GRA Assessment Part Two - Final Report

Utkarsh Nattamai Subramanian Rajkumar
urajkumar3@gatech.edu

***Abstract*—**This paper is aimed to provide a report on stripe-based sidewalk extraction method prototype implementation.

1 PURPOSE

For pedestrians, especially wheelchair users, sidewalks are an essential piece of infrastructure. According to data from the 2010 census [1], over 3.6 million disabled people in the United States rely on wheelchairs for transportation every day. For daily trips to be safe and unhindered, wheelchair users depend on well-maintained sidewalks. Public transportation agencies are currently under pressure to maintain sidewalks more affordably and to adhere more closely to the Americans with Disabilities Act (ADA) because of the dependence on the infrastructure and the rising demand from these users. However, because of the time-consuming and expensive data collection process used in present practice, these timely review and maintenance tasks are typically lacking. Therefore an effective and trustworthy automated sidewalk assessment approach is urgently required for ADA compliance and to provide fair accommodations for each type of transportation. Before automatically accessing sidewalk conditions, there needs to be an automated way to first extract sidewalks. Given that LiDAR technology has become increasingly affordable, accessible, and reliable, road scenes can be captured using LiDAR. Then, common urban ground objects like sidewalks can be extracted using the 3D point clouds. The stripe-based sidewalk extraction approach described in [2] is one of the known methods for extracting sidewalks from the provided 3D point cloud data. This paper explores the implementation of this method and addresses challenges encountered during the implementation process.

2 POINT CLOUD DATA FOR SIDEWALK EXTRACTION

A sample point cloud data was provided to test the implementation of the stripe-based sidewalk extraction method. Figure 1 shows the sample point cloud data provided as part of the assessment. **Note:** the sample data is annotated to help readers distinguish the various urban ground objects (such as sidewalk, grass regions, and road pavement) in the figure.

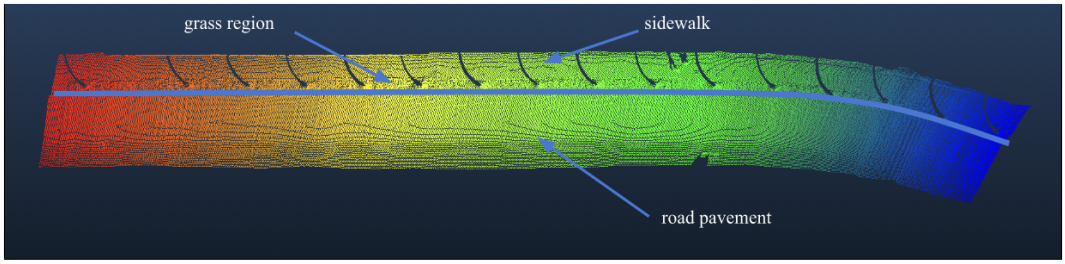

Figure 1—sample point cloud data provided to test the implementation of the stripe-based sidewalk extraction method

3 OVERVIEW OF STRIPE-BASED SIDEWALK EXTRACTION METHOD

The stripe-based sidewalk extraction method is one of the known approaches for extracting sidewalk regions from the given point cloud data. One of the advantages of this method is that elevation and lateral offset filtering process can be removed without impacting the overall performance of sidewalk extraction. The high level steps of the stripe-based sidewalk extraction method, as described in [2], are as follows:

1. **dividing point cloud into stripes:** the given point cloud data is divided into stripes which will be used in the following stages. Each stripe covers 1 meter in the distance along the traveling direction.
2. **converting point cloud stripe into octree:** points associated with man-made and natural terrain class and bounded by a stripe are converted into an octree structure
3. **stripe splitting process:** recursively splits the point cloud until each node in the octree structure only contains points that meet the coplanar criterion.
4. **stripe merging process:** attempts to merge neighboring nodes points if the combined node points still meet the coplanar criterion. This merging is done until two neighboring nodes can no longer be combined such that the combined node points still meet the coplanar criterion.
5. **longitudinal cluster process:** after all the stripes have been processed, k-NN method is utilized to longitudinally cluster all the segmentation results together.

4 PROTOTYPE IMPLEMENTATION OF THE STRIPE-BASED SIDEWALK EXTRACTION METHOD

4.1 Prototype Implementation Goals

The main goal for part two of the assessment is to have a prototype of the stripe-based sidewalk extraction method that can perform steps one through four (dividing point cloud into stripes, converting point cloud stripe into octree, stripe splitting process, and stripe merging process) as described in the previous section. **Note:** to limit the scope of the assessment, step five of the stripe-based sidewalk extraction method is not included as part of the prototype implementation.

4.2 Prototype Implementation Details

4.2.1 *Selecting Portion of Point Cloud Data for Testing*

To simplify the testing efforts and reduce complications that can potentially arise from using all of the given point cloud data, only a portion of the given sample point cloud data is used for testing the implementation of the stripe-based sidewalk extraction method. Specifically, the selected portion of the point cloud data does not include the curve at the right end. The portion of the given sample point cloud data used for testing is shown in Figure 2.

