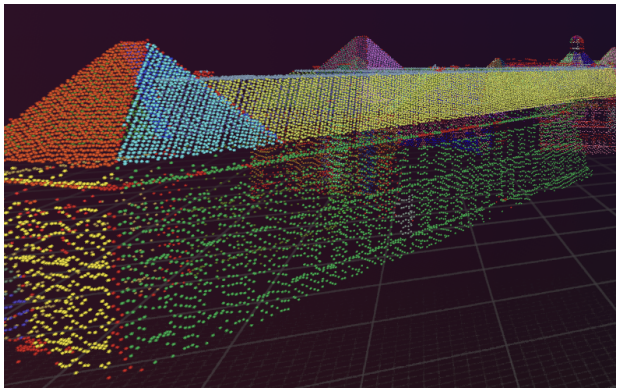


It fails to identify most points of the tower.

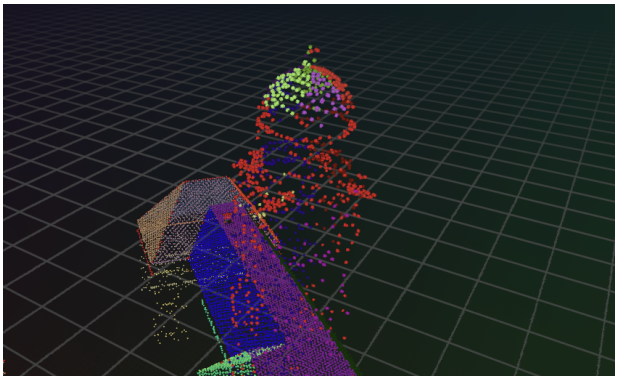


Some points are missed in a specific pattern due to post-process.

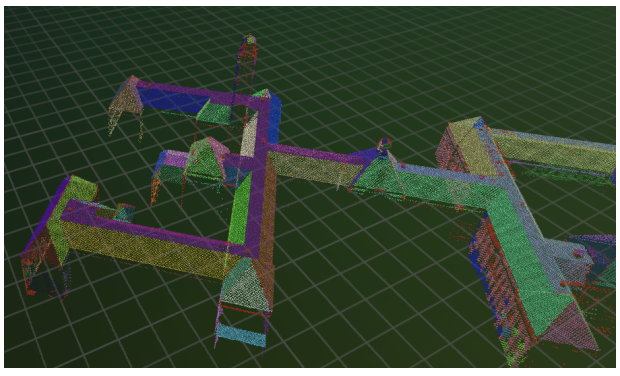
Figure 22: RANSAC outcome with $k = 1000$, $s_{min} = 200$, $\epsilon = 0.1$, $multiplier = 5$, $\delta = 3.5$, $n_{min} = 20$.



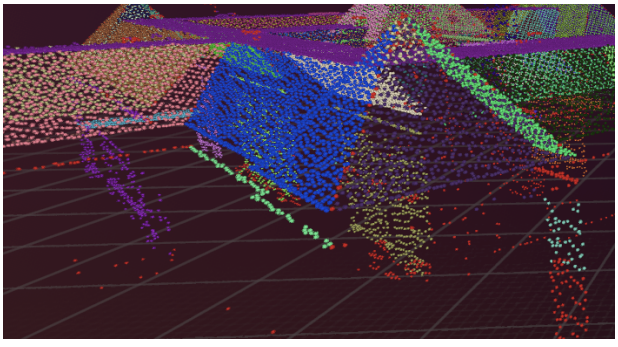
The walls and even windows are extracted.



Curved Surfaces are not clearly segmented

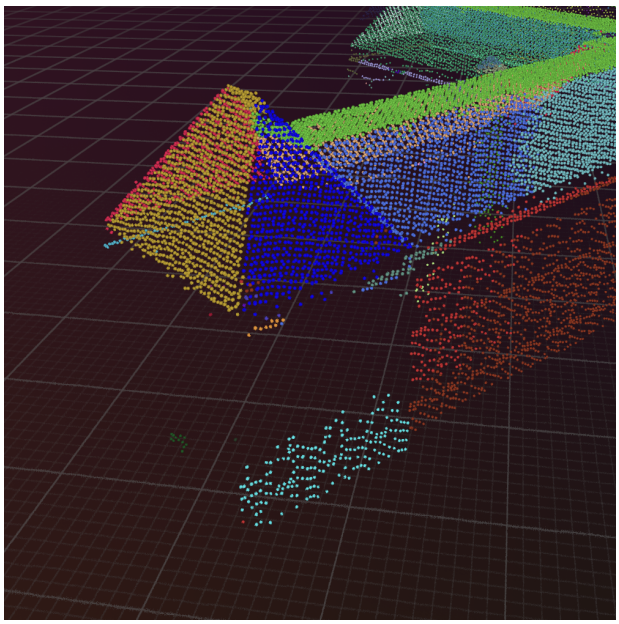


Bigger features are much clear and less noisy.

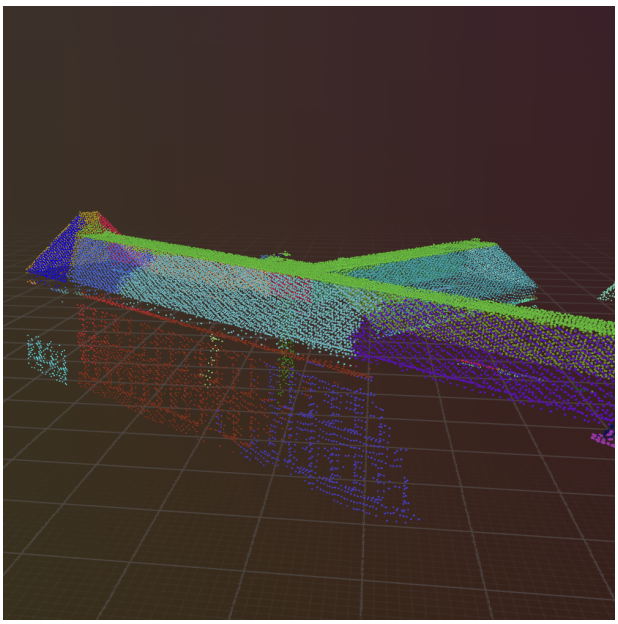


Small feature can be segmented with enough amount of seed points.

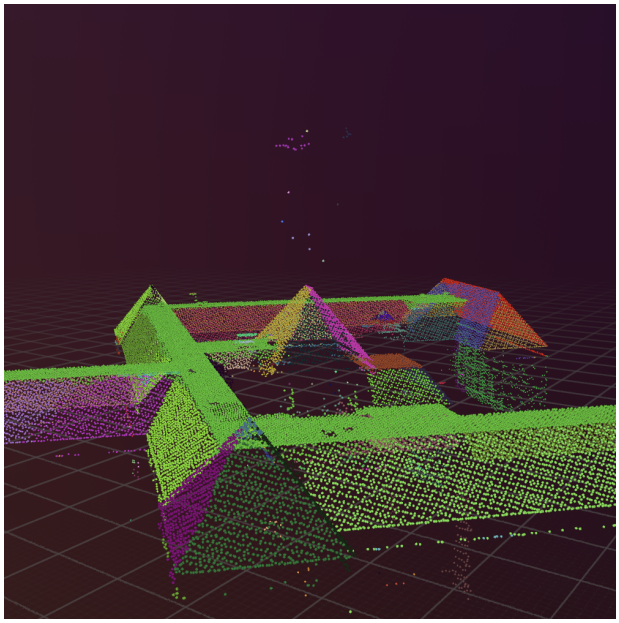
Figure 23: Region Growing outcome 165 planes. Above we use parameters: $k = 25, max\_angle = 30, min\_seeds = 20000, min\_region\_size = 10$.



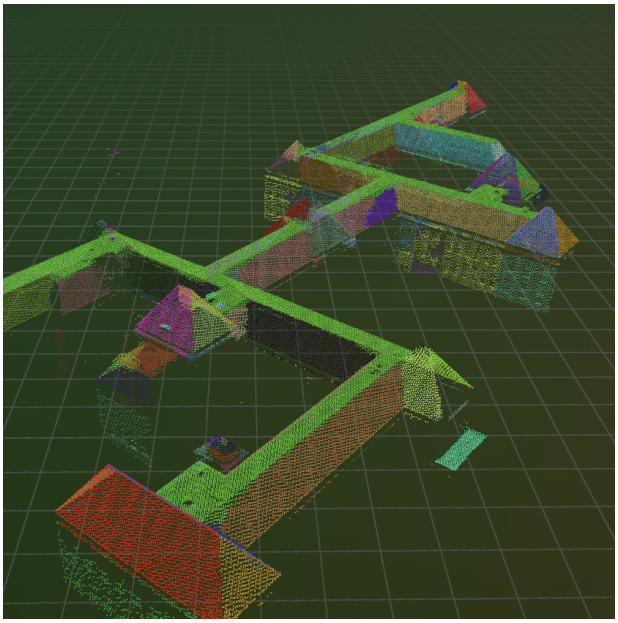
Some of the walls are missing



Some plains are still split, even after reprocessing



The tower was not captured effectively



Most planes were captured nicely, but performance could definitely be better

Figure 24: Hough Transform outcome: 150 planes. These images use the standard parameters

3.3 Comparison of Methods

3.3.1 Pros and cons of each algorithm

RANSAC
Due to RANSAC's nature of randomness, it's possible to have different result every time running the program (unless someone fix the random seed).

While RANSAC gives excellent results within a short time frame (around 3–5 minutes for the BK city dataset), it struggles to detect smaller or less-dense planes. This limitation arises from the multiple parameters that require fine-tuning, which can be challenging due to their interdependence within our pipeline structure. Additionally, a plane-detection-specific RANSAC cannot identify certain curved surfaces (e.g., the BK Tower). To extract curved surfaces, RANSAC requires the incorporation of multiple detection models tailored for such geometries.

Region Growing
Region growing achieves comprehensive plane detection with high local accuracy as it is a bottom-up segmentation approach in which normal computation is performed using PCA, followed by planarity-based seed selection and iterative region growth via comparison with neighbors. It takes around 10-15 minutes to process the full BK-City dataset. The method donest have noise in bigger planes and also tries to segment the curved surfaces with least number of unclassified points. If seed points are there for all the planes correctly and the parameters are tuned well enough, then this method will have all the planes segmented properly with least noise.
The bottleneck in the speed of the algorithm is outstandingly formed by KD-tree queries and normal computations. Performance varies by point cloud characteristics: planar regions process efficiently, whereas intricate geometries demand extra computation due to frequent normal comparisons. The method doesn't require prior knowledge of the number of planes, which increases its flexibility; independently-growing regions take more time in computation.

Hough Transform
Hough Transform has no identifiable pros. It is inherently slow due to the brute-force nature of the algorithm. Additionally the `Theta`, `Phi` and `Rho` parameters which have arguably the largest impact on both speed and quality need to be fine tuned for each point cloud, making it very fragile. Hough Transform also requires many modifications from the base algorithm before it becomes workable on larger datasets. The chunking step is especially vital, as without it the point cloud would resemble a layered cake. Further work could focus on automatically determining optimal values for `Theta`, `Phi` and `Rho`, although doing this would mean Hough Transform would become similar to Region Growing.

3.3.2 Comparison of algorithms (accuracy, speed, robustness)

	Time Complexity	Runtime (s)	Surfaces Found	Unclassified Points
RANSAC	$O(\|L_{\mathcal{I}}\| \times (kn + n \log n))$	235.36	224	16189
Region Growing	$O(L_{\mathcal{S}}\| \times \|R\| \times k \times (\log n + \|R\|)$	753.56	165	7874
Hough Transform	$O(\|P\| \times \|Theta\| \times \|Phi\| \times \|Rho\|)$	612.90	150	13433

4 Conclusion

The best shape detection algorithm is... yet to be determined.

In our opinion, without taking the overall run-time into account, the Region Growing algorithm gives the best result in terms of the *unclassified points*, *extracted surfaces*, *visually interpretability*. The overall run-time of the Region Growing algorithm is due to the (re)calculation of the normal's and the neighbours in the k-d tree due to the required for-loop, which is inherently part of the Region-Growing algorithm/paradigm.

We therefore also wrote a similar Region Growing approach: `regiongrowing_noloop.py`. Which initiates the k-d tree and all the neighbours for the seed points, without in-between updating and querying of the k-d tree for each candidate point.

On the other hand, RANSAC and Hough Transform operate on a similar logic when it comes to identifying planes, and they produce comparable results. However, Hough Transform requires tuning a larger number of parameters, and the algorithm itself demands more improvements to function effectively.