

# A SIMPLE ORDERING SOLUTION FOR EFFICIENT GEOMETRICAL LEVEL OF DETAILS ON LIDAR POINT CLOUD

Rémi Cura, etc.

## Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>	
1.1	Problem . . . . .	2	
1.2	Motivation . . . . .	2	
1.3	state of the art . . . . .	2	
1.4	what's missing in biblio . . . . .	2	
1.5	contribution . . . . .	2	
1.6	plan of the article . . . . .	2	
<b>2</b>	<b>Method</b>	<b>3</b>	
2.1	introduction . . . . .	3	
2.2	exploiting the order of points . . . . .	3	
2.2.1	principle . . . . .	3	
2.2.2	conceptual example . . . . .	3	
2.2.3	advantages . . . . .	3	
	no on-line processing time . . . . .	3	
	no data duplication . . . . .	3	
	portable . . . . .	3	
2.3	MidOc : an ordering for gradual geometrical approximation . . . . .	3	
2.3.1	principle . . . . .	3	
2.3.2	implementation . . . . .	4	
2.3.3	efficiency and performance example . . . . .	4	
2.3.4	conceptual example . . . . .	4	
2.3.5	advantages . . . . .	4	
	Well known, Simple and efficient . . . . .	4	
	fixed density . . . . .	5	
	permissive . . . . .	5	
	construction descriptors . . . . .	5	
2.4	using the ordering by-product as a crude dimensionality descriptor . . . . .	5	
2.4.1	principle . . . . .	5	
2.4.2	conceptual example . . . . .	5	
2.4.3	method . . . . .	5	
	crude dimensionality descriptor . . . . .	5	
	other simple features . . . . .	5	
	choosing class in class hierarchy . . . . .	5	
	balancing data set . . . . .	5	
	classifying . . . . .	5	
2.4.4	advantages . . . . .	6	
	simple . . . . .	6	
	Efficient . . . . .	6	
	Density and scale independent . . . . .	6	
	Mixed result . . . . .	6	
<b>3</b>	<b>Result</b>	<b>6</b>	
3.1	introduction to all experiments . . . . .	6	
3.2	Point Cloud server introduction . . . . .	6	
3.2.1	principle . . . . .	6	
	server . . . . .	6	
	fast filtering . . . . .	6	
	parallelism friendly . . . . .	6	
	point cloud splitting . . . . .	6	
3.3	Data Set used . . . . .	6	
3.3.1	IQmulus data set . . . . .	7	
3.3.2	Vosges data set . . . . .	7	
3.4	Exploiting the order of points . . . . .	7	
3.4.1	experiment summary . . . . .	7	
3.4.2	results . . . . .	7	
3.5	MidOc : an ordering for gradual geometrical approximation . . . . .	7	
3.5.1	experiment summary . . . . .	7	
3.5.2	visual evaluation . . . . .	7	
	Visual evaluation on typical objects (ground, facade, car, pole, vegetation) . . . . .	7	
	visual evaluation aerial lidar . . . . .	7	
3.5.3	size versus LOD tradeoff . . . . .	7	
3.5.4	large scale computing . . . . .	7	
	MidOc implementation . . . . .	7	
	Computing on very large dataset . . . . .	7	
3.5.5	LOD stream . . . . .	7	
3.6	using the ordering by-product as a crude dimensionality descriptor . . . . .	7	
3.6.1	experiment summary . . . . .	7	
3.6.2	choosing classes in class hierarchy . . . . .	8	
	canonical classification . . . . .	8	
	balancing data set . . . . .	8	
	classifying . . . . .	8	
<b>4</b>	<b>Illustrations</b>	<b>8</b>	
4.1	MidOc ordering . . . . .	8	
4.2	Crude dimensionality descriptor . . . . .	8	

# 1. Introduction

## 1.1. Problem

Point cloud data is becoming more and more common. Following the same trend, the acquisition frequency and precision are also increasing. Thus Lidar processing is clocking on the Big Data door.

Yet the usage of point cloud data is also spreading and going out of the traditional user communities. Lidar are now commonly used by non-specialized users.

For many usages, having the raw, complete point cloud is unnecessary, or even damageable. Thus we deal with a simpler version of a problem that the vector community has faced for a long time : how to generalize point cloud, with data sets that are several order of magnitude bigger than usual vector data set ?

It is after all a problematic very common in data processing. Having a big data set, how to reduce its size while preserving its characteristics. It is the essence of compression for instance.

