FINAL PROJECT REPORT

02807 Computational Tools for Big Data

Department of Applied Mathematics and Computer Science

Danmarks Tekniske Universitet

Lyngby

December 2016

Contents

1 Introduction

As introduced in the midterm assignment, OpenDroneMap [3] is an open source project that aims to provide a toolkit for processing of aerial images taken from civilian UAVs (Unmanned Aerial Vehicle) such as quadcopters, RC planes, kites and similar devices. The project at this current point in time is able to create point clouds [4] obtained from 2D images taken from UAVs. Using correspondences from characteristic points in overlapping images it is possible to create a 3D representation of an aerial panorama as can be seen in figure 5

Our challenge is based on processing these point clouds considering that the size can be very large and any processing involving millions of points is a challenging task. In our case we aim to develop one of the features that is in the roadmap of the Opendronemap project related to point classification. In order to achieve that we have implemented with PySpark the Progressive Morphological Filter (PMF) [5] to classify points as ground or non-ground. This feature will allow new functionality to the current state of the project since point classification is useful for analysing maps in different scenarios such as hydrological applications, construction applications, etc.

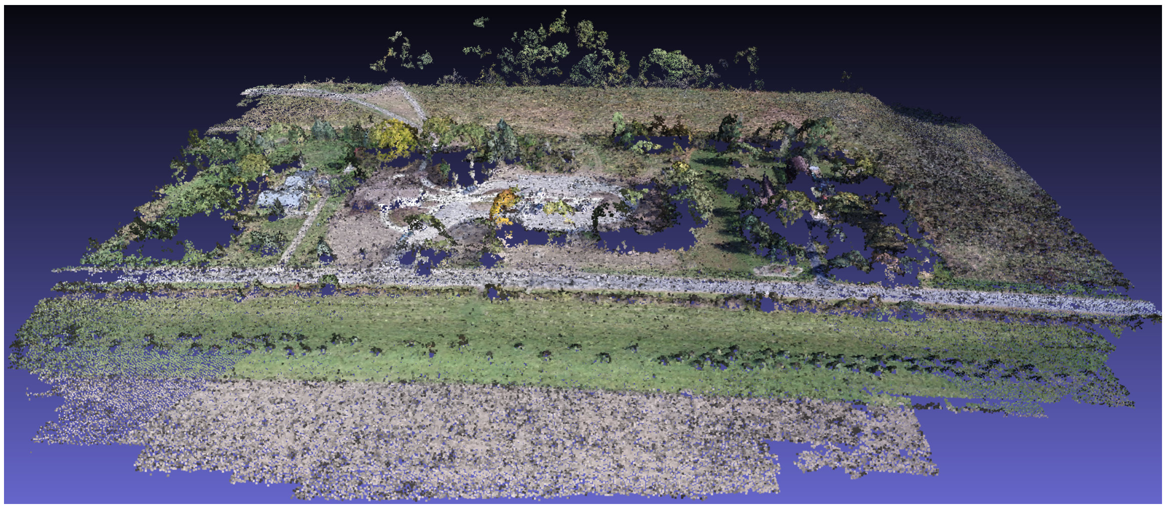


Figure 1: Textured point cloud

2 Spark implementation

The complete implementation can be found in Appendix 7.1.

2.1 Cartesian approach

As a direct translation of the naive implementation to Spark, we decided to use cartesian product - generating all possible pairs of elements from RDD_1 and RDD_2. Once all the combinations are obtained, using mapping and grouping functions we grouped all the pairs based on the condition that their distance lays within the size of the window/2. This approach resembles the idea of iterating all the points testing the distance between them in order to determine is they are neighbours. Additionally since the algorithm requires doing that for multiple windows we included second cartesian product (all possible pairs of points paired with all the windows). One of the benefits of this approach was the ability of performing the action of grouping neighbours in a set of simple map and reduce operations. Although this approach worked for small problem (43000 points) the complexity was $O(N^2 * M)$, where N is the number of points and M number of windows. We found problems when collecting results when the amount of points increased due to the need of building a NxN cartesian matrix in order to have all the pairs of points. Consequently, this approach turned out to be highly inefficient when the number of points increased.

```
pc_pairs = points.cartesian(points)
pc_pairs = pc_pairs.map(lambda (x,y): ((x[1][0], x[1][1]), y[1][2]) if ( y[1][0] >= x[1][0] -
    window_size/2 and y[1][0] <= x[1][0] + window_size/2 and y[1][1] >= x[1][1] - window_size
    /2 and y[1][1] <= x[1][1] + window_size/2 ) else None)
pc_pairs = pc_pairs.map(lambda x: x)
pc_win_pairs = windows_rdd.cartesian(pc_pairs)
```

Code 1: Cartesian approach

2.2 K-D Tree for finding neighbours

Once the first strategy was discarded after finding that our approach was not viable we decided to improve the initial way of finding the neighbours implemented with brute force with the use of a binary tree. In order to improve the complexity, which was the biggest hick-up of the program, we decided to use K-D Tree[1] structure for quick neighbours finding. Although, the worst case for querying the tree i $O(N)$, the average case has complexity $O(log(N))$, thus finding the neighbours is no longer $O(N^2 * M)$ but $O(Nlog(N) * M)$ what drastically reduces run time. For instance when using N = 1 000 000 points and M = 4, we do not have to do $4 * 10^{12}$ but only $2.4 * 10^7$. The new approach consists on mapping each point from the pointcloud the evaluation of the built tree when querying for neighbours within a distance corresponding to half of the window size. A small snippet of code representing this idea is presented in the next code block. As can be seen, the KDTree is broadcasted in order to make it available to all the workers and a mapping is performed for each point in the pointcloud in which the function applied is the neighbour search based on the size of the window.

```
def getNeighbours(point):
        return np.array(kdt_broadcast.value.query_radius(point, (window_size_broadcast.value
    /2.0))[0], dtype='int32')

# building K-D Tree
    kdt = KDTree(np_pointcloud[:,0:2])
    kdt_broadcast = sc.broadcast(kdt)
    tmp = pc_parsed.map(lambda x: ((x[0],x[1]), [point[2] for point in np_pointcloud[
    getNeighbours(x[0:2])]])).mapValues(min).map(lambda (x,y): [x[0], x[1], y])
```

Code 2: KDTree code

2.3 Erosion and dilation

According to the paper[5], one of the bottlenecks was erosion and dilation, which are nothing else than finding minimum and maximum values in the list considering that this involved finding the neighbours. Even though, they are very basic operations, they are performed a lot of times consequently constituting the most of the run time. Thanks to Spark implementation, this is now done in parallel in multiple clusters with multi-threats. This approach requires performing the action (collection) after each erosion, because dilation has to be applied to updated data set that is manipulated with erosion. As opposed to previously presented cartesian approach, the algorithm is not a one spark block but one window iteration requires 2 actions. In any case, applying these operations is still efficient, being the bottleneck of our approach the neighbour search.

```
# erosion
    tmp = pc_parsed.map(lambda x: ((x[0],x[1]), [point[2] for point in np_pointcloud[
    getNeighbours(x[0:2])]])).mapValues(min).map(lambda (x,y): [x[0], x[1], y])
    pc_eroded = np.array(tmp.collect())

    # dilation
    pc_dilated = pc_parsed.map(lambda x: ((x[0],x[1]), [point[2] for point in pc_eroded[
    getNeighbours(x[0:2])]])).mapValues(max)
```

3 Amazon Setup

In order to test the implemented algorithm as well as for learning purposes we managed and used a full stack Amazon AWS setup for the Pyspark implementation of our code. In sections 3.1, 3.2 and 3.3 the whole setup is explained in detail in order to keep the process documented for future projects.