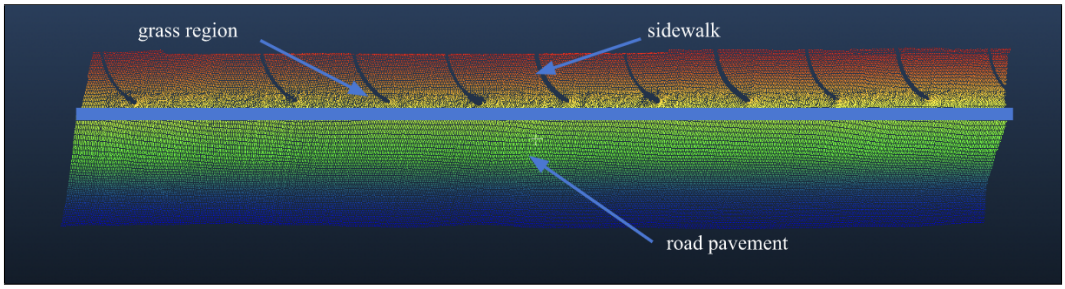

Figure 2—the portion of the given sample point cloud data used for testing

4.2.2 *Filtering Points Not Part of Man-Made Class and Dividing Point Cloud Data Into Stripes*

After the portion of the sidewalk was selected, all points that are not associated with man-made and natural terrain class were removed. This ensured that the selected portion of the sidewalk no longer contained tree trunks, a walking dog, and vegetations. After this step, the portion of the given sample point cloud data

was divided into stripes where each stripe covers 1 meter in the distance along the traveling direction. Figure 3 shows the portion of the given sample point cloud data divided into stripes of equal width.

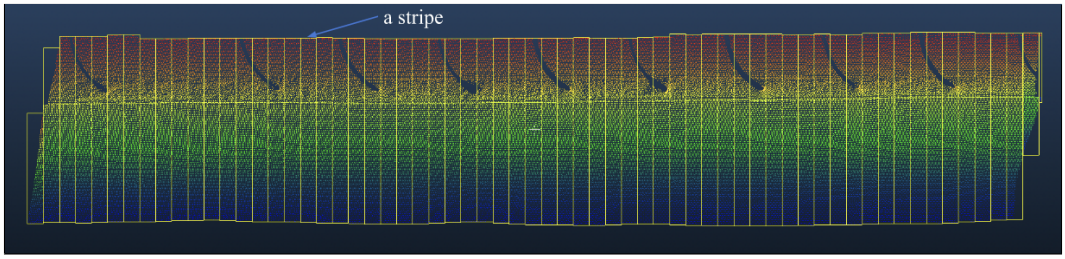

Figure 3—dividing the portion of the given sample point cloud data used for testing into stripes

After the portion of the given sample point cloud data is divided into stripes, stripe splitting and stripe merging processes are performed for each stripe. The below subsection will describe how the stripe splitting was implemented.

4.2.3 *Stripe Splitting Process*

Before stripe splitting process, all the points within a given stripe are converted into an octree node (root node). Then the root node is passed in as input to the stripe splitting function (code review for function provided in appendix). The stripe splitting function recursively splits the point cloud until each node in the octree structure only contains points that meet the coplanar criterion. To be specific, if an octree node does not satisfy the coplanar criterion, then this node will be split into eight children nodes. Figures 4a - 4c show the stripe splitting process using a 2D example. Figure 4a shows space contains all the point clouds within the stripe as the root node. Since the coplanar criterion is not satisfied, the space is split into eight sub-space. **Note:** only four spaces are shown in Figure 4b. Then point set in node 0 will be further split into eight sub-space since the point set does not satisfy the coplanar criterion. **Note:** only four spaces are shown in Figure 4c. In Figure 4c, since the points set in all the nodes pass the coplanar criterion, no further split is required. Additionally, the coplanar criterion is satisfied if the third eigenvalue (obtained from the points within a node) is less than or equal to Δ. In the implementation, the value for Δ was chosen to be 0.002. Section five will address the reasoning used to determine the value for Δ. The below subsection will cover how the third eigenvalue was found for a given set of points.