Generalization is also more difficult when mixing data set with varying densities. For instance an aerial Lidar map augmented at certain places by terrestrial scanners, or vehicle-based Lidar acquisition, where the density varies with speed and scene geometry.

Here we deal with a simplified version : given a point cloud, how to efficiently generate Level Of Detail (LOD for the remainder of this article) of this point cloud while preserving the geometric characteristic, without duplicating data. The key to LOD approach is efficiency. We accept to loose a part of the information in exchange of a massive reduction of data size. A solution using LOD must then by nature be efficient.

## 1.2. Motivation

- Point cloud : becoming common (why) Point cloud are becoming common because sensor are smaller, cheaper, easier to use. Point cloud from image (using Stereo Vision) are also easy to get with several mature structure from motion solutions. Point cloud complements very well images, Lidar point cloud allowing to avoid the ill-posed problem of stereo-vision, and providing key data to virtual reality.
- Growing data set + Multi sources At such the size of data set are growing, as well as the number of dataset and their diversity.
- Why is it important (size of the industry) The point cloud data are now well established in a number of industries, like construction, architecture, robotics, archaeology, as well as all the traditional GIS fields (mapping, survey, cultural heritage)
- PointCloud users = specialist in processing, not informatics/storing The LIDAR research community is very active. The focus of Lidar researchers is much more on Lidar processing and Lidar data analysis, or the sensing device, than on methods to render the data size tractable.

## 1.3. state of the art

RC:1.3.0. when I have access to Zotero

State of the art should include

- what people do with point cloud ? (oosterom 2014)

## 1.4. what's missing in biblio

Point cloud generalization methods are far from the subtlety of vector generalization ( // mettre des references).

More generally, proposed methods usually focus on data compression and computing acceleration.

Another common practice coming from Computer Graphics field is to compute an octree over the point cloud, then for each cell , compute a sub-sampled version of the point cloud. This allows to have simple and efficient Level of Detail , at the price of data duplication, and high sensibility to density variation.

Moreover, all the method are specific and depend on a specific data structure that has to be stored extra to the point cloud data. Steaming from this the interoperability is non existent (moving from one software to another, one loose the data structure).

We have then the classical and seemingly intractable trade-off between computing and storage : if we pre-compute a data structure, we have to store it extra. If we do the computing on the fly, we may end up performing the same operations a lot's of time.

## 1.5. contribution

In this paper, we focus on simplicity, efficiency and re-use of existing and well established methods. All the methods are tested on Billion scale point cloud, and are open source for reproducibility test and reuse. We propose a simple method that enable portable computation free geometrical level of detail. The first contribution is that we propose to store the LOD information directly into the ordering of points rather than compute new sub-sampled point cloud for each LOD. Thus, the more we read points, the more precise of an approximation of the point cloud we get. If we read all the points, we have the original point cloud.

The second contribution is a simple way to order points so that we have a varying geometric approximation of the point cloud when following this order.

The third contribution is to use the ordering construction by-product as simple and free dimensionality descriptors, with usability demonstrated in a Random forest classification experiment.

## 1.6. plan of the article

The rest of this article is organized as follow : In the next section we present the methods. In the section XX, we present some results and order of magnitude In the last section we discuss the results, the limitation and improvements of our solution.

## 2. Method

### 2.1. introduction

In this section we introduce a simple method which proposes free geometrical LOD features, at the price of a small preprocessing time. This method has been designed to work with Lidar data set, but may be used on noisy SfM point clouds after filtering. Similarly, this method has been used inside our point cloud data management system which is centred on a Point Cloud Server (the article is being written, see [] for a very detailed presentation).

It is then crucial to understand that all the experiment are not performed at the point cloud level, but at the patch level. Patch are groups of points obtained by cutting the original point cloud into regular pieces (1m cube for Paris data set, 50 meter cube for Vosges dataset). Patch are technically a subset of a point cloud, so also a point cloud. For generality we will use point cloud to describe both patch and global point cloud for the rest of this article.

This because our method can be used indifferently on patches and on the global point cloud. Lastly computing this LOD gives interesting by-product that can be used as crude classifiers, and allow to perform extremely fast rough-filtering of massive point cloud.

### 2.2. exploiting the order of points

#### 2.2.1. principle

We propose a very simple yet efficient method to provide geometrical LOD.

We start from a patch (i.e. a piece of point cloud) and we want to generate geometrical levels of details on it.

Point clouds are physically stored as a list of points.

Now, the information contained by a list of point is not the sum of information for each point. The big difference is that a list is ordered.

