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

Rémi Cura, etc.

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 point cloud 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 usage, having the raw, complete point cloud is unnecessary, or even damageable. Thus we deal with a simpler version of a problem that the G.I.S 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, cf figure 1) of this point cloud while preserving the geometric characteristic, without duplicating data ? The key to LOD approach is efficiency. Indeed LOD approaches sacrifices of a part of information in exchange of a massive reduction of data size. That's why a solution using LOD must by nature be efficient, or the information loss would be pointless.

1.2. Motivation

- Point cloud : becoming common 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 stereovision, and providing key data to virtual reality.
- Growing data set and Multi sources At such the size of data set are growing, as well as the number of dataset and their diversity.
- a now widely use data type 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)
- Much less focus on 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. discussion of bibliography

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 LOD , at the price of data duplication, and high sensibility to density variation.

Moreover, all the methods 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).

It is the classical and seemingly intractable trade-off between computing and storage : When pre-computing a data structure, it need to be stored extra. When doing the computing on the fly, the exact same operations many time.

1.5. contribution

This paper re-use and combine existing and well established methods with a focus on simplicity and efficiency. AS such, all the methods are tested on Billions scale point cloud, and are open source for sake of reproducibility test and improvements. We propose a simple method that enable portable, computation-free, geometrical Level Of Detail. Our first contribution is to propose to store the LOD information directly into the ordering of points rather than externally, avoiding any data duplication. 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.

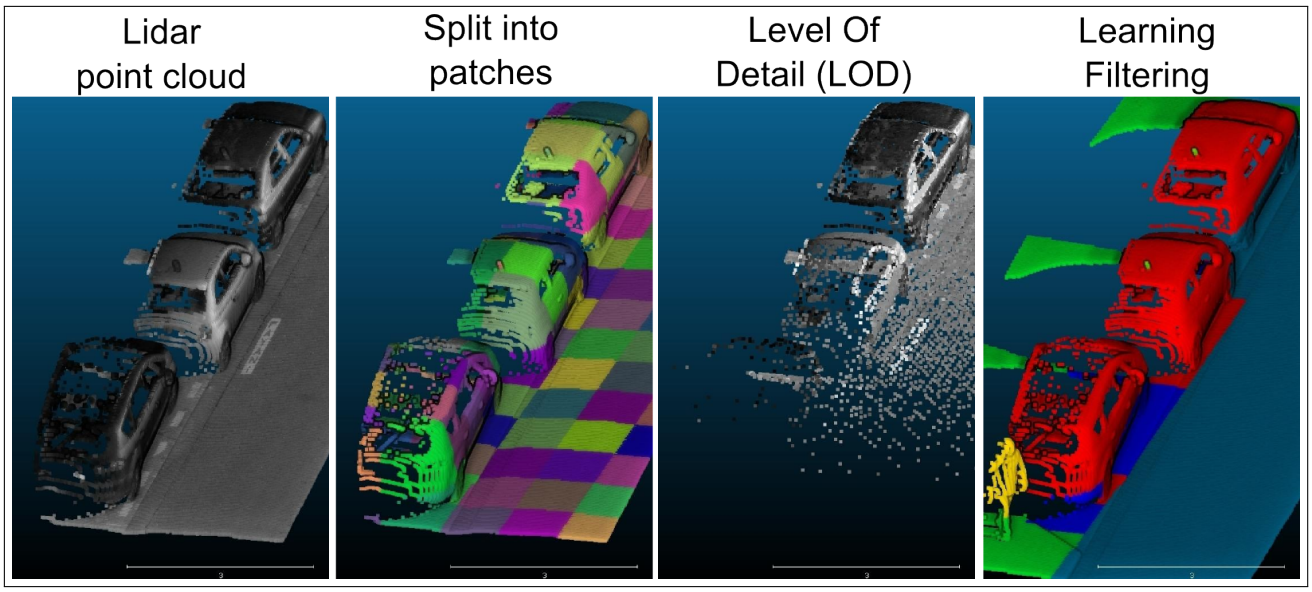


FIGURE 1: Data flow : starting from Lidar point cloud, we split it into patches, then reorder the patches to obtain free LOD (a gradient of LOD on this illustration). Lastly we use this ordering as feature for learning and efficient filtering

The second contribution is a simple way to order points in order to have a increasingly better geometric approximation of the point cloud when following this order.

The third contribution is to use the ordering construction by-product as a simple and free dimensionality descriptor. We demonstrate the interest of this descriptor in a Random forest classification (training-filtering) experiment.

1.6. plan of the article

The rest of this article is organized as follow : In the next section 2 we present the methods. In the result section 3 we give results and order of magnitude . We discuss it and possible limitations in the 4.

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. The method relies on ordering the points so that reading the point following this order is going to gradually increase details of the point cloud. We use a by-product of this method to perform efficient training and filtering with random forest.

This method has been designed to work with Lidar data set, but may be used on noisy Structure from Motion (SfM) point clouds after filtering. This method has been used inside our point cloud data management system which is centered on a Point Cloud Server (the article is being written, see

RC:2.1.0. put Point Cloud Server presentation

for a very detailed presentation of it).

We stress that Point Cloud Server work with patches, , which are groups of points obtained by cutting the original point cloud into regular pieces (1 meter wide cube for Paris data set,50 meter wide cube for Vosges dataset). A Patch is technically a subset of a point cloud, so also a point cloud. For generality we will use the term point cloud to describe both patch and global point cloud for the rest of this article, as 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 local dimensionality descriptor. We use that to perform extremely fast rough-filtering of massive point cloud.

2.2. Exploiting The Order Of Points

2.2.1. Principle

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

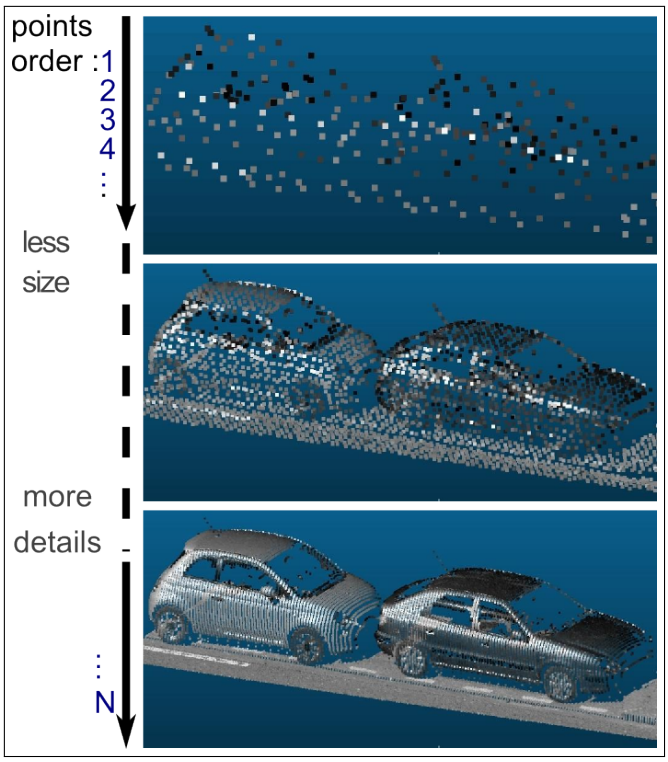


FIGURE 2:

Point clouds are physically stored as a list of points (compressed or not).

Once the points are written to the disk, 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, they are inherently ordered. In fact it would be quite hard to store the point in an 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 to order the points of the point cloud so that when reading the points from the beginning to the end, we get gradually a more accurate geometrical approximation of the point cloud.

We demonstrate it with an ordering presented in 2.3 , but we could use any ordering.

2.2.2. Conceptual Example

For a given 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, and reading another 16 points (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

This method has 3 majors advantages.

No on-line processing time. Except a pre-processing step to write the point cloud following a given ordering, each time the user wants 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 times at several levels and by several users simultaneously (for instance Point Cloud as a Service), no processing makes a big difference. (See this illustration 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 regarding the storage. This is an advantage on low level. First it saves disk space, it ensure that all the information is at the same place, which avoids to perform OS level commands like going trough directories, opening the structure file, etc. Having no data duplication is also a security in concurrent use. In the scenario where a point cloud and the associated data structure are separated, and both are accessed by concurrent users, it is possible that one user change the point cloud. The associated data structure is then updated, but during this laps of time it is possible for another user to get updated point cloud and the wrong associated data structure. 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 we avoid all precision-related issues raised by traditionnal subsample methods because the order doesn't change a bit of the existing points and their attributes, but only there 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 softwares do not change the points order inside a cloud. Even if a tool were to change the order, it is easy to add the ordering number as an attribute, which increases 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

Ordering the points necessitate an efficient method, because using LOD is already a trade-off between data information loss and data size reduction.

We note that having usable LOD is only possible 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 some parts of the cloud 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 a 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 (for instance, the octree subsampling is used in

RC:2.3.1. put reference to CloudCompare

). We name this ordering for clarity of this article, nonetheless we don't think we are the first to use it.

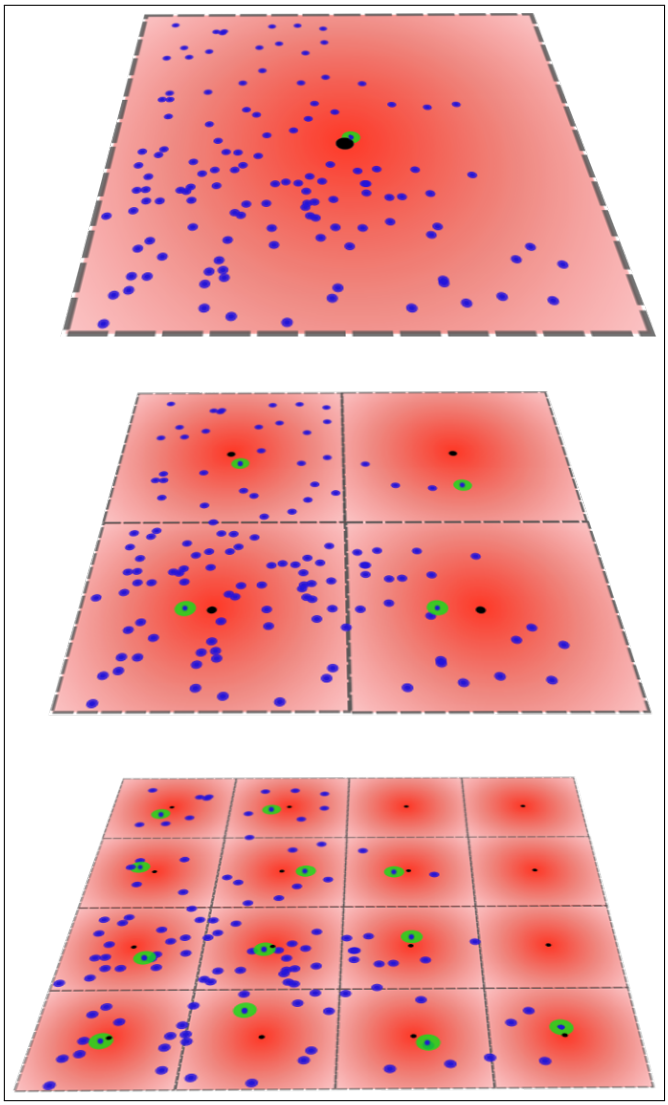


FIGURE 3:

Again the principle is very simple, and is to take for each level the points closest to centre of its octree voxel, if any.

We illustrate this principle on this figure in 2D (quad tree) for graphical comfort. The layer are level 0 to 2. For each layer we compute the closest point(large green point)

Data: A list of N points
Result: the MidOc ordering of the points for an octree of L depth
foreach $level$ *in [0..L-1]* **do**
  foreach $cell$ *of* $level$ *that is non empty* **do**
    compute $center$ of the $cell$;
    Choose the point in the $cell$ that is closest to the $center$, if it exists;
  end
end
order chosen points by level (coarse to fine), random (or hash or inverted Z curve);

Algorithm 1: MidOc principle

to the center of each cell (medium black point) using the squared euclidian distance (continuous red tone). The points picked on a level 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

This method has the same complexity as an octree construction. Similar strategies can be followed, pending on available resources.

The most straightforward implementation is streamed : The closest point to the center is stored in each octree cell. At the beginning the octree is empty. When receiving a point, traverse the octree update/creating cells and updating the closest point for each cell traversed. When the tree is computed, traverse it breadth-first to collect chosen points. The worst complexity is O(N*T), the entire octree must be stored in memory.

The simplest implementation use a recursive strategy. it only necessitate a method that given a cell and a list of points chose the point closest to the center of the cell, then split the cell and the list of points for the next level, and recursively calls itself on this subcells with the sublists.

We propose this kind of implementation in Python with extensive use of bit-level operations (cf 2.3.3) as proof of concept. The octree is not stored in memory, but it may prove difficult to parallelize (which is done at the data level anyway for our experiments).

2.3.3. efficiency and performance

Octree construction may be avoided by simply reading coordinates bitwise in a correctly centred/scaled point cloud. We centre a point cloud so that the lowest point of all dimension is $(0, 0, 0)$, and scale it so that the biggest dimension is in $[0, 1[$. The point cloud is then quantized into $[0..2**L-1]$ for each coordinate. The coordinates are now integers, and for each point, reading its coordinates bitwise left to right gives the position of the point in the octree for level of the bit read. This means performing that

this centring/scaling/quantization directly gives the octree. Moreover, further operations can be performed using bit arithmetic, which is extremely fast.

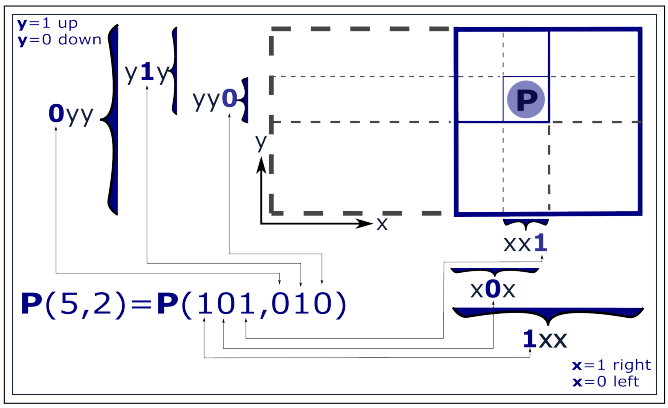


FIGURE 4:

Example. On this illustration the point P has coordinates $(5, 2)$ in a $[0, 2^3 - 1]^2$ system. Reading the coordinates as binary gives $(b'101', b'010')$. Thus we now that on the first level of a quad tree, P will be in the right (x=$b'1xx'$) bottom (y=$b'0yy'$) cell. For the next level, we divide the previous cell in 2, and read the next binary coordinate. P will be in the left (x=$b'x0x'$) up (y=$b'y1y'$) cell. There is no computing required, only bit mask and data reading.

2.3.4. advantages

Common, Simple and efficient. This method feels classical and is based on Octree. This makes it simple to implement, and possibly extremely memory and CPU efficient.

The complexity is $O(n_points * Levels)$ or less. It is very simple to implement. Octree construction has been commonly done on GPU for more than a decade.

Fixed density. This method can be used to guarantee an almost constant density for a 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 extrem variation in density (real world case illustrated in this illustration).

Generic. 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 DemantkÃ© 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 pieece 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 (Vmidoc=[N1,n2,n3,n4...]). 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 is 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 is it 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

RC:3.2.1. put reference to postgres presentation

). The key idea are 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 a 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 ,Est,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 tree 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, faÃ§ade, 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 langage), 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 spend 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 data base 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 perfom 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 $points_per_level$.

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 : - $points_per_level$, level 1 to 4 (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) - area of $patch_bounding_box$ (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.

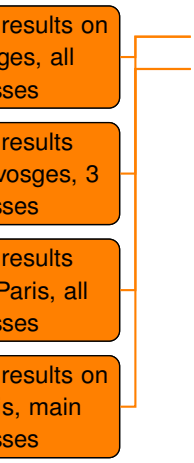
For the Paris data set, the representation of classes is very ill-balanced. Keeping only the class above 0.5 % reduce the number of classes to 12. The class underrepresented have a strong variability in result. Also, some class cannot be properly defined without context (e.g. the side-walk, which is 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 beteen 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 dataset and we remark that the error seems to be significately correlated to distance ot borders.



4. Discussion

4.1. ordering

limits : changing the order may be detrimental if we rely on it (acquisition topology)

4.2. MidOc

could use kdtree rather than octree if we don't want constant density minimal number of points per patch : issues with vegetation

5. Conclusion

RC:5.0.0. A ecrire !

6. Illustrations

6.1. MidOc ordering

RC:6.1.0. put figure about density variation

6.2. Crude dimensionality descriptor

FIGURE 5:

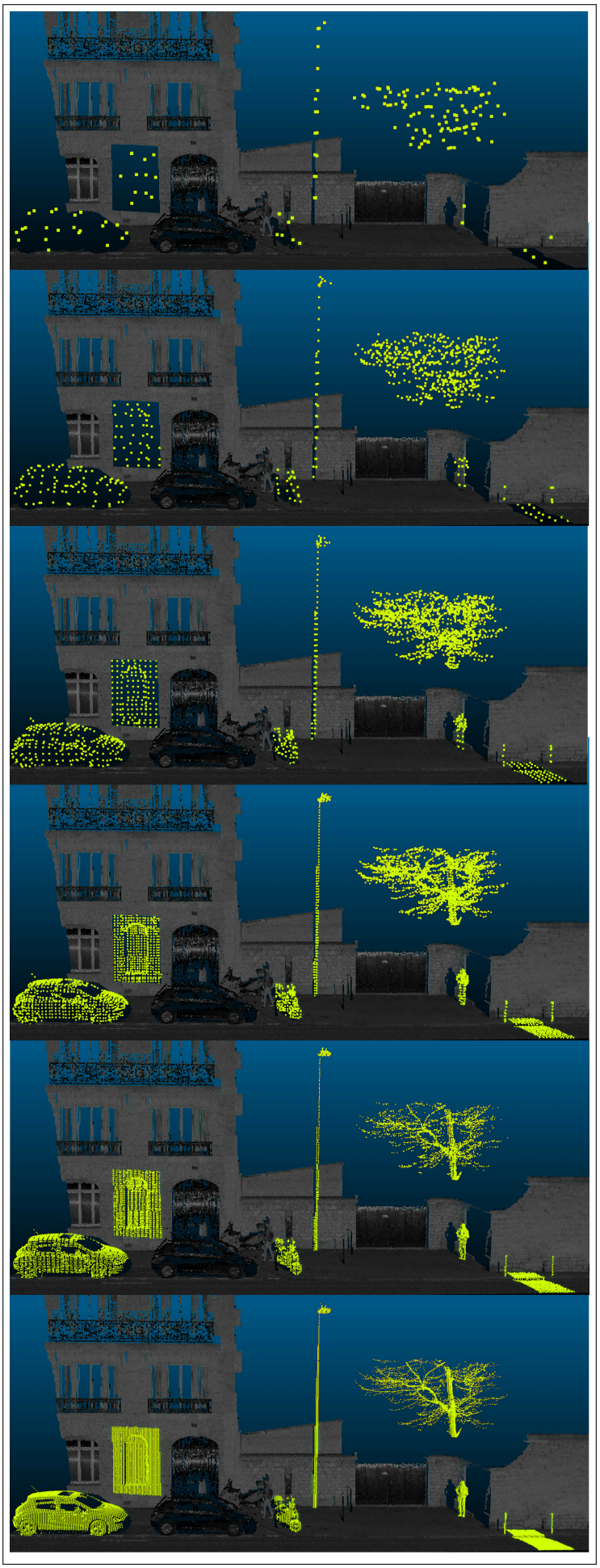


FIGURE 6: