# *TP 3 : Neighborhood descriptors*

NPM3*D* -*January 30*th, 2020
*mva.npm3d@gmail.com*

## *Objectives*

- *Compute normals on a point cloud*
- *Understand different types of local 3D descriptors*
- *[BONUS] Use those descriptors to classify a point cloud*

*The report should be a pdf containing the answers to the* **Questions** *and named "TPX_LASTNAME_Firstname.pdf". Your code should be in a zip file named "TPX_LASTNAME_Firstname.zip". You can do the report as a pair, just state both your names inside the report.*
*Send your code along with the report to the email address above. The object of the mail must be "[MVA_NPM3D] TPX LASTNAME Firstname". If you do the report as pair, the object of the mail must be "[MVA_NPM3D] TPX LASTNAME1 Firstname1 LASTNAME2 Firstname2".*

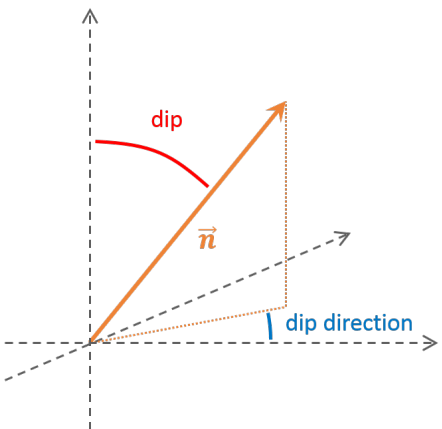
## *A. CloudCompare normals*

*In the first part of the practical session, you will use the "Normals" tool in CloudCompare software to visualize normals computed on a point cloud.*

1) *Load Lille_street_small.ply cloud in CloudCompare. Select the cloud and compute the normals: "Edit > Normals > Compute". You can choose "Plane" as the local surface model, and a radius of 50cm for the neighborhoods.*

2) *When a point cloud has normals, CloudCompare applies a rendering effect that depends on the normal orientations. If the cloud seems ugly, you can try to invert them: "Edit > Normals > Invert", or simply switch them off (in the property panel, uncheck the "Normals" box).*

3) *A better way to visualise normals is to convert them into scalar fields: "Edit > Normals > Convert to > Dip / Dip direction SFs". It will save dip and dip*



*direction as scalar fields. The dip (angle between the normal and the vertical direction) offers a simple visualisation to make the difference between flat and vertical planes. Select this scalar field in the properties panel.*

4) *You can estimate normals again with different radiuses of neighborhoods. To update the dip scalar field, convert your new normals again. Tip : If you have a slow computer, do not try a very big radius as it would take a very long time (2 meters should be enough).*

**Question 1 (2 points) : If you use a too small / too big radius, what is the effect on the normal estimation? Prove your point with screenshots of the dip scalar field.**

**Question 2 (1 point) : How would you choose the neighborhood scale for a good normal estimation?**

# *B. Local PCA*

***In this part of the practical session, you will compute principal component analysis (PCA) on neighborhoods, and use the principal components/vectors as local descriptors of the point cloud. If we call a point cloud*** $\mathcal{C} \subset \mathbb{R}^3$***, the definition of the spherical neighborhood of point*** $\mathrm{p}_0 \in \mathbb{R}^3$ ***in*** $\mathcal{C}$ ***with radius*** $r \in \mathbb{R}$ ***is:***

$$\mathcal{N}(\mathrm{p}_0, r) = \{\ \mathrm{p} \in \mathcal{C} \mid \|\mathrm{p} - \mathrm{p}_0\| \leqslant r\ \}$$

***Computing PCA on this neighborhood simply means finding the eigenvectors and the eigenvalues of its covariance matrix defined by:***

$$\mathrm{cov}(\mathcal{N}) = \frac{1}{|\mathcal{N}|} \sum_{\mathrm{p} \in \mathcal{N}} (\mathrm{p} - \bar{\mathrm{p}})(\mathrm{p} - \bar{\mathrm{p}})^T$$

***Where*** $\bar{\mathrm{p}}$ ***is the centroid of the neighborhood*** $\mathcal{N}$***.***

1) *In descriptors.py, implement the function local_PCA returning the eigenvalues and the eigenvectors of the covariance matrix.*
*Tip 1: This function is very similar to the function best_rigid_transform (TP2). You can see the similarity between the matrix H and the covariance matrix here.*
*Tip 2: you can use the function np.linalg.eigh to find the eigenvalues (in ascending order) and the corresponding eigenvectors .*

2) *To verify your implementation, apply your local PCA to Lille_street_small.ply cloud. You will find the good values as a comment in descriptors.py "main".*

*You can see a scheme of a PCA on 2D points on figure 1. The first eigenvector measures the direction in which the points are the most spread out. The first eigenvalue measures that spread (explicitly, it is the variance of the distribution of points in that direction). The following eigenvectors iteratively measure the directions in which the points are the most spread out, with the condition to be perpendicular to all previous eigenvectors. As a convention we will name the eigenvalues* $\lambda_1$, $\lambda_2$ *and* $\lambda_3$ *such that* $\lambda_1 \geq \lambda_2 \geq \lambda_3$. *The corresponding eigenvectors will be named* $e_1$, $e_2$ *and* $e_3$. *Be careful with your implementation as the function np.linalg.eigh returns the eigenvalues and eigenvectors in the ascending order, which is the inverse of the convention.*

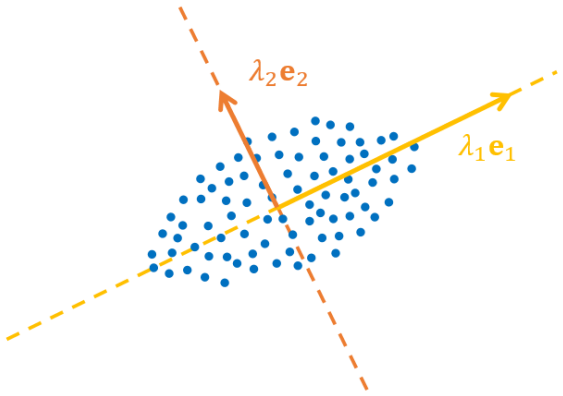



*Figure 1 : PCA illustration in 2D*

3) *If you consider a group of points describing a small part of a 3D surface, one of the three eigenvectors is expected to be the normal of the surface. We will use PCA to compute normals and compare them to CloudCompare normals.*

4) *In descriptors.py, implement the function neighborhood_PCA returning the eigenvalues and eigenvectors of each query point neighborhood in the cloud. This function should take a neighborhood radius as parameter, and the returned values should be stored in numpy arrays with the shapes [N x 3] and [N x 3 x 3] Tip : To reduce computation time, you can search for all neighbors at one with the query radius of KD Tree structures. Then loop over neighbors lists to compute local PCAs.*

5) *Use the result of this function to obtain normals on Lille_street_small.ply cloud with the radius of your choice and save the cloud. Tip : For the function write_ply to work, your normals should be in a [N x 3] np*

*array, and you should add the three new field names : "nx", "ny", "nz"*

**Question 3 (2 point) : Show a screenshot of your normals converted as "Dip" scalar field in CloudCompare.**

**Question 4 (1 point) : How could you measure the quality of your normal estimation with the eigenvalues?**

6) ***As explained earlier, "Dip" scalar field represents the angle between the normals and the vertical direction. We can use this angle to compute a simple descriptor that we could name "verticality". It should take values in [0,1] and be close to 1 if the normal is vertical and close to 0 if the normal is horizontal.***

$$verticality = \left| 1 - \frac{2\text{angle}(\mathbf{n}, \mathbf{e}_z)}{\pi} \right| = \frac{2\arcsin(|\langle \mathbf{n}, \mathbf{e}_z \rangle|)}{\pi}$$

***The normal verticality is not the only descriptor than you can get with a PCA. Here are three other simple descriptors based on eigenvalues :***

$$linearity = 1 - \frac{\lambda_2}{\lambda_1} planarity = \frac{\lambda_2 - \lambda_3}{\lambda_1} \qquad sphericity = \frac{\lambda_3}{\lambda_1}$$

7) ***In descriptors.py, implement the function compute_features returning the 4 features for all query points in the cloud. This function will use neighborhood_PCA and thus have a neighborhood radius parameter.***
***Tip 1 : you can use simple operations and more complex function like np.arcsin or np.abs directly on entire arrays. It will be faster than using "for" loops.***
***Tip 2 : if a point has no neighbors, the eigenvalues associated to it will all be equal to zeros. Add a small epsilon to the fractions denominators to avoid errors.***

8) ***Compute those 4 features on Lille_street_small.ply cloud with the radius of your choice and save the cloud.***
***Tip : the function write_ply can save as many scalar fields as you want with the points.***

**Question 5 (4 points) : Show screenshots of the 4 features as scalar fields of the cloud. Can you explain briefly the names of the 3 last features considering their definition with eigenvalues?**

*(BONUS)*

# *C. Mini-Challenge : point classification*

***The last part of this practical session is a little machine learning challenge. Now that you know how to compute 3D descriptors, you can use them to classify a point cloud.***

## *Mini-Challenge Rewards*

***In the report of this practical session, or in a separate report, explain your classification method briefly. If you tried several methods, you can show your cross validation results to support your choices. Bonus points will be awarded to the best scores but also for original ideas.***

***Point cloud classification can be a subject for the final project, if you are willing to go further, and implement more complex algorithm, you can ask more information by email.***

***The best/most original methods will be highlighted during the seminar on February 28, 2019.***

## *Instructions*

***The mini-challenge will be hosted on the npm3d website as a temporary benchmark. First create an account with "MVA" as affiliation. All necessary instructions are available on the "Benchmark MVA" page.***

***To be able to submit results, you have to login and create a new method in your dashboard. Do not forget to put your name in your method name or description.***

***There is a 3 hours limit between each submission, and you are not allowed to create multiple methods to overcome this limitation. Cheaters will be harshly penalized.***

## *Data*

***The data is a subpart of the bigger Paris-Lille-3D benchmark, coarsely subsampled with a 5cm grid. This drastically reduces the number of points, allowing you to still code your algorithm in python.***

***The training data consists of three clouds : MiniLille1.ply, MiniLille2.ply and MiniParis1.ply. Compared to the original Paris-Lille-3D dataset, the annotation comprises only 7 classes:***

0. ***Unclassified***
1. ***Ground***
2. ***Building***
3. ***Poles***
4. ***Pedestrians***
5. ***Cars***
6. ***Vegetation***

***You should not use the first class (Unclassified) during training. It will be ignored in the test. You can use a cross validation strategy on those three clouds to find the best parameters, features, or training/test strategy.***

***The goal is to classify every point of the cloud MiniDijon9.ply. The predicted labels should be should be stored as a text file named MiniDijon9.txt. You can use the function np.savetxt to write this file.***

***If you managed to implement an efficient algorithm (in C++ for example), you are allowed to use the bigger Paris-Lille-3D training set, but you will have to manage the different classes, and point densities. If you use this bigger dataset or any other training data than the one given for the challenge, you have to mention it by checking "Additional training data" in your method.***

## *Tips*

***In classification.py, a basic strategy is already implemented. As a training set, we chose to use the same number of point per class. It should be better than taking random points because the classes are heavily unbalanced. Then, in each of the three training clouds, we chose 500 points per class if possible. For every training point, we compute the four basic features seen in part B, with a fixed radius of 50cm. Eventually we train a random forest classifier on those features.***

***There are several ways to obtain better results. Here are some leads :***

- ***The features used here are very basic. You can find more of them by looking at the literature on point cloud classification. This article should be a good start.***
- ***The scale parameter was not optimized. You can also imagine using multiple scales instead of one, however, you can be limited by the computation time. Unless***

*you find a way to reduce it... If you want to go that way, this article should help you.*

- *Although you will have trouble finding a better classifier than random forests, the training strategy could be better than a random sampling. Try to search "active learning", it can be used to find the best training points, instead of picking them randomly.*
- *If you are willing to use Deep Learning, it is possible to use volumetric convolutional neural networks, or more specific architectures to classify 3D points.*