This fact is common for every point cloud format. At one point, points must be written to disk, and they become ordered. In fact it would be quite hard to store the point in a totally unordered fashion (true random is expansive). The first contribution of this article is to propose to exploit this ordering to store information. The key idea is : we order the points of the point cloud so that when reading the points from beginning to end, we get gradually a better and better geometrical approximation of the point cloud.

We demonstrate it with a given ordering ("MidOc"), but we could use any ordering.

#### 2.2.2. conceptual example

So for instance if we have a list L of points ordered from 1 to n. Reading the points 1 to 5 is going to give a rough approximation of the point cloud. Reading another 16 (points 1 to 21) is going to give a slightly better approximation. Reading points 1 to n is going to get the exact point cloud, so there is no data loss, nor data duplication.

#### 2.2.3. advantages

*no on-line processing time.* The advantages of this method is that except a pre-processing step to write the point cloud following the MidOc ordering, each time we want to get a level of detail version of the point cloud, there is no computing at all (only data reading). This may not make a big difference for non-frequent reading, but in a context where the same point cloud is going to get used several time at several level and by several users simultaneously (for instance Point Cloud as a Service), no processing makes a big difference. (See result section for an example with LOD visualisation.)

*no data duplication.* Another big advantage of using the order of points to store the LOD information rather than using external structure data is that it is free storage wise. This is an advantage on low level. First we need less space on disk. Second we are sure that all the information is at the same place, which avoids to perform OS level commands like going through directories, opening the structure file, etc. Having no data duplication is also a security in concurrent use. If somebody changes the point cloud, another user could use the old structure with the new points in the time where the new structure is computed. On the opposite, when the ordering gives the LOD information, a user either get the old points with the old ordering, or the new points with the new ordering, which guarantee that data is in a coherent state. Lastly by ordering rather than creating a sub sampled version, we avoid all precision-related issues, because the order doesn't change existing points and their attributes, but only the order in the point cloud.

*portable.* The last advantage comes from the simplicity of using the ordering. Because it is already something that all point cloud tools can deal with (a list of points !), this way to create LOD is portable. Most software don't change the points order inside a cloud. If a tool changes the order, it is easy to add the ordering number as an attribute, which increase a little bit storage size, but is totally portable and can be used with all existing tools. This simplicity also implies that adapting tools to use this ordering is very easy.

### 2.3. MidOc : an ordering for gradual geometrical approximation

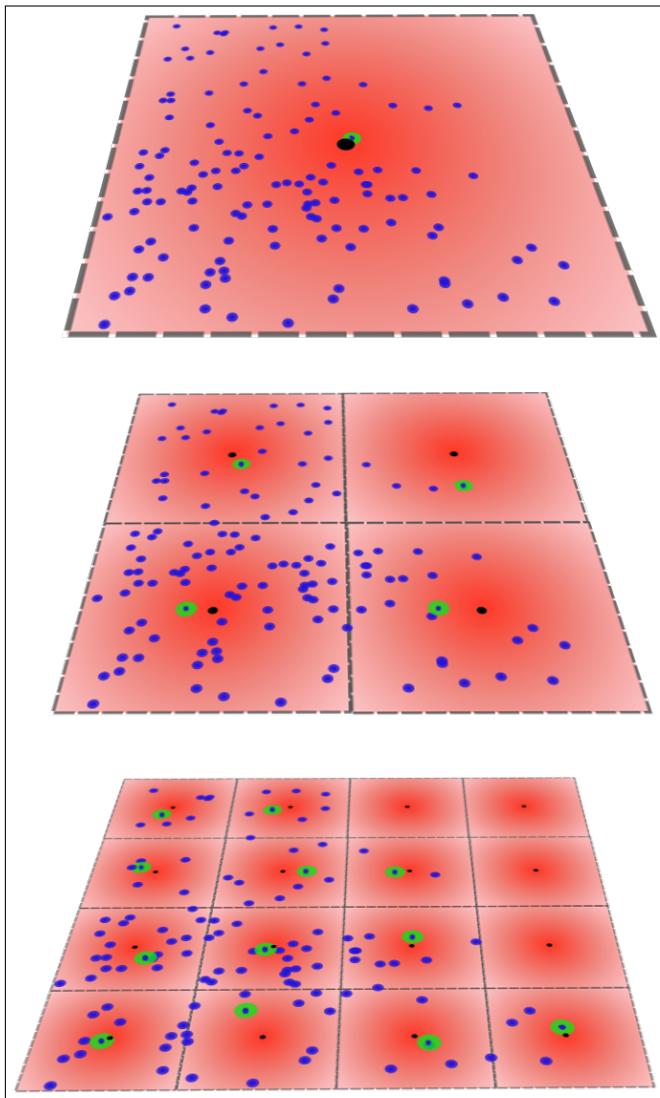
#### 2.3.1. principle

The key of LOD is efficiency, because using LOD is already a trade-off between data information loss and data size reduction. Thus we need an efficient method that order the points so that reading from the start toward the end is going to give a gradually better geometrical approximation of the point cloud.

