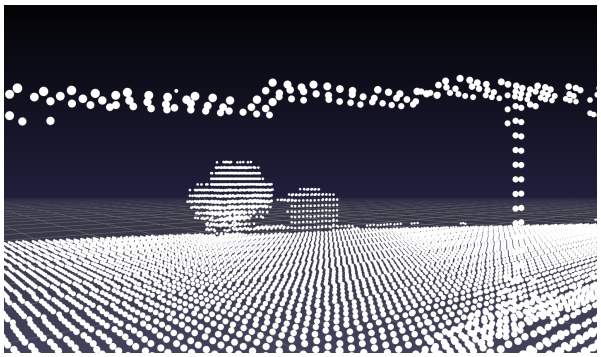
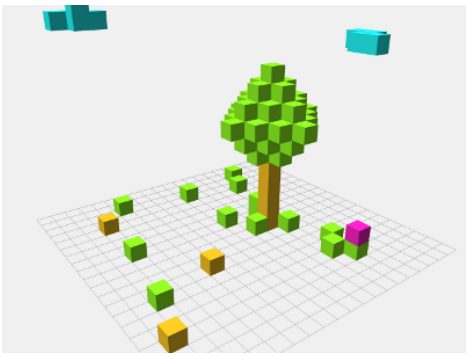
One of the promising solutions of automating classification of point clouds is comparing how much pointclouds that exists in specific voxels are similar to Euclidian bodies such as lines or planes.

Think about voxels

Example of pointcloud



Example of voxels



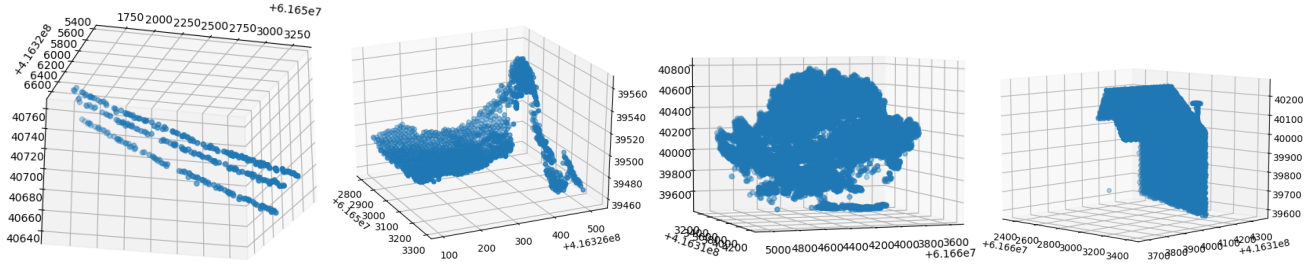
Let's look at these two pictures. The first one demonstrates enormous amount of points captured from helicopter while inspecting high voltage powerlines using lidars. The second one is assumed to be beginning of an answer to the most important problem: how to recognise classes of these points such as ground, trees, houses, wires, poles and so on. First picture display 3D points of those cubic centimeters where they have been fixed. Every sector of a lenght of just a few kilometers can contain up to 10 - 100 millions of points. To perform algorithms of visual recognition, a lot of computational power is needed. The second one reveals the way of look that is pretty enough for elements of landscape to be recognised.

Voxel is a volumetric rectangle. It is enough to store only those rectangles of landscape that contains at least one point. That's how we define it.

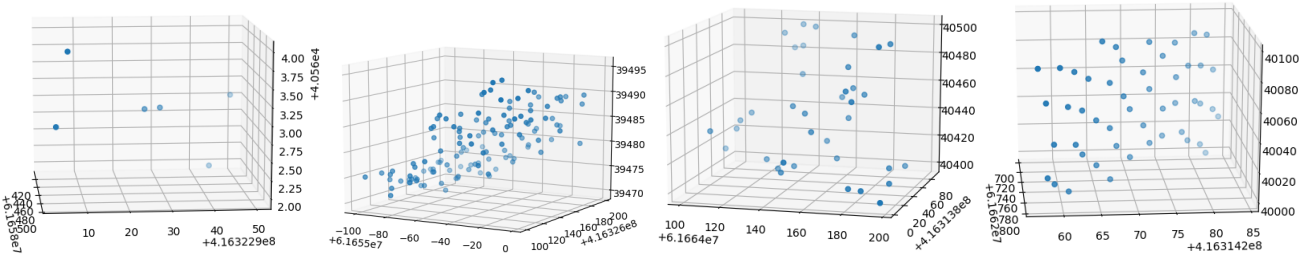
Looking inside data

At the first stages of this project, the main attention has been paid for increasing flexibility of accessing and managing data that we get from pointcloud.

Let's take a look at a few samples to see the landscapes of what is expected to be inside the voxels:

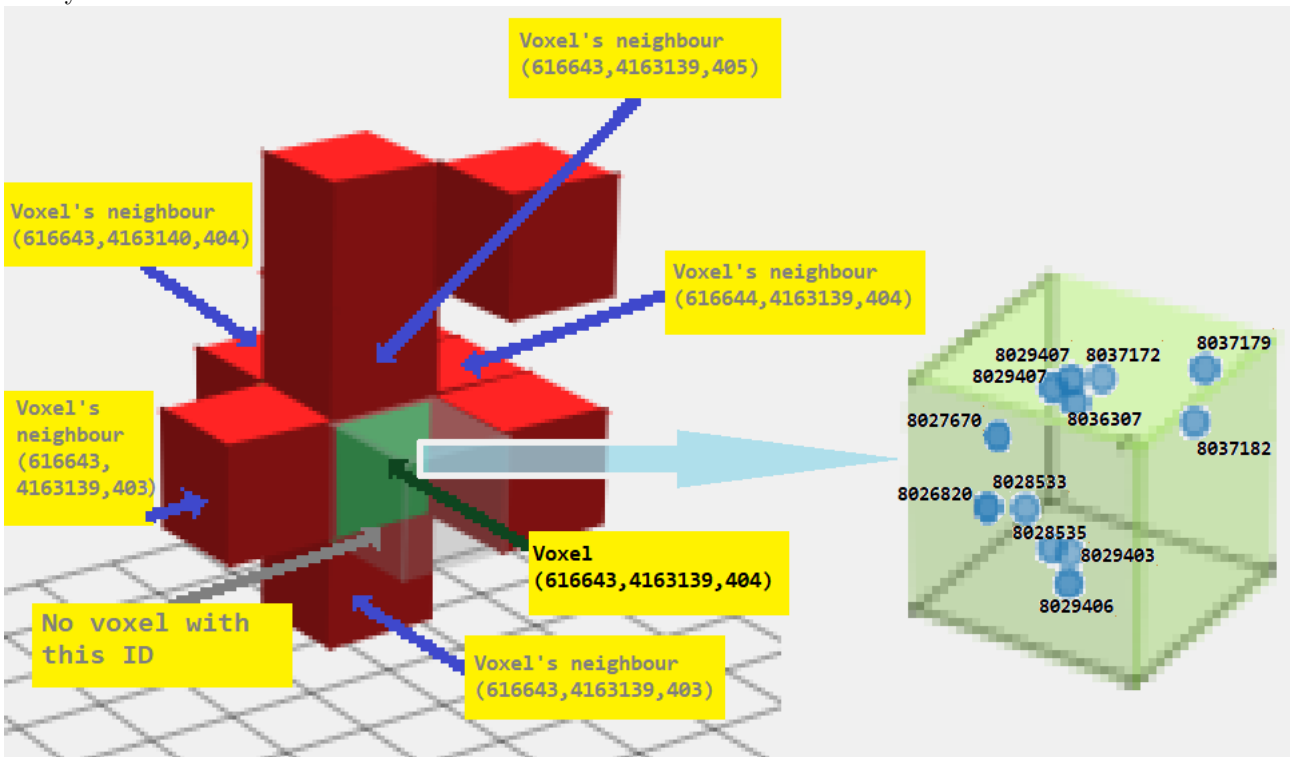


All of these samples illustrates the case of high (up to 10 - 20 meters) volume voxels. Now let's move to standard split which is the case of standard $1m \times 1m \times 1m$ voxels used in algorithm:



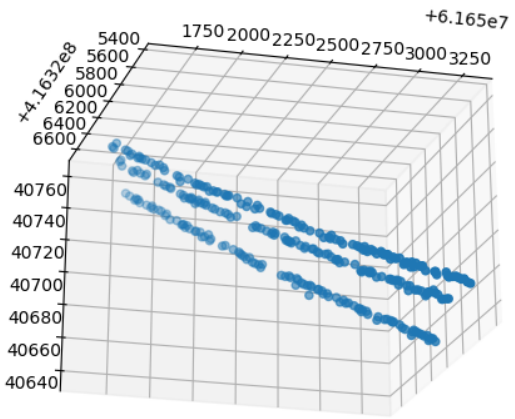
After some research it has been decided to store these points in array of dictionary type that supports 3D indexation. For example, one of terms of this dictionary has index (616643, 4163139, 404). This is a specific id of voxel bringing info about its position in space. Advantages of dictionaries are spectacular: unlike in the case of simple list used in Python (with huge loss of time, at least), one can check easily if six neighbours of this voxel, especially (616643, 4163139, 403), (616643, 4163139, 405), (616643, 4163138, 404), (616643, 4163140, 404), (616642, 4163139, 404), (616644, 4163139, 404), are still in a pointcloud. Under assumption that voxels

are connected if they are neighbours, this scenario unlocks a promising opportunity to apply knowledge of graph theory later if it's needed.



If we check voxel that has, say, index (616643, 4163139, 404), we will find a heap of indexes such as 8026820, 8027670, 8028533, 8028535, 8029403, 8029406, 8029407, ..., 8036307, 8037172. They tells us ids of the points that are inside this voxel in respect with the order of points recorded in .las file sent from lidar.

Classifying wires



After storing, retrieving and visualising data successfully, we can move to the easiest part: classification of wires. Solution should be splitted into smaller parts:

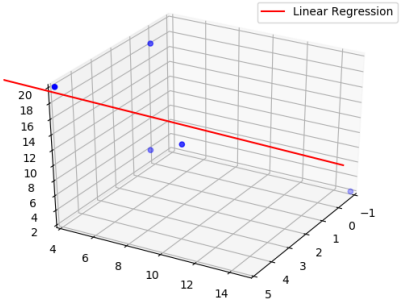
- How can 3D regression line can be found for a given voxel?
- How can we calculate distances from each point from this line?
- How to decide whether there exists regression line or not?
- What if there are not the only one?

Finding regression line

This question and its pythonic solution can be found here. Note that a solution is based on mathematical deduction shared by mathematicians. I didn't investigate the meanings of variables used in a statement therefore it is sufficient to access the output of algorithm only. I refactored the code proposed in StackOverflow to make algorithm more clear and usable in this project:

```python
class MathPoint:
    def __init__(self, points):
        self.points = points
        self.center = None #mass center of points
        self.centered = None  # translated version of points so that center fits origin
        self.cov = None  # returns matrix of covariance

    def calculate_center(self):
        if self.center is None:
            self.center = np.mean(self.points, axis=0)

    def get_center(self):
        return self.center

    def calculate_centered(self):
        self.calculate_center()
        if self.centered is None:
            self.centered = self.points - self.center

    def get_centered(self):
        return self.centered

    def calculate_cov(self):
        self.calculate_centered()
        if self.cov is None:
            self.cov = self.points.T.dot(self.centered) / self.points.shape[0]

    def plot_regression(self):
        self.calculate_cov()
        xmin, ymin, zmin = self.points.min(axis=0)
        xmax, ymax, zmax = self.points.max(axis=0)

        plt.figure()
        ax = plt.axes(projection='3d')
        plt.xlim(xmin, xmax)
        plt.ylim(ymin, ymax)
        ax.set_zlim(zmin, zmax)
        ax.scatter(self.points[:, 0], self.points[:, 1], self.points[:, 2], c='blue')
        linearReg = self.center + self.cov[:, 0] * np.vstack([xmin, xmax])
        # alternatively, use:
        # linearReg = self.center + self.cov[:, 1] * np.vstack([ymin, ymax])
        # linearReg = self.center + self.cov[:, 2] * np.vstack([zmin, zmax])
        ax.plot(linearReg[:, 0], linearReg[:, 1], linearReg[:, 2], 'r', label='Linear Regression')
        plt.legend()
        plt.show()
```

Result of algorithm



Finding distances from points to regression line

In order to use any further calculation we need to „combine" output of the above program with input of theoretical stuff accessible. At first, it was expected to continue with deriving algorithm from theory. Description of an algorithm has been done but not tested. Hence it's moved to **Extra theory**. Alternative way is to use fitting a line in 3D algorithm based on Singular Value Decomposition.

Regression line can't be identified if there is smth more than just wire (e.g. noise). This article offers solution:

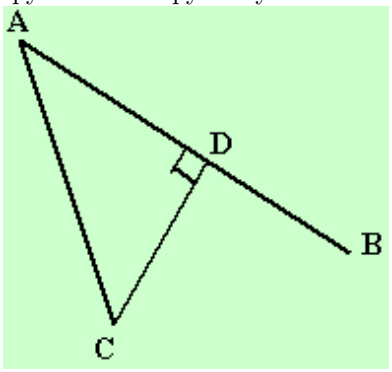
Roofs

Fitting regression line is evident right now. Let's move to a tutorial on the total least squares method for fitting a straight line and a plane (the last page).

Fitting a plane to many points in 3D

Extra theory

Assume that calculations above are proceeded for an object `mathpoint = Mathpoint(points)` where `points` is a pythonic numpy array.



Borrowing from key example of distance from a point to a line, denote
$\overrightarrow{AC} = (a_1, a_2, a_3)$, $\overrightarrow{AB} = (b_1, b_2, b_3)$. Then

$$\overrightarrow{AC} \times \overrightarrow{AB} = \begin{vmatrix} \mathrm{i} & \mathrm{j} & \mathrm{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix} = \mathrm{i} \begin{vmatrix} a_2 & a_3 \\ b_2 & b_3 \end{vmatrix} - \mathrm{j} \begin{vmatrix} a_1 & a_3 \\ b_1 & b_3 \end{vmatrix} + \mathrm{k} \begin{vmatrix} a_1 & a_2 \\ b_1 & b_2 \end{vmatrix};$$

A distance CD follows from $\dfrac{|\overrightarrow{AC} \times \overrightarrow{AB}|}{|AB|} = \dfrac{\sqrt{\begin{vmatrix} a_2 & a_3 \\ b_2 & b_3 \end{vmatrix}^2 + \begin{vmatrix} a_1 & a_3 \\ b_1 & b_3 \end{vmatrix}^2 + \begin{vmatrix} a_1 & a_2 \\ b_1 & b_2 \end{vmatrix}^2}}{\sqrt{b_1^2 + b_2^2 + b_3^2}} = a \cdot a - \left(\dfrac{a \cdot b}{b \cdot b}\right)^2$

In a script above, a regression line of 3D pointcloud is given by $f(x) = P + Qx$ where

$$\begin{cases} P = \texttt{mathpoint.center} \\ Q = \texttt{mathpoint.cov[:, 0]} \\ x \in \mathbb{R} \end{cases}$$

and C is any point taken from `mathpoint.points`. It's also convenient to introduce variables

```
xmin, ymin, zmin = mathpoint.points.min(axis=0)
xmax, ymax, zmax = mathpoint.points.max(axis=0)
```

to express points A, B, C in terms of script used:

$$\begin{cases} A = f(\texttt{xmin}) \\ B = f(\texttt{xmax}) \\ C = (t_1, t_2, t_3) \end{cases}.$$

Now it's possible to calculate coordinates for both vectors: $\overrightarrow{AC} = C - A = C - P - Q\texttt{xmin} = \texttt{self.centered} - Q\texttt{xmin}$

$\overrightarrow{AB} = B - A = P + Q\texttt{xmax} - P - Q\texttt{xmin} = Q(\texttt{xmax - xmin})$.

It can be shown that scalar factors next to Q doesn't effect a value of $\frac{|\overrightarrow{AC} \times \overrightarrow{AB}|}{|AB|}$. Calculations are simplified then to:

$$\begin{cases} AC = \texttt{mathpoint.centered - mathpoint.cov[:, 0]} \\ AB = \texttt{mathpoint.cov[:, 0]} \end{cases}$$

A vector $T = \overrightarrow{AC} \times \overrightarrow{AB}$ is calculated later using `np.cross` method and the result $\dfrac{|\overrightarrow{X}|}{|\overrightarrow{AB}|}$ is easy to compute.

Bug fixes

$\overrightarrow{AB}$ shouldn't be zero vector. When is it possible? $\overrightarrow{AB}$ is extracted from `mathpoints.cov[:,0]`

- Cases of one points should be ignored.
- Cases of two points: (a_1, a_2, a_3) and (b_1, b_2, b_3):

  `mathpoint.cov` $= \begin{pmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{pmatrix}$ `.dot` $\begin{pmatrix} a_1 - c & b_1 - c \\ a_2 - c & b_2 - c \\ a_3 - c & b_3 - c \end{pmatrix}$ where $c =$ `mathpoints.center` (`.dot` method multiplies matrices in a standard way)

1 KD trees

Look at explanation about storing this data. Assuming you have `coords` of your pointcloud, it's possible to tell 2 closest points for each point in your coords according to this:

```
from sklearn.neighbors import KDTree
tree = KDTree(coords)
nearest_dist, nearest_ind = tree.query(coords, k=3)  # family of k=3 nearest neighbors
```

Additionally, following this, classification of connected components is also accesible:

```
from sklearn import neighbors
from scipy.sparse import csgraph
adj = neighbors.kneighbors_graph(X, 3)
n_components, labels = csgraph.connected_components(adj)
```

but it has some small problems
But look, there is very good question