3.1 Simple Storage Service (S3)

Amazon S3 (Simple Storage Service) provides a reliable and flexible storage system for multiple kinds of applications. Once a bucket is obtained, the billing process varies based on the amount of storage used as well as the band-with consumed during the bucket lifetime. In our case, the S3 bucket serves as a container both for the application code, the pointclouds used in the algorithm as well as the EC2 instance logs from the Master and Slaves.

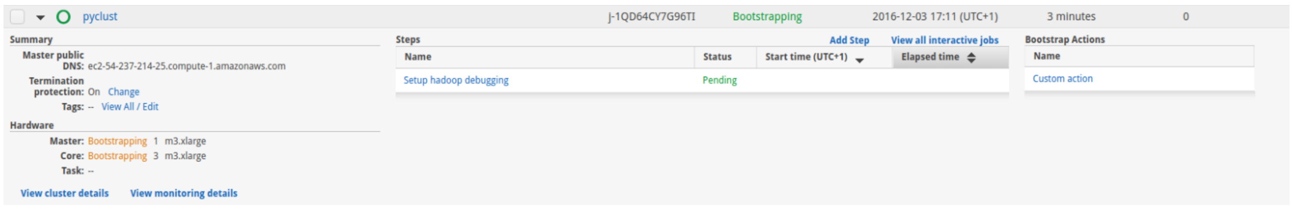
[1]Definition: K-D Tree is a multidimensional search tree for points in k dimensional space. Levels of the tree are split along successive dimensions at the points.

3.2 Elastic Compute Cloud (EC2)

Amazon EC2 (Elastic Compute Cloud) is a service which allows to make use of virtual servers based on the demands of each particular application. In our project we made use of 4 m3.xlarge instances which provide 4 threads and 15GB of RAM per node in a 1 Master + 3 Workers setup. Due to the need of specific dependencies for our application, all the machines have been provided with the libraries via bootstrapping when creating the cluster as can be seen in Section 3.3

3.3 Elastic Map Reduce (ERM)

Amazon ERM (Elastic Map Reduce) is a service that uses other AWS products in order to build a large-scale data processing cluster. Using the EC2 instances mentioned in 3.2 as can be seen in 2 a cluster with Spark has been created in order to process the PMF code. The dependencies have been processed via bootstrapping when creating the nodes in the cluster in which Scipy, Numpy and Scikit-learn is installed in order to be able to use the KDTree implementation.



pyclust   j-1QD64CY7G96TI   Bootstrapping   2016-12-03 17:11 (UTC+1)   3 minutes   0

Summary

Master public DNS: ec2-54-237-214-25.compute-1.amazonaws.com

Termination protection: On Change

Tags: -- View All / Edit

Hardware

Master: Bootstrapping 1 m3.xlarge

Core: Bootstrapping 3 m3.xlarge

Task: --

View cluster details   View monitoring details

Steps   Add Step   View all interactive jobs

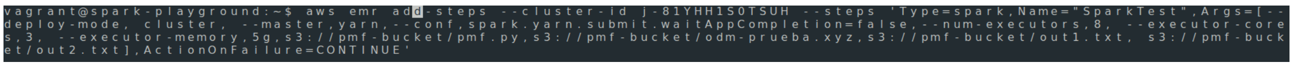
Name	Status	Start time (UTC+1)	Elapsed time
Setup hadoop debugging	Pending		

Bootstrap Actions

Name
Custom action

Figure 2: Running Amazon ERM

In order to make use of the cluster, we used the amazon CLI in which we specified the arguments needed to send the code to the Master as can be seen in 3.



```
vagrant@spark-playground:~$ aws emr add-steps --cluster-id j-81YHH1S0TSUH --steps 'Type=spark,Name="SparkTest",Args=[--
deploy-mode, cluster, --master,yarn,--conf,spark.yarn.submit.waitAppCompletion=false,--num-executors,8, --executor-core
s,3, --executor-memory,5g,s3://pmf-bucket/pmf.py,s3://pmf-bucket/odm-prueba.xyz,s3://pmf-bucket/out1.txt, s3://pmf-buck
et/out2.txt],ActionOnFailure=CONTINUE'
```

Figure 3: Submitting job to EMR

4 Results

Taking into consideration the complexity of the algorithm ($O(Nlog(N) * M)$) it was not feasible to work with data set even of 1GB. In this project, the weight of the data set does not reflect the size of the problem. Even though we run the program using relatively small data (650MB) it was containing around 10 000 000 points with in (X, Y, Z) format. As presented in Table 1, our benchmark was set by naively implemented python program ran on data set with 43 000 points that on our computer[2] took around 50 minutes. Due to the complexity, it was not possible to benchmark this setup with any bigger data set. With spark implementation run on the same machine, we managed to reduce the time to less than 2 minutes which was already a great improvement. The same program uploaded to Amazon was ran only in just 14 seconds. Consequently, when run on laptop the speedup is ≈27 while and using Amazon service ≈215.

Next, to challenge spark implementation, we replicated our point cloud 9 times, which gave around 387k points and weighted ≈25MB. This time it took over an hour on our laptop and just 8minutes on Amazon.

The last tested data set contained 10 000 000points and weighted 650MB. It was replicates of our initial point cloud. took almost 10hours on Amazon and was impossible to be ran on our computer.

Note that in this kind of problem the main problem related to Big Data is related to processing instead of storage problems. Considering this, the values of the data set sizes are emphasized in terms of points. For each point a set of mathematical operations are performed, process which yields a big complexity in the problem.

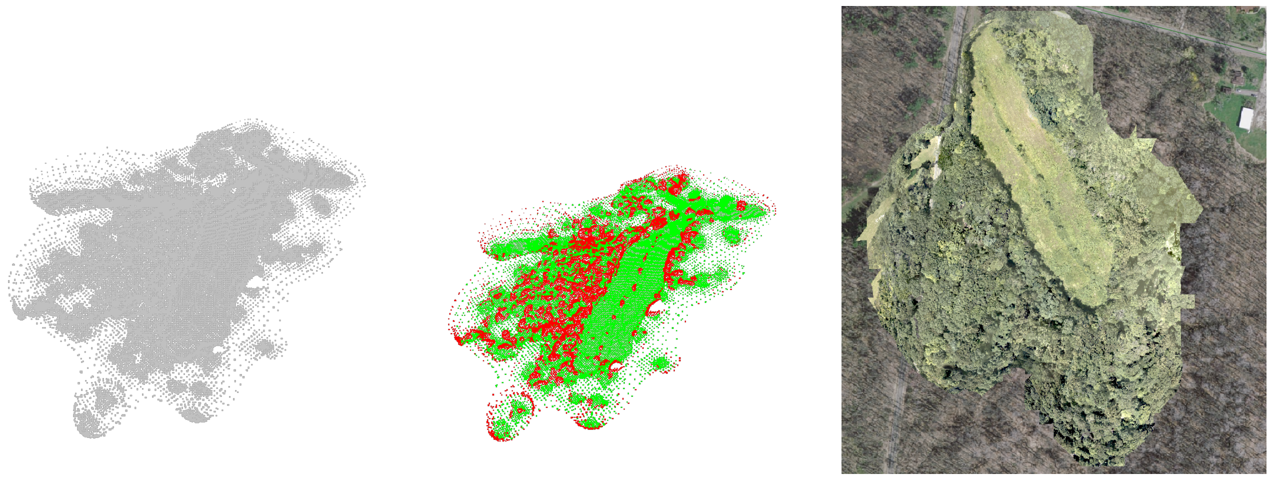
Next figure shows the smallest tested pointcloud before been processed.

Finally an image of the biggest pointcloud is attached. As stated above, this was computed with PySpark in an Amazon server. As we can observe, the output is what we expected in which the red colored points represent non-ground points and from the original 2D set of images from which the pointcloud was built, it is easy to observe how they represent trees.

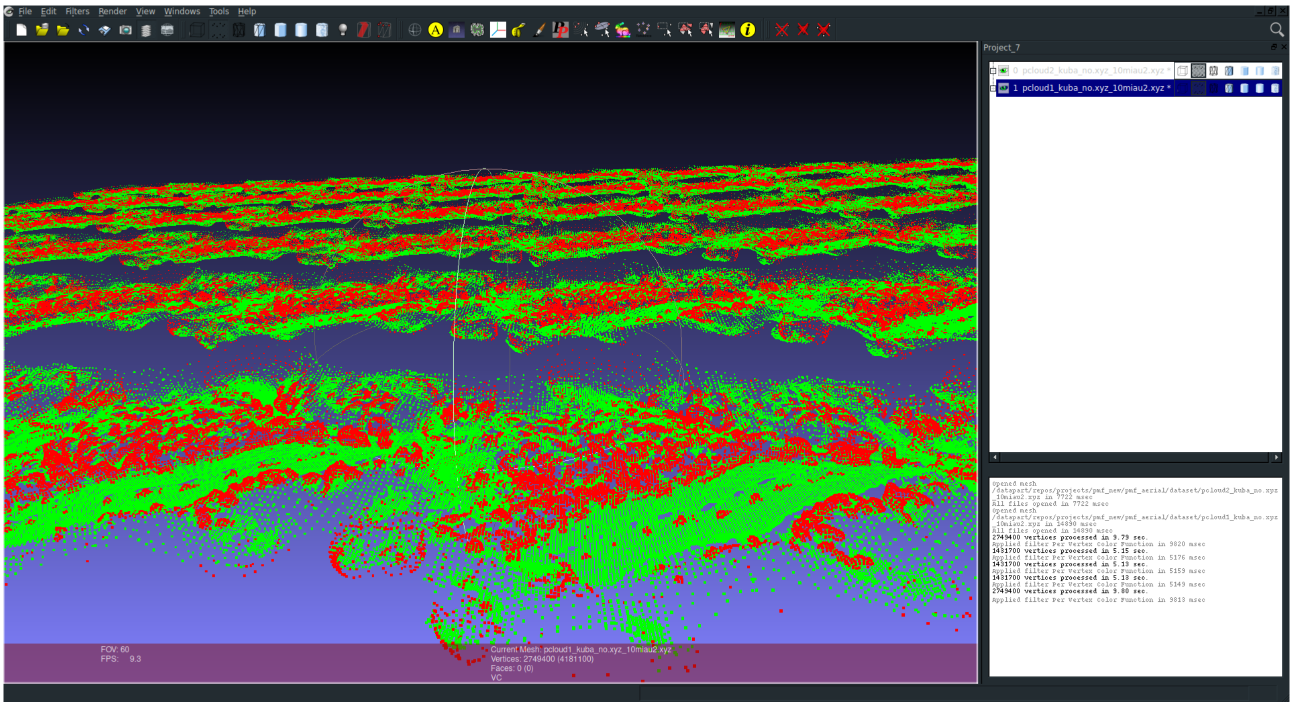
[2]Skylake i3-6300U @ 2.3GHz

#Points	Data set size	Naive	Spark laptop	Spark Amazon
43.000	2.7MB	3.000s = 50mins	110s	14s
387.00	25MB	-	3.720s = 62mins	480s = 8mins
10.000.000	650MB	-	-	9.5h

Table 1: Summary of the experiments



(a) Original pointcloud (b) Original pointcloud processed (c) Othophoto


File
Edit
Filters
Render
View
Windows
Tools
Help
Project_7
0 pcloud2_kuba_no.xyz_10miau2.xyz *
1 pcloud1_kuba_no.xyz_10miau2.xyz *

```
FOV: 60
FPS:     9.3
Current Mesh: pcloud1_kuba_no.xyz_10miau2.xyz
Vertices: 2749400 (4181100)
Faces: 0 (0)
VC
```

```
Opened mesh
/datapart/repos/projects/pmf_new/pmf_aerial/dataset/pcloud2_kuba_no.xyz
_10miau2.xyz in 7722 msec
All files opened in 7722 msec
Opened mesh
/datapart/repos/projects/pmf_new/pmf_aerial/dataset/pcloud1_kuba_no.xyz
_10miau2.xyz in 14890 msec
All files opened in 14890 msec
2749400 vertices processed in 9.79 sec.
Applied filter Per Vertex Color Function in 9820 msec
1431700 vertices processed in 5.15 sec.
Applied filter Per Vertex Color Function in 5176 msec
1431700 vertices processed in 5.13 sec.
Applied filter Per Vertex Color Function in 5159 msec
1431700 vertices processed in 5.13 sec.
Applied filter Per Vertex Color Function in 5149 msec
2749400 vertices processed in 9.80 sec.
Applied filter Per Vertex Color Function in 9813 msec
```

Figure 5: The largest pointcloud processed

5 Encountered technical issues

While translating the standard Python code to Spark, we encountered a couple of technical issues and actual bugs. First of them was not correct output of chain cartesian product. It turned out that there exist a bug in PySpark that for large dataset cartesian products the result yielded less points than the actual product.

The second problem we came across was related to broadcasting K-D Trees from master to workers. In order to make it possible for spark workers to quickly look up the neighbours in the tree we had to broadcast the tree first to make it available to the workers. While doing that, Spark automatically pickles (serializes) the objects and sends them to all the workers. Although, there was no problem with doing this, the distributed object was empty meaning the tree was not correctly pickled. Interesting fact is that when we tried to pickle it manually it was working just fine. On the end we decided to use different implementation (Scikit-learn instead of Scipy.spatial) of K-D Tree that did not have this issue when serializing the object.

Finally, the last problem we faced was caused by the Vagrant virtual machine that was distributed during the course. The problem seemed to be caused by an old version of Spark, that was working correctly with extra small files and crushing with others. On the end we found another Vagrant VM that had installed Spark v2.0.0 and could successfully process even bigger data sets locally.

6 Future work

Our final implementation still lacks the parallelization we aimed in the beginning. One of the main ideas was to develop the whole implementation avoiding for loops and using only Spark actions. Once our first attempt was discarded we still need to map all the points in a loop-fashion even though this can be paralelized due to the fact that each iteration does not rely on a previous one for the neighbour search. As a result, we still have some linear code in the implementation. As a future work, one of the ways of improving the performance would be exploring implementations of binary tree search implemented in Spark code but at a first glance we did not find any available option.

Another way of improving the performance is to test different scenarios and setups for the cluster structure, parameters and servers used. As a result of having to change our approach we could not experiment as much as we wanted initially with the EMR resources. This fact could potentially bring a better performance.

7 Appendix

7.1 Spark implementation

```
import numpy as np
import sys
import time
from sklearn.neighbors import KDTree
import pyspark

if __name__ == "__main__":
    def getNeighbours(point):
        return np.array(kdt_broadcast.value.query_radius(point, (window_size_broadcast.value
    /2.0))[0], dtype='int32')

    start_time = time.time()

    # Build different windows and height thresholds
    windows = []
    height_thresholds = []

    # Default parameters
    slope = 1.0
    cell_size = 1.0
    base = 2.0
    max_distance = 1
    initial_distance = 0.5
    max_window_size = 20
    window_type = 'linear'

    window_size = 0.0
    i = 0
    while window_size < max_window_size:
        # Create different windows
        if window_type == 'linear':
            window_size = cell_size * ((2*(i+1)*base) + 1)
        elif window_type == 'exponential':
            window_size = cell_size * ((2*base**(i+1) + 1))
        if window_size > max_window_size:
            break
        windows.append(window_size)
        # Create corresponding height threshold
        if i == 0:
            height_threshold = initial_distance
        elif height_threshold > max_distance:
            height_threshold = max_distance
        else:
            height_threshold = slope*(windows[-1]-windows[-2])*cell_size + initial_distance


        height_thresholds.append(height_threshold)
        i += 1

    # load file
    sc = pyspark.SparkContext()
    pc= sc.textFile(sys.argv[1])
    pc_split = points.map(lambda x: (x.split(" ")))
    pc_parsed = pc_split.map(lambda line: [float(line[0]), float(line[1]), float(line[2])])
  np_pointcloud = np.array(pc_parsed.collect())

  # building K-D Tree
    kdt = KDTree(np_pointcloud[:,0:2])

    window_size = windows[0]
    thres = height_thresholds[0]
    kdt_broadcast = sc.broadcast(kdt)
    window_size_broadcast = sc.broadcast(window_size)

  # erosion
    tmp = pc_parsed.map(lambda x: ((x[0],x[1]), [point[2] for point in np_pointcloud[
    getNeighbours(x[0:2])]])).mapValues(min).map(lambda (x,y): [x[0], x[1], y])
    pc_eroded = np.array(tmp.collect())

    # dilation
    pc_dilated = pc_parsed.map(lambda x: ((x[0],x[1]), [point[2] for point in pc_eroded[
    getNeighbours(x[0:2])]])).mapValues(max)
```

```

     # assigning flags
   pc_parsed_tuple = pc_parsed.map(lambda x: ((x[0], x[1]), x[2]))
    flags = pc_parsed_tuple.cogroup(pc_dilated).map(lambda (x,y): (x,(list(y[0])[0],list(y[1])[0])) if(list(y[0])[0] != "" and list(y[1])[0] != "") else None)
    flags = flags.map(lambda (key, tuple): (key,1) if(abs(tuple[0] - tuple[1]) > thres) else (key,0))


  # one iteration is done outside of the loop in order to initialize the flags
    i = 0
    for window, thres in zip(windows, height_thresholds):
        if i > 0:
            window_size_broadcast = sc.broadcast(windows[i])
            thres = height_thresholds[i]

      # erosion
            tmp = pc_parsed.map(lambda x: ((x[0],x[1]), [point[2] for point in np_pointcloud[getNeighbours(x[0:2])]])).mapValues(min).map(lambda (x,y): [x[0], x[1], y])
            pc_eroded = np.array(tmp.collect())

            # dilation
            pc_dilated = pc_parsed.map(lambda x: ((x[0],x[1]), [point[2] for point in pc_eroded[getNeighbours(x[0:2])]])).mapValues(max)

            # assigning flags
            pc_parsed_tuple = pc_parsed.map(lambda x: ((x[0], x[1]), x[2]))

            flags_new = pc_parsed_tuple.cogroup(pc_dilated).map(lambda (x,y): (x,(list(y[0])[0],list(y[1])[0])) if(list(y[0])[0] != "" and list(y[1])[0] != "") else None)
            flags_new = flags_new.map(lambda (key, tuple): (key,1) if(abs(tuple[0] - tuple[1]) > thres) else (key,0))
            flags = flags_new.cogroup(flags)
            flags = flags.map(lambda (x,y): (x,(list(y[0])[0],list(y[1])[0]))).map(lambda (x,y): (x, (y[0] or y[1])))
        i+=1

  # process format of the flags
    output = flags.cogroup(pc_parsed_tuple).map(lambda (x,y): ((x[0],x[1],list(y[1])[0]),list(y[0])[0]))
    non_ground = output.filter(lambda (x,y): y == 1).map(lambda (x,y): x)
    ground = output.filter(lambda (x,y): y == 0).map(lambda (x,y): x)

    end_time = time.time()
    print "\nExecution time: ", (end_time-start_time), "sec"

  # Save ground & non-ground points to files
    non_ground.saveAsTextFile(sys.argv[2])
    ground.saveAsTextFile(sys.argv[3])
```

Code 3: Progressive Morphological Filter PySpark implementation

7.2 Naive implementatation

```

import numpy as np
import csv
import sys

#Erosion function
def erosion(pointcloud, neighbours, point_idx):
  z_vals = [pointcloud[neighbour,-1] for neighbour in neighbours]
  pointcloud[point_idx, -1] = min(z_vals)
  return pointcloud

#Dilation function
def dilation(pointcloud, neighbours, point_idx):
  z_vals = [pointcloud[neighbour,-1] for neighbour in neighbours]
  pointcloud[point_idx, -1] = max(z_vals)
  return pointcloud


#Find neighbours for a given point and window size
def boundbox_neighbours(point, pointcloud, window_size):

  minx = point[0] - window_size/2.0
  miny = point[1] - window_size/2.0
  maxx = point[0] + window_size/2.0
  maxy = point[1] + window_size/2.0

  neighbour_inds = []
  for point_idx, p in enumerate(pointcloud):
    if (p[0]<=maxx and p[0]>=minx and p[1]<=maxy and p[1]>=miny):
      neighbour_inds.append(point_idx)
  return neighbour_inds


#Load .xyz file where point cloud is stored
with open('../dataset/odm_mesh_small.xyz', 'rb') as csvfile:
  csvreader = csv.reader(csvfile, delimiter=' ')
  pointcloud = [map(float, row) for row in csvreader]

np_pointcloud = np.array(pointcloud)

# Build different windows and height thresholds
windows = []
height_thresholds = []

# Default parameters
slope = 1.0
cell_size = 1.0
base = 2.0
max_distance = 80
initial_distance = 10
max_window_size = 20
window_type = 'linear'

window_size = 0.0
i = 0

while window_size < max_window_size:
  # Create different windows
  if window_type == 'linear':
    window_size = cell_size * ((2*(i+1)*base) + 1)
  elif window_type == 'exponential':
    window_size = cell_size * ((2*base**(i+1) + 1))
  if window_size > max_window_size:
    break
  windows.append(window_size)
  # Create corresponding height threshold
  if i == 0:
    height_threshold = initial_distance
  elif height_threshold > max_distance:
    height_threshold = max_distance
  else:
    height_threshold = slope*(windows[-1]-windows[-2])*cell_size + initial_distance

```

```python
  height_thresholds.append(height_threshold)
  i += 1


flags = np.zeros(np_pointcloud.shape[0])

#Create a copy of the original file
pointcloud_copy = np.copy(np_pointcloud)

for window, thres in zip(windows, height_thresholds):

  for point_idx, point in enumerate(pointcloud_copy):

    neighbours = boundbox_neighbours(point, pointcloud_copy, window)

    if neighbours:
      # Open operator (erosion + dilation)
      pointcloud_copy = erosion(pointcloud_copy, neighbours, point_idx)
      pointcloud_copy = dilation(pointcloud_copy, neighbours, point_idx)


  for point_idx in range(np_pointcloud.shape[0]):
    if ( abs(np_pointcloud[point_idx,-1] - pointcloud_copy[point_idx,-1])) > thres:
      flags[point_idx] = 1


#Create file with non ground points
with open('../dataset/ground.xyz', 'wb') as csvfile:
  csvwriter = csv.writer(csvfile, delimiter=' ')
  csvwriter.writerows(np_pointcloud[np.where(flags == 1)[0], :].tolist())

#Create file with ground points
with open('../dataset/nonground.xyz', 'wb') as csvfile:
  csvwriter = csv.writer(csvfile, delimiter=' ')
  csvwriter.writerows(np_pointcloud[np.where(flags == 0)[0], :].tolist())
```
Code 4: Progressive Morphological Filter naive python implementation

References

[1] Paul E Black. *Dictionary of algorithms and data structures*. National Institute of Standards and Technology, 2004.

[2] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Reading, Massachusetts: Addison-Wesley, 1993.

[3] *Open Drone Map project*. URL: `https://github.com/OpenDroneMap/OpenDroneMap`.

[4] *Point Cloud structure*. URL: `https://en.wikipedia.org/wiki/Point_cloud`.

[5] Keqi Zhang et al. “A progressive morphological filter for removing nonground measurements from airborne LIDAR data”. In: *IEEE Transactions on Geoscience and Remote Sensing* 41.4 (2003), pp. 872–882.