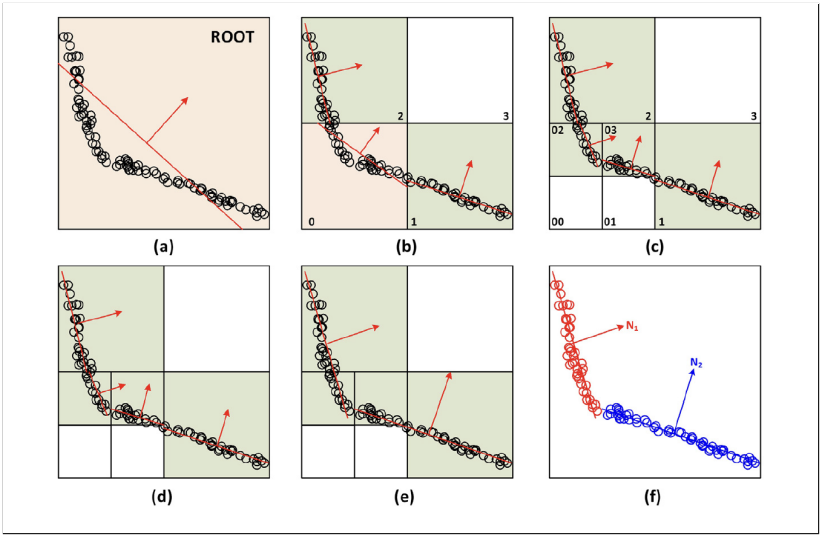

Figure 4—stripe splitting and merging process using a 2D example

4.2.4 *Determining the Third Eigenvalue for Given Set of Points*

The getThirdEigenvalue() function (code review for function provided in appendix) was used to retrieve the third eigenvalue for a set of points of interest. This function first uses Numpy's cov() function to get the covariance matrix for the set of points. Then, the covariance matrix is used to find the three eigenvalues/eigenvectors where each eigenvalue/eigenvector corresponds to one of the X,Y, and Z directions using Numpy's eig() linear algebra function. From here, the third eigenvalue is returned. In this context, a smaller magnitude for third eigenvalue suggests that points have low variance in the Z axis while a larger magnitude for third eigenvalue suggests that points have high variance in the Z axis.

4.2.5 *Stripe Merging Process*

Before the stripe merging process, all the leaf nodes that are in the octree structure are first stored into a list. Then, the stripe merging algorithm is applied to combine neighboring nodes in the list if the points in the combined node still satisfy the coplanar criterion. The merging process is exhaustively conducted until no neighboring nodes in the list can be merging without violating the coplanar criterion. Figures 4d-4f show the stripe merging process using a 2D example. In Figure 4d, some of the neighboring nodes share a similar normal direction.

This indicates that the points in these neighboring nodes can be merged into the same cluster given that the combined points pass the coplanar criterion. The neighboring nodes are merged into single nodes as shown in Figure 4e. Figure 4f shows the results of the clustering. As shown in Figure 4e, two nodes (two clusters) are identified in this point cloud. The next subsection mentions the Y-axis filtering performed before the points in the remaining nodes are stored in CSV files.

4.2.6 *Y-Axis Filtering Process*

Although this stage is not part of the stripe-based sidewalk extraction method, it was included after noticing a recurring pattern during the testing phase. Without this Y-axis filtering process, it was noticed that a lot of the road pavement points were also included as part of the sidewalk. The points associated with road pavements should not be part of the implementation output. Figure 5 shows this problem visually. As seen in this figure, points that are part of the road pavement are also included as part of sidewalks.

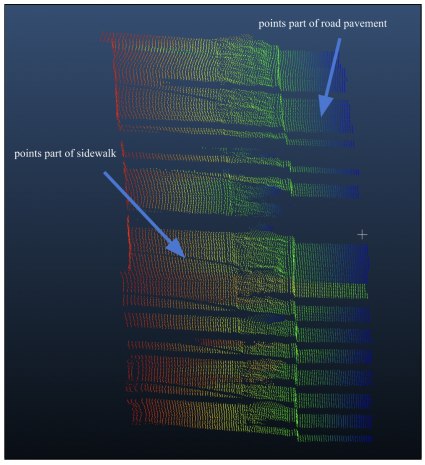

Figure 5—implementation results without Y-axis filtering process

Therefore, the Y-axis filtering process was introduced to address this issue. The threshold value used for Y-axis filtering is 3740621.59. Specifically, points whose y value is less than 3740621.59 are filtered out. As seen, this threshold value is currently manually defined. Additionally, section five addresses the reasoning used to determine this threshold value. Furthermore, the remaining points after this filtering process are then stored in CSV files.

5 DETERMINING THRESHOLDS/HYPERPARAMETERS

As seen in the previous section, two threshold values were used during the implementation of the stripe-based sidewalk extraction method. The first threshold value was associated with coplanar criterion while the second threshold value was associated with Y-axis filtering process. The below subsections will describe the details on how these threshold values were determined.

5.1 Determining Threshold Value for Coplanar Criterion

As mentioned in section four, points within a node are considered to satisfy the coplanar criterion if the third eigenvalue is less than or equal to Δ. The value for Δ was mentioned to be 0.002. This subsection provides the reasoning used to determine the value for Δ.

5.1.1 *First Iteration - Threshold Value of 0.0508*

In [2], the Δ value of 2 inches was experimentally selected so that the splitting and merging process can be less affected by some of the typical surface disturbances captured by the mobile LiDAR. The conversion of 2 inches to meters would be 0.0508 meters. To adhere with the paper, 0.0508 was first chosen as the threshold value for coplanar criterion. However, it was noticed that most of the stripes had third eigenvalue below 0.0508. In fact, the third eigenvalue for most of these stripes were well below 0.0508. Figure 6 shows the results derived by using coplanar criterion threshold value of 0.0508 without applying Y-axis filtering. As seen in the figure, most of the stripes are not splitted and merged. This suggests that the coplanar criterion threshold value was much larger than expected.

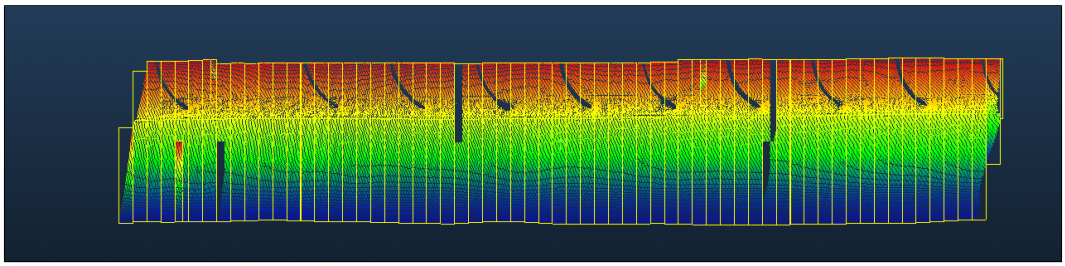

Figure 6—results after using 0.0508 for coplanar criterion threshold value

5.1.2 *Second Iteration - Threshold Value of 0.00508*

Next, the value of 0.00508 was chosen for the coplanar criterion value. This value was chosen due to the realization that the third eigenvalue for most of the stripes were most likely way below 0.0508. Even though this value was ten times lower than the first proposed value, this value also encountered the same problem as the previous value of 0.0508.

5.1.3 *Third Iteration - Threshold Value of 0.001*

In this iteration, the value of 0.001 was chosen for the coplanar criterion value. Figure 7 shows the results derived by using coplanar criterion threshold value of 0.001 without applying Y-axis filtering. As seen in the figure, a lot of the points associated with the sidewalk region are missing. This suggests that 0.001 is not the right threshold value for the coplanar criterion.

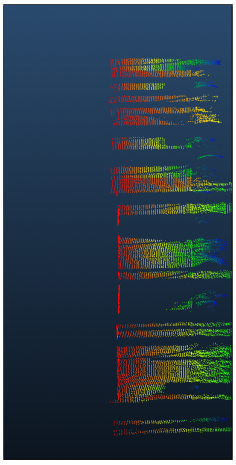

Figure 7—results after using 0.001 for coplanar criterion threshold value

5.1.4 *Fourth Iteration - Threshold Value of 0.002*

Next, the value of 0.002 was chosen for the coplanar criterion value. Since choosing lower values as the threshold seemed to loose points associated with the sidewalk region, a value higher than 0.001 was experimented as a potential coplanar threshold value candidate. By choosing 0.002 as the threshold value, it seemed like most of the points associated with the sidewalk region were not lost. Additionally, the third eigenvalue for nearly all the stripes were only slightly larger than 0.002. This suggested that the stripes would be split since they would

not satisfy the criterion. Figure 8 shows the results derived by using coplanar criterion threshold value of 0.002 without applying Y-axis filtering.

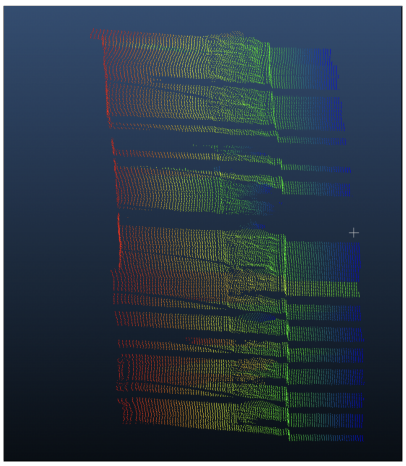

Figure 8—results after using 0.002 for coplanar criterion threshold value

5.1.5 *Other Values as Coplanar Criterion Threshold*

Values higher than 0.002 were also experimented as potential candidates for coplanar criterion value. However, as previous mentioned, it was noticed that a lot of stripes had third eigenvalues slightly higher than 0.002. Therefore choosing values of 0.003 for instance would have the same effects as choosing 0.00508. Therefore, through these manual efforts, 0.002 was determined to be the best value for the coplanar criterion.

5.2 Determining Threshold Value for Y-Axis Filtering

The value for Y-axis threshold was determined by seeing the place where grass region separate from the sidewalk region. This was done using CloudCompare software. Figure 9 shows the point of separation. The "Yg" value in the figure indicates the value of the y-axis where the separation roughly takes place. **Note:** it was realized that specific values for Y-axis filtering should not be used since the place of separation between grass regions and sidewalk regions can vary in different regions of the given point cloud data. However due to time constraints, the current implementation used the specific value.

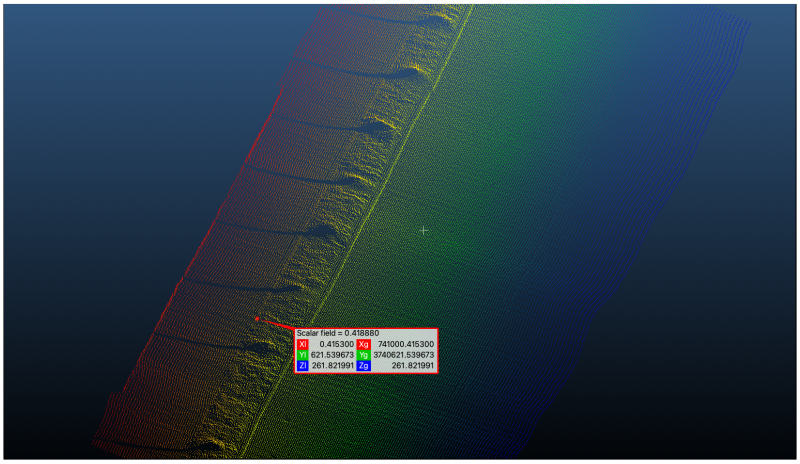

Figure 9—point of separation between the grass region and sidewalk region

6 BEST IMPLEMENTATION RESULT

The best result from the implementation is shown in Figure 10. This result was obtained by choosing 0.002 as the coplanar criterion threshold value and 3740621.59 as the Y-axis filtering threshold value.

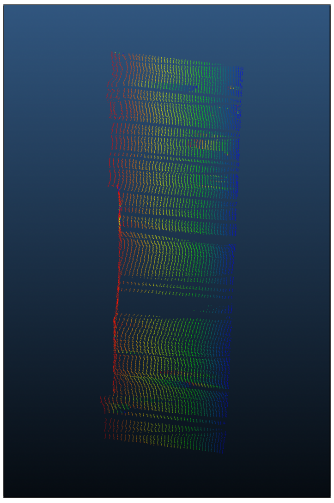

Figure 10—best implementation result obtained by choosing 0.002 as coplanar criterion threshold value and 3740621.59 as Y-axis filtering threshold value

7 CURRENT VALIDATION PROCEDURE AND FUTURE POTENTIAL VALIDATION APPROACH

7.1 Current Validation Procedure

Currently the results are tested via a visual approach. Since we know what the ideal result should look like, we can compare a generated result with this ideal result to determine whether the generated result correctly extracts the sidewalk region. The visual approach was used for validation throughout the implementation process due to time constraints. But it is important to realize that there are limitations to this method. The most important limitation is that minor difference in results will be hard to judge visually. If a couple of points are missing, it will be hard to realize there are missing points via a visual approach. Additionally, there is no proper way to label a result. For instance, different users might have different opinions on what "good" result means in this situation. Therefore the labeling of the results will vary from user to user. To perform better validation of the results, a numerical approach can be explored.

7.2 Future Potential Validation Approach

7.2.1 *Manually Labeling Ground Truth*

This approach suggests that we can manually label points that are associated with sidewalk regions in the provided point cloud data. The assumption with this approach is that the given point cloud data only covers a small region as manually labeling individual points in a point cloud that covers a large region would be impractical. Once the points associated with the sidewalk have been labeled, we can then see how many of these labeled points are in the results generated by the implementation. A high match would suggest that the implementation accurately extracts major parts of the sidewalk region. One downside with this approach is that it will only indicate if the result has less points for the sidewalk region. However, it will not see if the implementation result covers more than the sidewalk region (such as including points associated with road pavements).

7.2.2 *Other Approaches*

There are a couple of validation approaches mentioned in the Performance of the Sidewalk Extraction section in [2]. These validation approaches can be taken

in order to see if the implementation results accurately extract sidewalk region.

8 LIMITATIONS OF THE PROTOTYPE

One of the main limitations of the current prototype implementation is that thresholds are manually defined. This is a drawback since the goal of the stripe-based sidewalk extraction method is to automatically extract sidewalks from a given point cloud data. Another limitation with this current approach is that a generalized value for the coplanar criterion threshold (Δ = 0.002) does not work well for different regions of the given point cloud data. The particular chosen threshold value only worked well for the portion of the given point cloud selected for testing. However, when this threshold value was used for the whole given point cloud data, the results were not accurate in many regions. The third limitation with the prototype is that without the Y-axis filtering process, the result includes points that are associated with road pavements. This should obviously be avoided since points that are associated with road pavements should not be in the result which is focused on solely extracting sidewalk region points. The Y-axis filtering process is also not part of the stripe-based sidewalk extraction method. This process was only included in the prototype as a quick way of addressing the third limitation. Additionally, due to the scope of the assessment, the fifth step of the stripe-based sidewalk extraction method is not implemented. Without the implementation of this step, the prototype does not provide an end-to-end solution for automating extracting sidewalk regions given point cloud data. Finally, another limitation with this prototype is that a lot of other processes of the stripe-based sidewalk extraction method are manually performed. For instance, processes such as dividing the point cloud data into stripes are performed manually in the prototype implementation.

9 POTENTIAL IMPROVEMENTS AND REAL-WORLD IMPLEMENTATION OF THE PROTOTYPE

Before this prototype can be converted into a real-world algorithm that can be used to extract sidewalk regions, there are a couple of things that must be addressed. First, there needs to be an automated way of determining the two threshold values mentioned in this paper. Currently, the prototype has determined these two threshold values manually. Since the goal is to automatically extract sidewalk regions from a given point cloud data, having manual processes like determining thresholds hinders from achieving the goal. In addition, the

current prototype manually divides the given point cloud data into stripes of equal width. However, it would be beneficial to make this process automatic in the real-world implementation to save both time and effort. Additionally, the implementation only covers steps one through four in the stripe-based sidewalk extraction method as mentioned in the previous section. In order to make sure the real-world implementation is an end-to-end solution, all parts of the stripe-based sidewalk extraction method must be implemented. This includes everything in the prototype implementation along with the automatic classification of sidewalk segments and longitudinally clustering the segmentation results from each stripe. As a bonus, GPS data can also be used for 1) getting better results and 2) for validation purposes. Most importantly, the real-world implementation must ensure results are accurate for all regions of the given point cloud data not just the selected portion that is used for testing. As mentioned in section four, a portion of the given point cloud was selected for testing in order to simplify the testing efforts. But to translate the prototype into a real-world implementation, it must be able to extract sidewalk accurately given any region of point cloud data. Furthermore, the Y-axis filtering process should be eliminated since it is not part of the proposed stripe-based sidewalk extraction method in [2]. If it is absolutely necessary to include this filtering process, the threshold for filtering must be automatically defined instead of manual efforts. In case it is difficult to automatically define the threshold for Y-axis filtering, another method can be explored. Since the goal of Y-axis filtering is to remove points that are not part of the sidewalk region, any other method that achieves this purpose can also be used as an alternate to the Y-axis filtering process. One such method would deal with retroreflectivity values. From the given point cloud data, it was noticed that points associated with road pavements, grass regions, and sidewalks regions had different retroreflectivity values. Figure 11 shows this pattern.

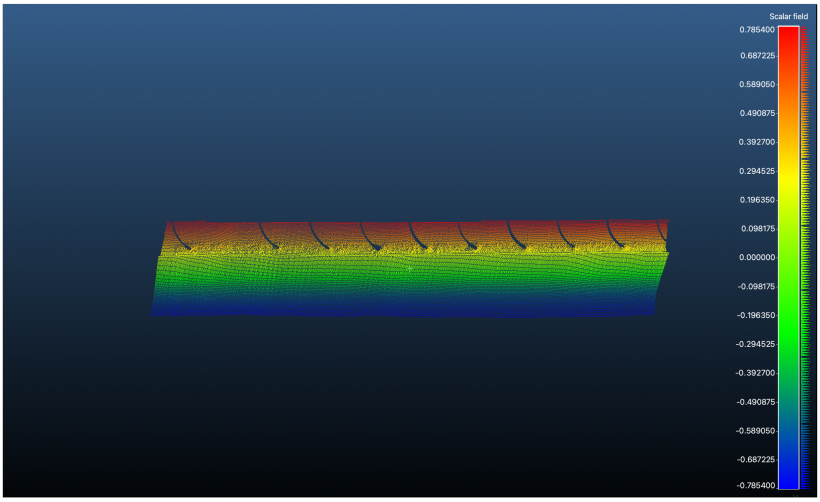

Figure 11—different retroreflectivity values for sidewalks, road pavements, and grass regions

This idea can be used to effectively remove points that are not associated with sidewalks. Overall, these are some of the general steps that can be taken to convert the prototype into a real-world implementation. Most importantly, while developing a real-world implementation it is essential to focus heavily on the validation stage. It is highly recommended to determine robust validation approaches so that results can be validated accurately and implementation can be improved before used in real scenarios.

10 REFERENCES

[1] Ai, C., Tsai, Y. (2016). Automated sidewalk assessment method for Americans with disabilities act compliance using three-dimensional mobile LiDAR. Transportation Research Record, 2542(1), 25-32.

[2] Hou, Q., Ai, C. (2020). A network-level sidewalk inventory method using mobile LiDAR and deep learning. Transportation research part C: emerging technologies, 119, 102772.

11 APPENDICES

The appendix is included to provide a brief code review of the prototype.

11.1 OctreeNode Class Code Review

```
class OctreeNode:
    def __init__(self, value, children, points, indices, depth):
        self.value = value
        if (indices == None):
            self.indices = None
        else:
            self.indices = indices
        if (points == None):
            self.points = None
        else:
            self.points = points
        if (children == None):
            self.firstChild = None
            self.secondChild = None
            self.thirdChild = None
            self.fourthChild = None
            self.fifthChild = None
            self.sixthChild = None
            self.seventhChild = None
            self.eigthChild = None
        else:
            self.firstChild = children[0]
            self.secondChild = children[1]
            self.thirdChild = children[2]
            self.fourthChild = children[3]
            self.fifthChild = children[4]
            self.sixthChild = children[5]
            self.seventhChild = children[6]
            self.eigthChild = children[7]
        self.depth = depth
```

Figure 12—OctreeNode class

- The code review below references Figure 12.
- This is the OctreeNode class. In the constructor, it requires the value of the node, children (children of the current node), points (number of points in the node), indices (indices of the points that are in the node), and depth of the node.
- These values are then used to initialize an OctreeNode.
- As seen in Figure 12, if a value of a parameter is None, the corresponding instance variable values is also set to None

11.2 getThirdEigenvalue() Function Code Review

- The code review below references Figure 13.
- This is the getThirdEigenvalue() function. It takes in a set of points (data) as input.
- Then the X,Y, and Z values of the set of points are stored as separate arrays (samplePointsX, samplePointsY, and samplePointsZ)
- Numpy's vstack() function is then used to stack the retrieved arrays in sequence vertically (row wise) which is stored in variable 'x'.
- The covariance matrix is then found using Numpy's cov() function.

```
def getThirdEigenvalue(data):
        samplePointsX = data[:,0]
        samplePointsY = data[:,1]
        samplePointsZ = data[:,2]
        x = np.vstack([samplePointsX,samplePointsY,samplePointsZ])
        cov = np.cov(x)
        eigen_vals, eigen_vecs = np.linalg.eig(cov)
        return eigen_vals[2]
```

Figure 13—getThirdEigenvalue() function

- Using the covariane matrix, the three eigenvectors/eigenvalues are determined. From this, the third eigenvalue is returned as output.

11.3 performExtraction() Function Code Review

```
def performExtraction(coplanarThresholdValue, elevationFilter,fileNameIndex, fileName):
    data = pd.read_csv(fileName)
    pointCoordinates = data[['X','Y','Z']]
    pointCoordinates = np.asarray(pointCoordinates)
    #print(f"point coordinates length: {len(pointCoordinates)}")
    octree = open3d.geometry.Octree(max_depth=0)
    pcd = open3d.geometry.PointCloud()
    pcd.points = open3d.utility.Vector3dVector(pointCoordinates)
    octree.convert_from_point_cloud(pcd)
    depth = 0
    rootNode = OctreeNode(octree.root_node, None, len(octree.root_node.indices), octree.root_node.indices,depth)
    stripeSplitting(rootNode, data,rootNode.indices,depth,coplanarThresholdValue)
    leafList = []
    getAllLeafNodes(rootNode,leafList)
    counter = 0
    for i in leafList:
        counter += len(i)
    #print(f"counter: {counter}")
    returnList = stripeMerging(leafList,data,coplanarThresholdValue)
    getCsv(returnList,data,elevationFilter,fileNameIndex,fileName)
```

Figure 14—performExtraction() function

- The code review below references Figure 14.
- This is the performExtraction() function. It takes in as input coplanar threshold value, elevation filter for Y-axis filtering, fileNameIndex, and fileName.
- Essentially for a stripe, all the points within this stripe are converted into an octree node (root node).
- Then, all root node is passed into the stripe splitting function to recursively split the point cloud until each node in the octree structure only contains points that meet the coplanar criterion.
- After the previous step, the leaf nodes in the octree are retrieved and the stripe merging function is called to merge neighboring nodes points if the combined

node points still meet the coplanar criterion. This merging is done until two neighboring nodes can no longer be combined such that the combined node points still meet the coplanar criterion.

- After the last step, the Y-axis filtering is performed by using the elevationFilter as the threshold value for Y-axis filtering. All the remaining points after this filtering process are then stored in a CSV file.

11.4 stripeSplitting() Function Code Review

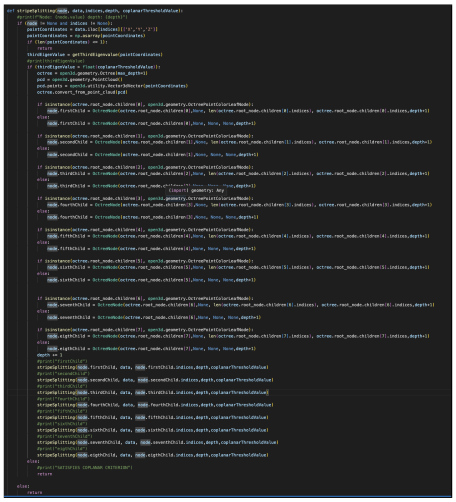

Figure 15—stripeSplitting() function

- The code review below references Figure 15. **Note:** the code segment can also be viewed in stripeExtraction.py file.
- This is the stripeSplitting() function. It takes in the current octree node, the data provided as part of assessment, the point indices for points in the current node, the depth of the current node, and the coplanar criterion threshold value as input.
- Then, the third eigenvalue for the points that are part of this node are determined. If the third eigenvalue is greater than the coplanarThresholdValue, then the node is split into eight children (this is the splitting process).
- If a node is split, then the stripeSplitting() function is called on each of the child nodes in a recursive manner.

11.5 getAllLeafNodes() Function Code Review

```
def getAllLeafNodes(node, leafList):
    if (node != None):
        if isinstance(node.value, open3d.geometry.OctreePointColorLeafNode):
            printStatement = "  " * (node.depth) + "node: {} depth: {}".format(node.value,node.depth)
            if (node.firstChild == None and node.secondChild == None and node.thirdChild == None and node.fourthChild == None and node.fifthChild == None
            and node.sixthChild == None and node.seventhChild == None and node.eigthChild == None):
                leafList.append(node.indices)
            getAllLeafNodes(node.firstChild,leafList)
            getAllLeafNodes(node.secondChild,leafList)
            getAllLeafNodes(node.thirdChild,leafList)
            getAllLeafNodes(node.fourthChild,leafList)
            getAllLeafNodes(node.fifthChild,leafList)
            getAllLeafNodes(node.sixthChild,leafList)
            getAllLeafNodes(node.seventhChild,leafList)
            getAllLeafNodes(node.eigthChild,leafList)
        else:
            return
    else:
        return
```

Figure 16—getAllLeafNodes() function

- The code review below references Figure 16.
- This is the getAllLeafNodes() function. It essentially retrieves the leaf nodes that are in the octree structure in a recursive manner.
- These leaf nodes will then be used as input to the stripe merging function to perform stripe merging process.

11.6 stripeMerging() Function Code Review

```
def stripeMerging(leafNodes,data,coplanarThresholdValue):
    clusterList = []
    combined = True
    passNumber = 1
    while (combined):
        combined = False
        i = 0
        j = 1
        while (j < len(leafNodes)):
            if (leafNodes[i] != None and leafNodes[j] != None):
                combinedIndices = leafNodes[i] + leafNodes[j]
                specificData = data.iloc[combinedIndices]
                pointCoordinates = specificData[['X','Y','Z']]
                pointCoordinates = np.asarray(pointCoordinates)
                thirdEigenValue = getThirdEigenvalue(pointCoordinates)
                if (thirdEigenValue <= (float(coplanarThresholdValue))):
                    leafNodes[i] = None
                    leafNodes[j] = combinedIndices
                    combined = True
            i += 1
            j += 1
        passNumber += 1
        clusterList = []
        for i in leafNodes:
            if (i != None):
                clusterList.append(i)
        leafNodes = clusterList

    clusterList = []
    for i in leafNodes:
        if (i != None):
            clusterList.append(i)
    return clusterList
```

Figure 17—stripeMerging() function

- The code review below references Figure 17.
- This is the stripeMerging() function. It takes in input the leafNodes (list of leaf nodes from octree), the data provided as part of assessment, and the coplanar criterion threshold value.
- Essentially, the function attempts to merge neighboring nodes points together

if the combined node points still meet the coplanar criterion.

- This merging is done until two neighboring nodes can no longer be combined such that the combined node points still meet the coplanar criterion.
- The remaining nodes after the stripe merging process are then returned as output via the clusterList.

11.7 getCSV() Function Code Review

```
def getCsv(l,data, elevationFilter,filenameIndex,fileName):
    i = 0
    for j in l:
        fileNameString = "results/" + fileName[:6] + str(filenameIndex) + "child" + str(i) + ".csv"
        print(fileNameString)
        dataValue = data.iloc[j]
        dataValue = dataValue[dataValue['Y'] >= float(elevationFilter)]
        if (len(dataValue) > 0):
            dataValue.to_csv(fileNameString,index=False)
        i += 1
```

Figure 18—getCSV() function

- Figure 18 shows the getCSV() function. It has two roles.
- The first role is to perform Y-axis filtering by removing points that have y values less than the elevationFilter threshold value.
- The second role is to store the remaining points after filtering process into a CSV file.

11.8 main() Function Code Review

```
if __name__ == "__main__":
    coplanarThresholdValue = input("Enter coplanar threshold:")
    elevationFilteringValue = input("Enter elevation filter:")
    #stripesFileNames = ["stripe1.csv","stripe2.csv"]
    stripesFileNames = []


    for i in range(0,64):
        fileName = "stripe" + str(i) + ".csv"
        stripesFileNames.append(fileName)


    for i in range(len(stripesFileNames)):
        performExtraction(coplanarThresholdValue,elevationFilteringValue,i, stripesFileNames[i])
```

Figure 19—main() function

- The code review below references Figure 19.
- Here, the coplanar criterion threshold value (0.002) and elevation filtering value (3740621.59) used for Y-axis filtering are asked as input from the user.
- Then, the performExtraction() method is called for each stripe.