We can do this trade-off only because Lidar point clouds have intrinsic structure we must exploit. Precisely because we do a trade-off, we must exploit the intrinsic structure of the point cloud. For this, we make some assumption that are mostly verified on Lidar point cloud. - Point cloud represents 3D surfaces sensed by a fix or

mobile sensor (with the exception of multi-echo, which is correctly dealt with anyway). - geometrical noise (outliers) is low. - the density may vary, but we don't want to preserve it, nor does it give information about the nature of the object being sampled.(that is, depending on the sensing situation, we may have parts of the cloud that are more or less dense, but this has nothing to do with the nature of the object sensed, thus must not be preserved).

We rely on the classical middle of octree subsampling (called MidOc in this article for simplicity) to create an ordering. This is a re-use of well known and well proven existing methods. We name this ordering for clarity of this article, nonetheless we don't think we are the first to use it.



**FIGURE 1:**

Again the principle is very simple, and is to take for each level the points closest to centre of its octree voxel, if any. This figure illustrate the principle on a quad tree for graphical comfort. The layer are level 0 to 2. For each layer we compute the closest point(large green point) to the center of each cell (medium black point) for squared euclidian distance (continuous red tone). The points pi-

cked are not available for further level (no duplication of information). When reaching the desired max level, all the remaining points are ordered either randomly or deterministically (for instance, with an inverted Z Morton curve (read backward)). We use the same random or deterministic order for points picked on the same level. Overall, the ordering is then (tree depth ascending, random or inverted Morton).

### 2.3.2. implementation

Overall the method has the same complexity as octree construction, and we can use similar strategies.

We have several possibilities for implementation, depending on what resources we can use. The more straightforward implementation is to loop through all points, and for each point, update both the octree and the closest point stored in each octree node walked through. At the end, we walk through the octree to collect chosen points. This is easy to parallelise if  $T$  is not too big. The worst complexity is  $O(N*T)$ .

We can also use a recursive strategy, where we compute level 0 to  $T$ , each time dividing the points into parts that are in the next level cell. We propose this kind of implementation in python with extensive use of bit-level operation as proof of concept.

### 2.3.3. efficiency and performance

In fact we don't really need to construct the octree because octree can be obtained by reading coordinates bitwise in a correctly centred/scaled point cloud. We centre a point cloud so that the lowest point is all dimension is  $(0,0,0)$ , and scale it so the biggest dimension is in  $[0,1]$ . Then we quantize this point cloud into  $[0,2^{**L}-1]$  for each coordinate. Coordinates are now integer, and for each points, reading the coordinates bitwise left to right gives the position of the point in octree for level 1-L. This means performing this centring/scaling/quantization directly gives the octree, and that further operation can be performed using bit arithmetic, thus be extremely fast.

*example.* For instance for a 2D point cloud with  $L = 3$ , and two points  $P1(4,3)$  and  $P2(5,2)$  already quantized in  $[0,7]$ . Their binary form is  $P1(100,011)$  ,  $P2(101,010)$ . Reading the first bit (left-right order) of the points gives the coordinate of the cell in the level 1 of the octree. for  $P1$  and  $P2$  it is  $(1,0)$ , which means the bottom right cell of the quad-tree. The second bit gives the position inside the second level of the octree. It is  $(0,1)$ , which means the upper left sub-cell of the previous pixel. The last bit is  $(0,1)$  for  $P1$  and  $(1,0)$  for  $P2$ , which allows to tell they won't be in the same cell. @

### 2.3.4. conceptual example

### 2.3.5. advantages

*Well known, Simple and efficient.* This method is extremely classical and based on Octree This make it simple

put a scheme with pyramid 4 levels, octal quad tree and bitwise coordinate with accolades

put example image of an area with very high density in the middle

to implement, and possibly extremely memory and CPU efficient.

The complexity is  $O(n*L)$  or less . Octree construction can even be performed on GPU for extreme speed up. It is a matter of days to implement it.

*fixed density*. This method guarantee an almost constant density for given levels, even when the acquisition process produced varying-density point cloud. Thus using the output of this method is a safeguard for most of the complex point cloud processing methods that may be badly affected by singularity in density.

*permissive*. This method is permissive because we make very few hypothesis on the points properties. In particular, this method works well with 2.5D point cloud (aerial single echo Lidar) and full 3D point cloud (urban 3D cloud with multi echo).

*construction descriptors*. Lastly, we can leverage the information given by this ordering as a good geometrical descriptor of the point cloud. We can keep the number of points per level for each level, which is a crude dimensionality descriptor of the geometrical nature (at the patch scale) for the object contained by the point cloud.

## 2.4. using the ordering by-product as a crude dimensionality descriptor

### 2.4.1. principle

During the ordering of a point cloud, we can keep the number of chosen point for each level. Now this number of chosen points per level gives an indication on the geometric nature of the object in the point cloud. We demonstrate the use of this extremely simple descriptor with a classical Random Forest classifier on a real world dataset publicly available. @

RC:2.4.1. cite Demantik® thesis because he creates sophisticated dimensionality indicators

We don't try to provide a per point classification, but a classification per group of points (i.e patch) . The idea is that for many patches (hence points) we may only use this simple classifier. So we could speed up a very complex classifier by first using the simple one, then use the complex one on parts that the simple classifier is not confident with. This is very similar to cascaded classifier or hierarchical classifier approach. Another application is filtering. If the classification process is very fast, one can easily eliminate large parts of the point cloud when looking for a given type of points (for instance, looking for ground points, or tree points).

### 2.4.2. conceptual example

For instance, we manually segmented interesting typical street extracts in the Paris dataset : acar, a wall with window, a 2 whelers, a public lightn a tree, a person, poles and piecee of ground including curbs. Due to geometry of acquisition and sampling, the public light is almost a 3D line, thus the points will be concentrated in very few octree cells. A typical number of points chosen per level for the public light would then be (1,2,4,8), thus average a

$2^{**L}$  function. A piece of ground is often extremely flat and very similar to a plan. Thus the points chosen per level could be (1,4,16,64), a  $4^{**L}$  function. Lastly a piece of tree foliage is going to be very volumetric in nature, due to fact that leaf are about the same size as point spacing and are partly transparent to laser (leading to several echo). So a foliage patch would typically be (1,8,64,512) (if enough points), so a  $8^{**L}$  function.

### 2.4.3. method

*crude dimensionality descriptor*. Again we don't work at the point level, but at the patch level (1 or 50 meter cube). We order all patches following MidOc ordering, and for each patch ordered, we associate the number of points per level that where chosen. We train a random forest classifier using the number of chosen points per level ( $V_{midoc}=[N1,n2,n3,n4\dots]$ ). We use the number of points per level 1 to 4 included. For each level, we normalize number of points by the maximum number of points possible ( $8^{**i}$ ), so every feature is in [0,1].

*other simple features*. To put things in perspective we also use other very simple features that are almost free computing wise. We can then analyse afterwards which features are most used. We limit ourself to use feature that can be obtained almost without computing because they are used by the compression mechanism at the patch level. The feature can then use any expression involving a min,max,average of any attribute. We also restrain to use contextual feature at all, because they are more in the spirit of complex classification, would require computing, and could introduce a bias (in our favor) in the result.

*choosing class in class hierarchy*. Both data sets have a class hierarchy. Because our goal is not to classify on all class, but efficiently filter patches, we first may want to determinate which classes are susceptible to be successfully classified using only this descriptor. We then can chose which level of the hierarchy we can use.

*balancing data set*. We tried two classical strategies to balance the data set regarding the number of observation per class. The first is undersampling : we randomly undersample the observations to get roughly the same number of observation in every class.

The second strategy is to compute a statistical weight for every observation based on the class prevalance. We then use this to weight the learning.

To ensure significant results we use a K-fold strategy. We randomly split the observations into K parts, then for each part, we use the K-1 others to learn and predict on the part.

All the evaluation are then performed on the total of predicted observations.

*classifying*. The second step is to perform classification to evaluate its potential. Contrary to classical classification method, we are not only interested in precision and recall per class, but also by the evolution of precision when prediction confidence varies.

In fact, for a filtering application, we can leverage the confidence information provided by the Random Forest method to artificially boost precision (at the cost of recall diminution). We can do this by limiting the minimal confidence allowed for every prediction.

We stress that if the goal is to detect objects (and not classify each point), this strategy can be extremely efficient. For instance if we are looking for objects that are big enough to be in several patches (e.g. a car). In this case we can perform the classification (which is very fast and efficient), then keep only highly confident predictions, and then use the position of predictions to perform a local search for car limits. In this case the classical solution would be to perform a per point classification on each point.

#### 2.4.4. advantages

*simple.* Dimensionality feature for point clouds are already well researched, and can be more precisely computed (?), with less sensibility to outliers (but more to density variation). However This kind of feature is generally designed at the point level, and is more complex. Using the result of the MidOc ordering has the advantage of being free and extremely simple.

*Efficient.* Moreover, because  $x$ ,  $x^{*}2$ ,  $x^{*}3$  diverge very fast, we only need to use few levels to have a quite good descriptor. For instance, using  $L=2$ , we have  $D=4$ ,  $16$  or  $64$ , which are very distinguishable values, and don't require a density above 70 points/patch.

*Density and scale independent.* As long as the patch contains a minimal number of points, the descriptors are density and scale invariant.

*Mixed result.* Lastly a mixed result (following neither of the  $x^{*n}$  function) can be used as an indicator that the patch contains mixed geometry, either due to nature of the objects in the patch, or due to the way the patch is defined (sampling).

## 3. Result

### 3.1. introduction to all experiments

We design and execute several experiments in order to validate all points that have been introduced in the "method" part. First we prove that it is effectively possible to leverage points order, even using canonical open sources software out of the box. Second we perform MidOc ordering on very large point cloud and analyse the efficiency and quality of the results. Third we use the number of points chosen in the MidOc ordering as a descriptors for a random forest classifier on two large data sets. We analyse the potential of this free descriptors, and what it brings when used in conjunction to other simple descriptors.

### 3.2. Point Cloud server introduction

#### 3.2.1. principle

*server.* All the experiments are performed using a Point Cloud Server (article is being written, but a very detailed presentation is accessible

RC3.2.1. put reference to postgres presentation

). The key idea is that point clouds are stored inside a DBMS (postgres), as patch. Patch are groups of points along with some basic statistics about points in the group. Patch are compressed using various strategies. This organisation is based on the observation that in typical point cloud processing workflow, we never need a point alone, but more often a points and most likely its surrounding points.

*fast filtering.* Each patch of points is then indexed in an R tree for most interesting attributes (obviously X and Y, but also time of acquisition, meta data, number of points, distance to source, etc.)

Having such a meta-type with powerful indexes allows use to find points based on various criteria extremely fast. (order of magnitude : ms). As an example we can find all points with given for a data set over 2 Billion points in a matter of milliseconds - between -1 and 3 meters high in reference to vehicle wheels - in a given 2D area defined by any polygon - close to a street called Rue Madame (according to IGN BDTopo) - between 3 and 5 meters to the sensor position - not in buildings according to Open Data Paris building layer - acquired in the second passage of the vehicle at this place - acquired between 8h and 8h10

*parallelism friendly.* Cutting a point cloud into patches provides also a very easy parallel processing possibility, which we extensively use in our experiments.

*point cloud splitting.* For our experiments we cut terrestrial Lidar point cloud into 1m3 cubes oriented on (North, East, Z) axis. We cut aerial lidar point cloud into 50 m cubes. The choice of size is a compromise between speed, index size, patch size, typical feature size, etc. In fact the patch can be cut arbitrary, we chose this splitting for simplicity.

### 3.3. Data Set used

We use two data sets.

### 3.3.1. IQmulus data set

#### First

RC:3.3.1. cite IQmulus data set

, an open source urban data set with varying density, singularities, and very challenging point cloud geometry. Data set is about 600 Millions points , over 12 kms of road. Points are typically spaced by 5cm to 0.2 cm. It is a multi echo laser (Riegl). We have access to a training set where every point is labeled in a hierarchy of 100 classes. The training set is only 12 millions points. Only 22 classes are represented. We will refer to this data set as Paris data set.

### 3.3.2. Vosges data set

We also use the Vosges data set, which is a very wide spread aerial data set of 6 billions points.

Density is much more constant to 10k pts/patch . We have access to a vector ground truth about surface occupation nature (type of forest), produced by the French Forest Agency. We will refer to this data set as the Vosges data set.

## 3.4. Exploiting the order of points

### 3.4.1. experiment summary

In this first experiment we check that point cloud ordering is correctly preserved by common open source point cloud processing software. For this, we use a real point cloud, which we order by MidOc ordering. We export it as a text file as the reference file. Then for each processing software, we read the reference file and convert it into another format, then check that the conversion didn't change the order. The three common open source software tested are CloudCompare, LasTools and Meshlab.

### 3.4.2. results

All software passes test. We stress that even if software change order, it is still very easy to add the order as an attribute, thus making it fully portable.

## 3.5. MidOc : an ordering for gradual geometrical approximation

### 3.5.1. experiment summary

In this experiment we first test ordering on typical street objects,then on terrestrial dataset to visually appreciate the fitness to use it for geometrical LOD. Then we compute MidOc for both our dataset and evaluate the trade-off between point cloud size and point cloud LOD. We briefly consider computing bottleneck.

Lastly as a proof of concept we stream 3D point cloud with various LOD to a browser.

### 3.5.2. visual evaluation

*Visual evaluation on typical objects (ground, facade, car, pole, vegetation). See fig XX*

RC:3.5.2. put correct renv

RC:3.5.2. Visual results on aerial (Vosges).

*visual evaluation aerial lidar. See fig XX*

RC:3.5.2. put correct renv

### 3.5.3. size versus LOD tradeoff

We compute the size and canonical transfer time associated for a representative street and aerial point cloud.

**TABLE 1:** number of points per LOD, plus estimated transfer time with modern internet connection

Level	Typical spacing(cm)	points number(k)	percent of total size	estimated time Internet(s)
All	0.2 to 5	1600	100	60
0	100	3	0.2	0.1
1	50	11.6	0.7	0.4
2	25	41	2.6	1.5
3	12	134	8.4	5
4	6	372	23	14

### 3.5.4. large scale computing

*MidOc implementation.* We use 3 implementations of MidOc, two being pure plpgsql (postgreSQL script language), and one python.

*Computing on very large dataset.* We successively order all the Paris and Vosges data sets with MidOc, using 20 parallel workers, with a plpgsql implementation. The ordering is successful on all patches, even in very challenging areas where there are big singularities in density, and many outliers. The total speed is about 100 millions points/hour, which we consider as extremely slow. We briefly analyse performances, and conclude that data IO limits the number of efficient workers to 10, and that most of the time is actually not spent on computing, but on getting the points and writing them back. We note that a huge speed up is easily reachable by directly reading and writing binary patches (and not array of points), and use more reasonable C code instead of plpgsql script.

### 3.5.5. LOD stream

As a proof of concept we stream points from the database to a browser

RC:3.5.5. citer iTowns

. For this experiment we only stream a given number of points per patch, which allows to accelerate greatly data loading.

limitation However we didn't use the LOD to perform a distance-dependend density : display patch with varying LOD given their distance to camera.

? ?(parler du stream de patch dans iTowns ?) ?? limitation = have to put the whole patch in memory, even when getting only few points. Conclusion : as is : interesting when bandwidth limited.

## 3.6. using the ordering by-product as a crude dimensionality descriptor

### 3.6.1. experiment summary

For each patch, we store the associated number of points chosen per level while computing MidOc ordering. We call this descriptor  $\text{points}_{\text{per level}}$ .

put results  
vosges, all  
classes

put results  
on vosges  
classes

put results  
on Paris, a  
classes

put results  
Paris, mai  
classes

This dimensionality descriptor alone cannot be used to perform sophisticated classification, because many semantically different objects have similar dimension (for instance, a piece of wall and of ground are dimensionally very similar, yet semantically very different).

We descriptors we use are ( $P$  : for Paris ,  $V$  : for Vosges : -  $\text{points}_{\text{per level}}$ ,  $\text{level1to4}(P+V)$  – average of intensity( $P+V$ ) – average of number of echo( $P+V$ ) – average of height regarding laser origin( $P$ ) – average  $Z(V)$  – patch height( $P$ ) – patch bounding box area( $P$ ) :

### 3.6.2. choosing classes in class hierarchy

Choosing which level of the class hierarchy uses depends on data set and applications.

*canonical classification.* In a canonical classification perspective, we have to strongly reduce the number of classes if we want to have significant results. However reducing the number of class (i.e use a higher level in the classes hierarchy) also means that classes are more heterogeneous.

We planned to automatically chose classes by clustering the confusion matrix (understood as an affinity matrix for the clustering method), but the wide difference in statistical weight and the variance of predictions for small classes deterred us from this approach.

Both data set are extremely unbalanced (factor 100 or more frequent). Thus our simple and direct Random Forest approach is ill suited for dealing with extremely small classes. (we would need to use some kind of cascading or appropriate one versus all learning).

For Vosges data set a short analysis convince us to use 3 classes : Forest, Land, and other, on this three classes, the Land class is statistically overweighted by the 2 others.

For the Paris data set, the representation of classes is very ill-balanced. Keeping only the class above 0.5 % reduce the number of classes to 12. The class under-represented have a strong variability in result. Also, some class cannot be properly defined without context (e.g. the side-walk, which is by definition the space between building and road, hence is defined contextually).

The main sources of error is that many patches contains mixes of classes, especially for class of small objects.

*balancing data set.* We abandoned the under-sampling approach because we think that in this case it can introduce significant bias. The urban objects are very diverse, and so each class is quite heterogeneous. Thus, by under-sampling, we artificially increase the variability of the results. We prefer the approach with weight, even if it performs poorly when the difference betteen classes is to large.

*classifying.* The computing time (10 fold, tree = 100) is between one and 20 minutes depending on the number of observation and the number of classes.

We note that most errors are on patches with mixed contents. We perform a analysis of error on Vosges data-set and we remark that the error seems to be significantly correlated to distance ot borders.

## **4. Illustrations**

**4.1. MidOc ordering**

**4.2. Crude dimensionality descriptor**



FIGURE 2:

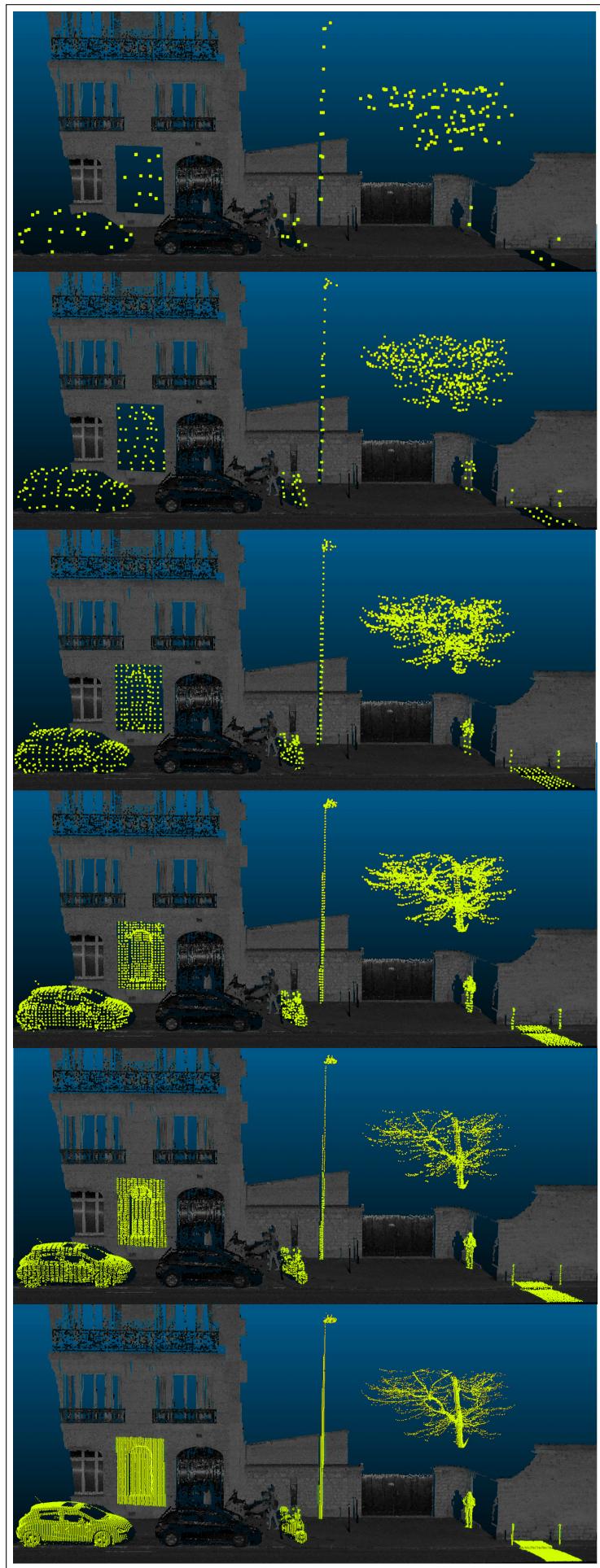


FIGURE 3: