From Points to Prints

Monthly Presentation (3)

Alexandre Bry

April 20, 2026

TU Delft

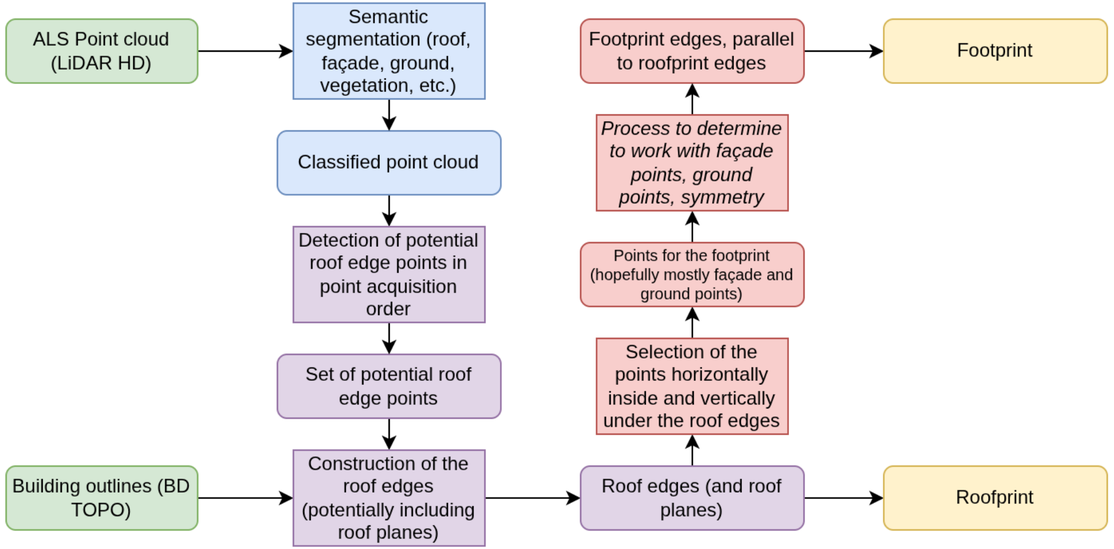
IGN

Outline

Outline

Context

Planned pipeline



General structure

General structure of the method:

- **Identifications of points** to deform the polygons on.
- **Creation of a set of candidate configurations**:
  - By translating each edge perpendicularly to itself.
  - The other edges are translated if necessary to keep the same topology.
- **Definition of a criterion to evaluate the quality of each configuration**, based on:
  - The position of the points.
  - Weights computed for the points.
  - The initial configuration of the edges.

Point cloud topology

Point cloud topology

Pulse

A single emission of the LiDAR sensor, which may result in **zero, one or more echoes** (points) depending on the number of surfaces the pulse hits.

Scan line

A **set of pulses** emitted during one rotation of the LiDAR scanner.

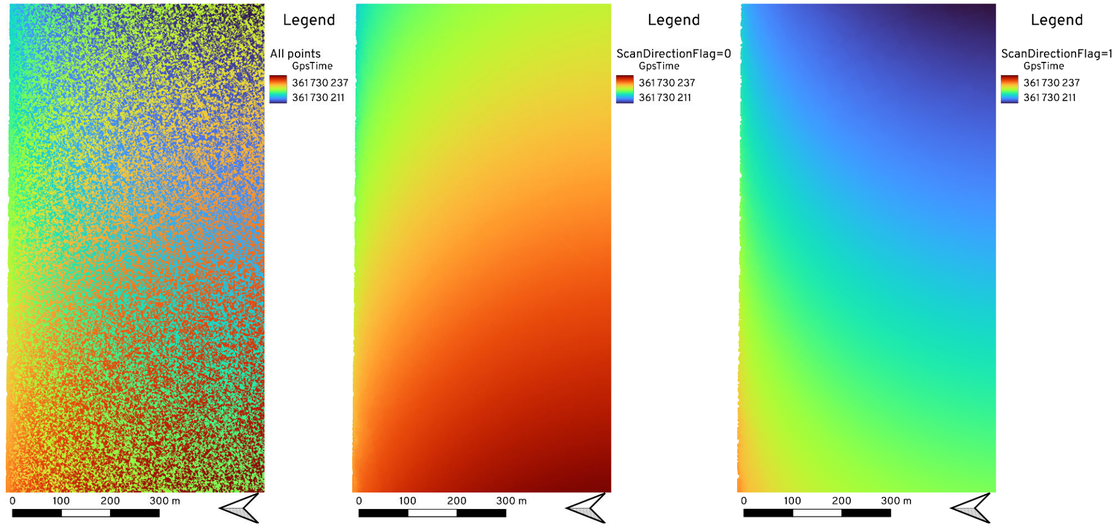
Flight strip

A **set of scan lines** collected along one pass of the aircraft over the ground.

- This creates a **hierarchy** in the point cloud, with points belonging to pulses, pulses belonging to scan lines, and scan lines belonging to flight strips.
- With multiple flight strips in the same area, this creates a sort of **irregular 3D structure**:
  - 1st dimension: the flight strip
  - 2nd dimension: the scan line
  - 3rd dimension: the pulse

It is then possible to **navigate** in this structure along any of these dimensions.

Point cloud topology



(a) All points of one flight strip. (b) Front scan line. (c) Back scan line.

Figure 1: Visualization of the topology of the point cloud, with points coloured by their GPS Time.

Due to the ellipsoidal scanning pattern of the LiDAR scanner, the GPS Time values follow a **curved pattern**, and are mixed up when looking at the whole flight strip. However, using the **Scan Direction Flag** to separate front and back scan lines shows the distribution.

Flight strips trajectories

- Trajectories are computed using **multi-echo pulses** (code written by **Wu Teng**)
- Necessity to **reconcatenate the different parts of each flight strip** for better precision (not done yet)

> This method allows to use the trajectory **without relying on its availability**.

- I use code written by Wu Teng to **retrieve the trajectory** of the scanning vehicle, using **multi-echo pulses**
- The precision of the method increases with the number of multi-echo pulses, meaning that it is necessary to **reconcatenate the different parts of each flight strip** (scattered between tiles) in order to get a good estimation of the trajectory.
- This method allows to use the trajectory **without relying on its availability**, and therefore without adding new requirements on the input data.

Edge points

We use the **height differences between consecutive points** on the same scan line to identify points that are likely to be on the edges of roofs (more details in the slides of the previous presentation).

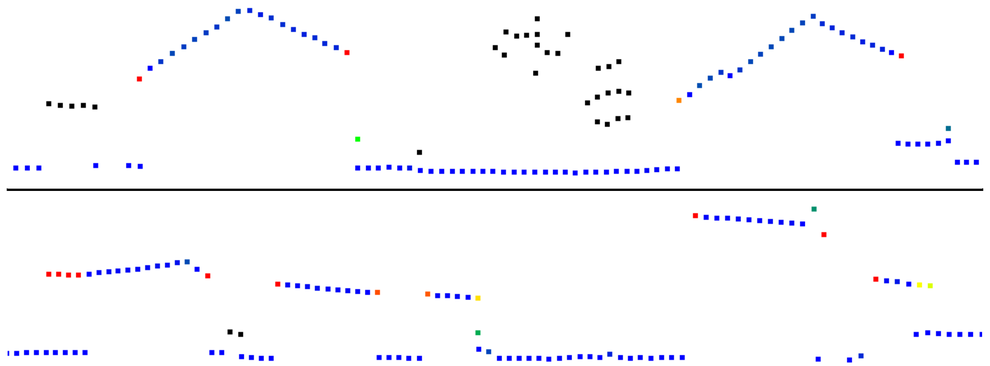


Figure 2: Two examples of the height differences obtained on a scan line. Black points are vegetation points, and for the other points the color represents the value of Δh, with blue-green-yellow-red from low to high values.

- Points at the edges of what looks like building roofs have **high values of** Δh
- Other points get **low values** of Δh
- Points classified as **vegetation** would have gotten high values if not excluded
- These scan lines are actually **curved** when looked at from above, but this is not a problem as the **angle is very small** between consecutive pulses

Candidate roofprint configurations

Constraints over the movements

We modify the polygons by applying the following rules:

- **Never rotate an edge.**
- **Never flip an edge.**
- **Move overlapping edges together.**

This ensures that we **preserve some topology** while keeping a **lot of freedom** for the edges to move. However, this **does not ensure that we preserve the same topology**, as can be seen in the examples in the next slides.

In practice, we only ensure that:

- At the level of one polygon, for each edge:
  - The edge is never flipped.
  - Therefore its two neighbours do not intersect.
- At the level of multiple buildings:
  - Two initially overlapping edges will remain collinear (not necessarily overlapping).

Constraints over the movements

One point becomes two

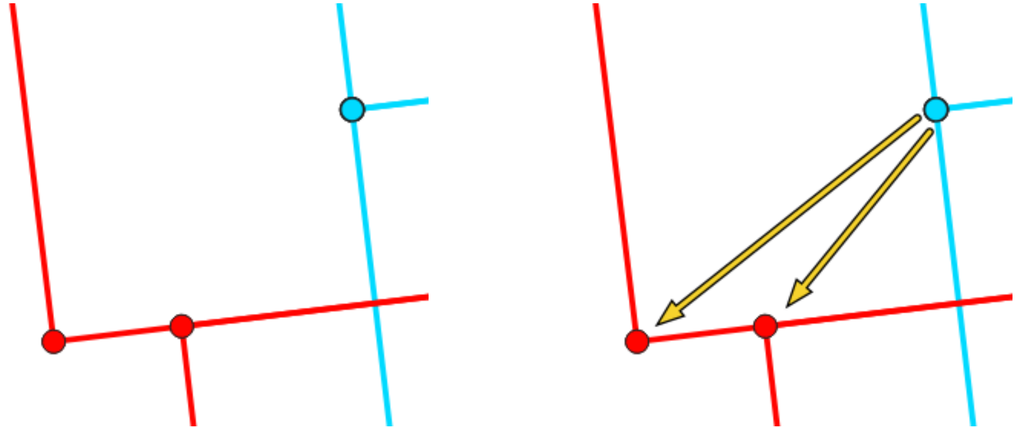


Figure 3: Two vertices at the exact same position become two distinct vertices at the boundary between two buildings.

- Simplest example of topology breaking: the shared edge between two buildings was moved by the same extent for both, but **the neighbouring edge of each building was moved by a different extent**, resulting in the shared vertex being split into two distinct vertices.
- Fixing this is **theoretically simple**: it requires only to authorize more than two edges per vertex, and therefore more than two neighbours per edge. However, it may not always be easy to identify which edges from different buildings are connected, especially when there is no shared vertex. This is the case with the final situation in red for the vertex on the right: it is part of one building but not of the other.

Constraints over the movements

One point becomes two

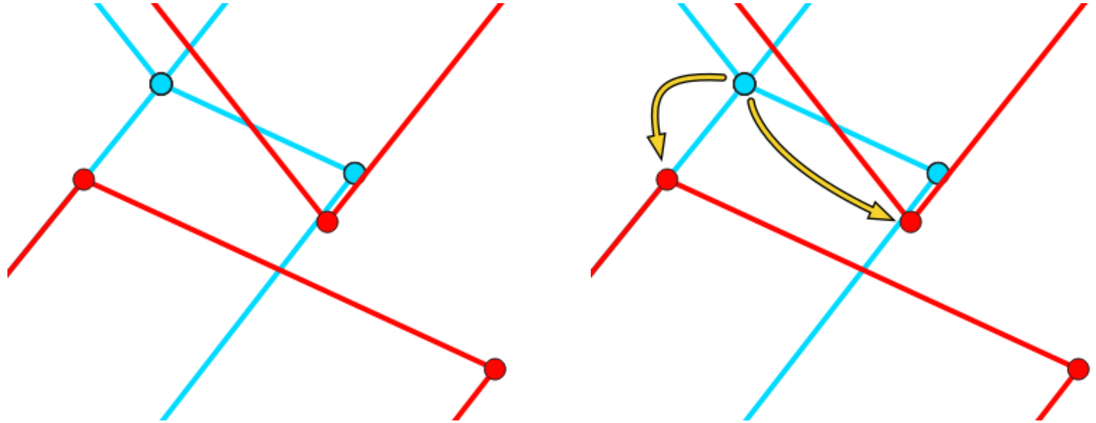
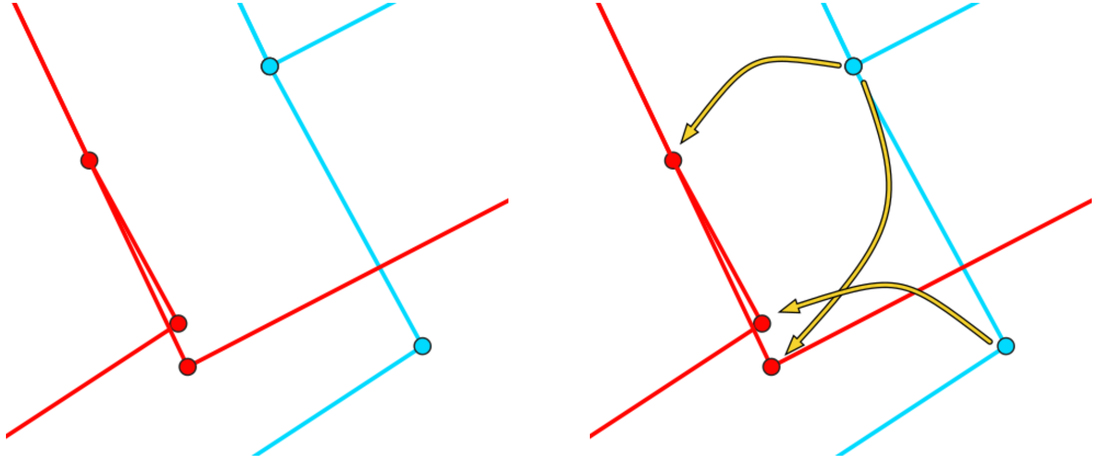


Figure 4: Two vertices at the exact same position become two distinct vertices at the boundary between two buildings.

- Similar situation except there was not even a shared edge between the two buildings, only a shared vertex.
- The conclusion about the solution is the same, if we manage to merge the two vertices into one, we can prevent this issue.

Constraints over the movements

One point becomes two and Two buildings intersect



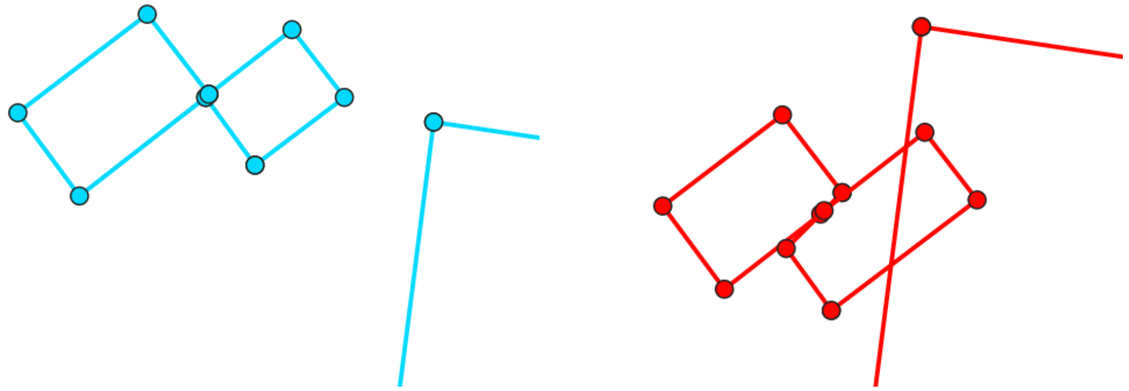
(a) Before (b) After

Figure 5: Two buildings that were initially separate end up intersecting after the deformation.

- Here, the issue comes from the fact that the shared vertex is considered separately for each building. Therefore, moving the edge of the building on the right does not move the edge of the building on the left.
- Like the previous examples, this could be fixed by **rebuilding the actual topology including shared vertices**, instead of only polygons and shared edges.

Constraints over the movements

Two buildings intersect



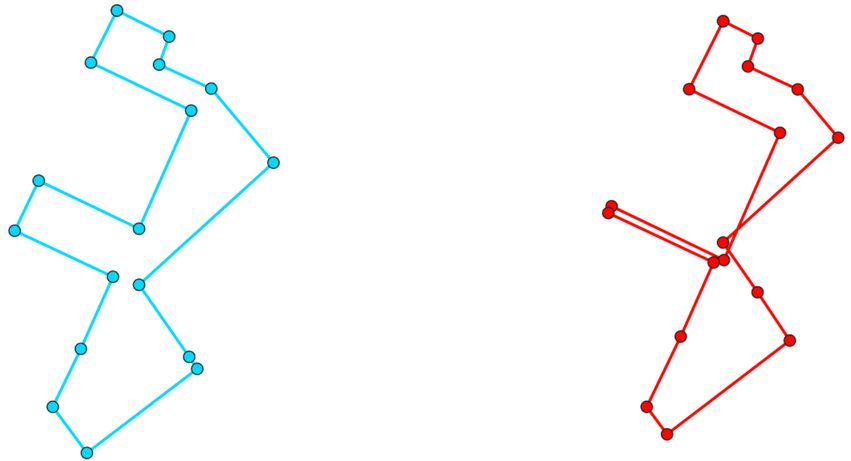
(a) Before (b) After

Figure 6: Two buildings that were initially separate end up intersecting after the deformation.

- Here, we have a relatively simple situation where two non-overlapping buildings end up intersecting after the deformation.
- Fixing this would require potentially complex operations and structures to **efficiently check for intersections between all pairs of edges of different buildings** at each iteration. This also means **increasing the number of edges to consider together for optimization**, as any two buildings that are close enough to potentially intersect would need to be optimized together.

Constraints over the movements

One building self-intersects



(a) Before

(b) After

Figure 7: Two opposite sides of the same building that were initially close end up intersecting after the deformation.

- Here we can see that **no edge was flipped**, but the left and the right side of the polygon were pulled closer together, resulting in a self-intersection.
- This result is only possible because the polygon is **not convex**.
- One way to prevent this would be to check not only for edge flips, but also for **intersections between non-neighbour edges** of the same polygon, but this would be much more costly to check.

General idea

We focus on **edges one by one**, except for overlapping edges which are treated together. Edges are treated as **directed lines**, which can be translated in any direction, but not rotated.

Therefore, to satisfy the constraints described previously, there is only one property $\mathcal{P}(l)$ that we need to preserve for each line l, regarding its previous line l^- and next line l^+:

> ***The intersection between l and l^- should occur before the intersection between l and l^+.***

This property is **equivalent to not flipping the edge l**.

Moreover, we know that if we shift l, l^- and l^+ by the **same extent in the same direction**, this property will still hold for l.

The ideas behind this slide is actually quite simple:

- We want to **move edges** without rotating them, so it is easier to simply consider them as **lines** that we shift
- By considering lines, the definition of points depends on the **intersection with the previous and next lines**, therefore not flipping edges becomes a simple property about 3 lines.
- Translating the whole polygon in one direction preserves the topology completely, so we know that there will be a **solution for any shift applied to the focus edge**.

Our simple approach

Simple algorithm to find a correct configuration given a line l_0 to move by an extent δ:

1. Compute the shift direction as the normal $\vec{n}$ of l_0.
2. Move l_0 by $\delta\vec{n}$.
3. If $\mathcal{P}(l_0)$ is not satisfied, move l_0^- and l_0^+ by $\delta\vec{n}$ as well. This ensures that $\mathcal{P}(l_0)$ is satisfied.
4. Set $l_p = l_0^-$. While either of $\mathcal{P}(l_p)$, $\mathcal{P}(l_p^-)$ or $\mathcal{P}(l_p^+)$ is not satisfied, move l_p by $\delta\vec{n}$ (if not already moved) and set $l_p = l_p^-$.
5. Set $l_n = l_0^+$. While either of $\mathcal{P}(l_n)$, $\mathcal{P}(l_n^-)$ or $\mathcal{P}(l_n^+)$ is not satisfied, move l_n by $\delta\vec{n}$ (if not already moved) and set $l_n = l_n^+$.

The only guarantees of this algorithm is that it will **terminate** and that the resulting configuration will **satisfy the constraints**.

However, it will **not find an optimal solution**, and in particular, it is **not continuous** with respect to δ.

Illustrations

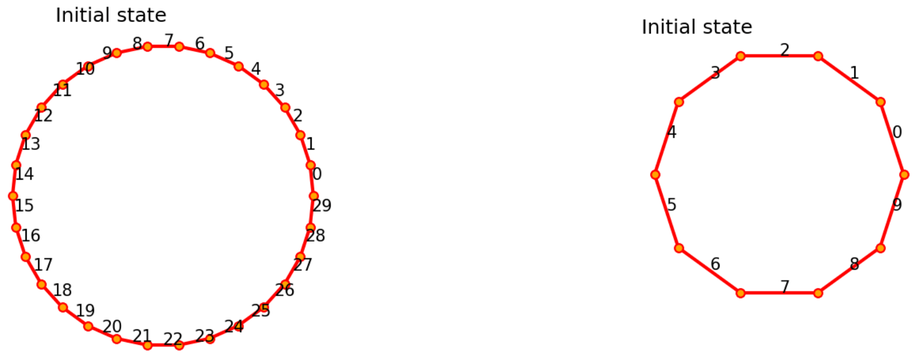


Figure 8: Simple illustrations of the polygon deformation algorithm (click to view the animated versions).

Illustrations

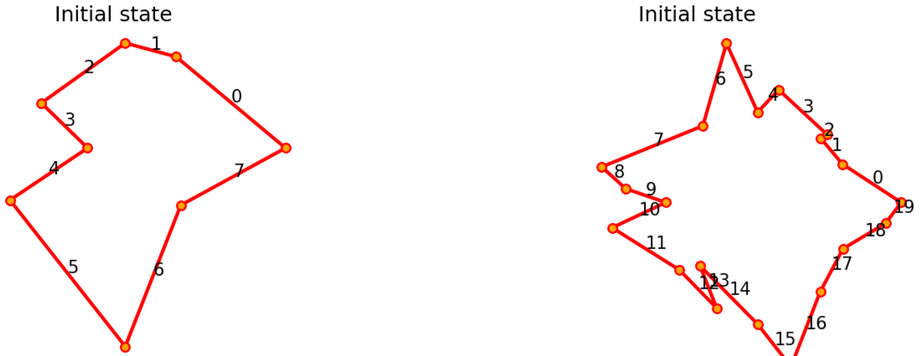


Figure 9: More complex illustrations of the polygon deformation algorithm (click to view the animated versions).

Illustrations

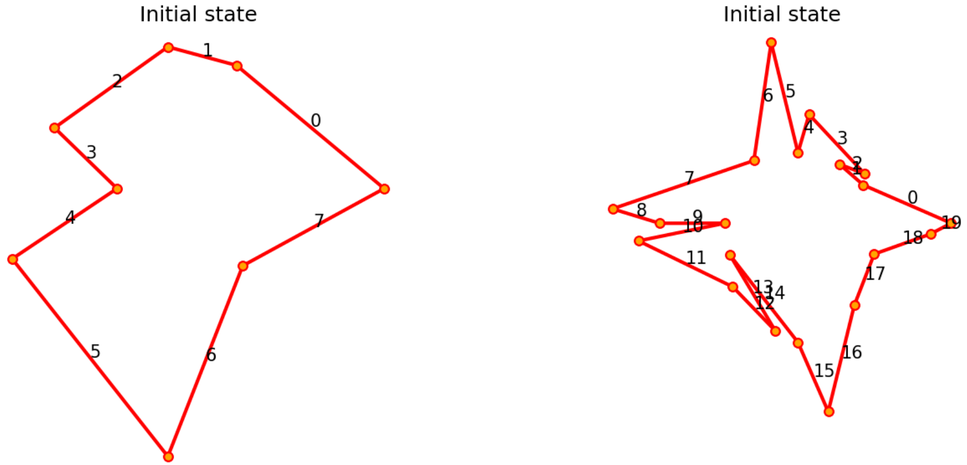


Figure 10: Illustrations of self-intersections with the polygon deformation algorithm (click to view the animated versions).

There we can see two different examples of how the algorithm can create configurations with **self-intersections**.

Criterion

Total energy to minimize

The energy that we try to minimize for each group of roofprints is the following:

$$E = \underbrace{-\sum_{i \in \mathcal{P}} w_i \max_{j \in \mathcal{L}} \{\text{score}(p_i, l_j)\}}_{\text{proximity to the points}} + \alpha \underbrace{\sum_{j \in \mathcal{L}} \left(|l_j| - |l_j^0|\right)^2}_{\text{similarity to the initial edges}}$$

Where:

- $\mathcal{P}$ is the set of points, and $\mathcal{L}$ is the set of edges (lines). $\mathcal{L}$ contains all the edges moved in any configuration and their neighbours.
- w_i is the weight of point p_i, which is independent of the lines.
- $\text{score}(p_i, l_j)$ is the score of point p_i for edge l_j, which is defined in the next slide.
- $|l_j|$ is the length of edge l_j in the current configuration, and $|l_j^0|$ is its initial length.
- α is a parameter to adjust between the two terms.

- The energy is divided in two terms:
  - The first term is a **proximity term** that both counts the number of points close to the edges while prioritizing points with higher weights.
  - The second term is a **regularization term** that entices the edges to keep their initial length, to prioritize a shape closer to the initial one.
- Many regularization terms are possible, but we chose this one because it makes sure that by default, the edges will **keep their initial length** and otherwise **share the change equally** between them. This is desired if we assume that the scaling that we need to perform is the same on both sides for a given direction, no matter the size of the edges, because the roof overhang is the same.

Definition of the proximity score

The score of a point p_i for an edge l_j is defined as follows:

- The score is 0 if any of the following conditions is true:
  - The orthogonal projection $p_{i\perp j}$ of p_i on the line supporting of l_j is outside of the segment l_j.
  - The distance from p_i to $p_{i\perp j}$ is greater than a certain threshold ε.
  - The dot product of the inward vector v_i of p_i with the normal n_j of l_j oriented towards the inside of the building is negative.
- Otherwise, the score is:

$$\text{score}(p_i, l_j) = \underbrace{(v_i \cdot n_j)}_{\substack{\text{alignment of} \\ \text{point and edge} \\ \text{'normals'}}} \times \underbrace{\left(1 - \frac{|p_i - p_{i\perp j}|}{\varepsilon}\right)}_{\text{proximity to the edge}} \geq 0$$

We use $\varepsilon = 30$ centimetres.

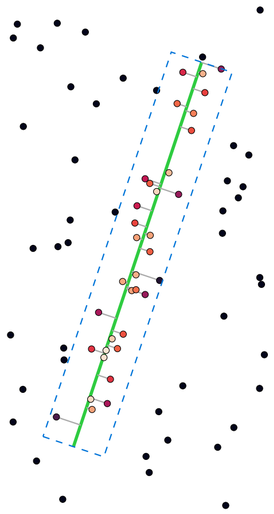


Figure 11: Illustration of the proximity score for an isolated edge.

- Any point **outside the rectangle** defined by extruding the edge along its normal by ε has a score of 0. This geometric ensures that only points that are close enough to the edge will count.
- Then, the dot product between the inward vector and the normal of the edge ensures that points that are part of a parallel but opposite edge will not count, **preventing matching to the neighbour building.**

Definition of the inward direction

The goal of the **inward direction** is to be as close as possible to the 2D normal of the roof edge, pointing towards the inside of the building. To compute it for a given point p_i, we use the following method:

1. Identify all the points p_j that are less than a certain distance δ from p_i.
2. For each point p_j, compute the vector from p_i to p_j, and normalize it into a unit vector $u_{i \to j}$.
3. Compute the inward vector v_i as the average of the vectors $u_{i \to j}$:

$$v_i = \frac{1}{|\mathcal{B}(p_i, \delta)|} \sum_{p_j \in \mathcal{B}(p_i, \delta)} \frac{p_j - p_i}{|p_j - p_i|}$$

We use $\delta = 2$ metres.

- The idea behind this method is that points on the edge of the roof should have **many points towards the inside** of the building, and **few points towards the outside**.
- Using unit vectors and averaging them allows for all points in the neighbourhood to contribute equally, meaning that we are somewhat counting the **density of points** in each direction. Therefore, the **magnitude** of the inward vector is an indication of the confidence of the direction.

Illustrations

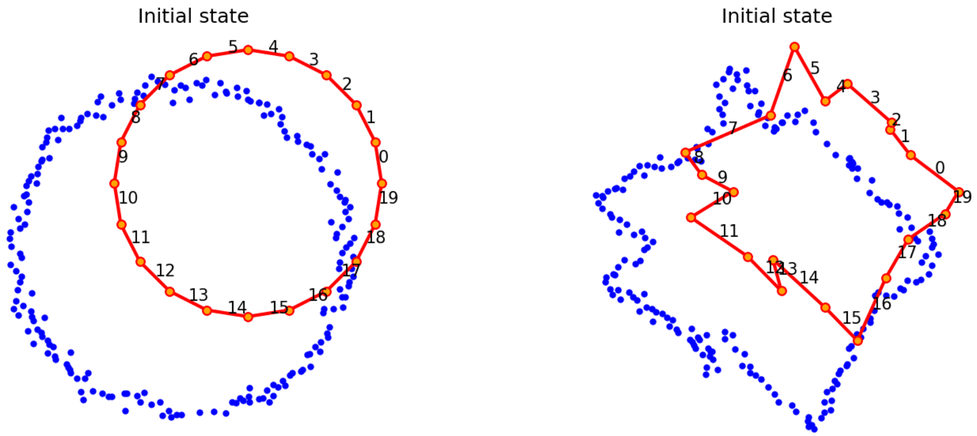
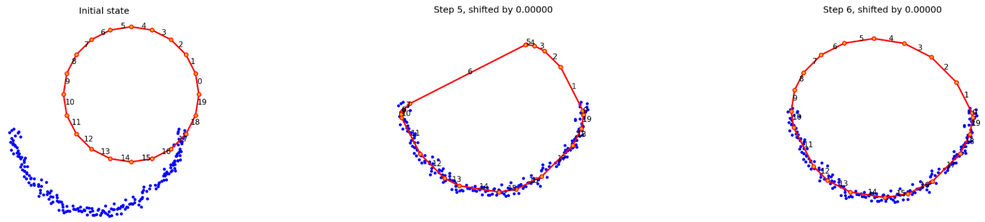


Figure 12: Simple illustrations of the full polygon deformation algorithm (click to view the animated versions).

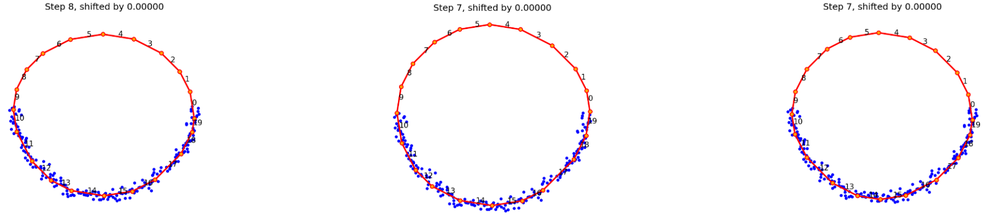
Here and in the following, this figures show:

- In blue, the points we match to
- In red, the current version of the polygon, with each edge numbered and vertices shown in orange
- At the top, the number of iterations (i.e. the number of times we went through all the edges) and the sum of the shifts performed during the last iteration, which is an indication of the convergence of the algorithm.

Illustrations



(a) Initial state (b) $\alpha = 0.00$ (no regularization) (c) $\alpha = 0.05$

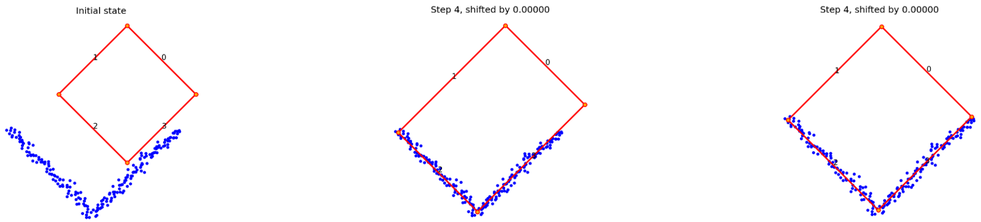


(d) $\alpha = 0.20$ (e) $\alpha = 0.50$ (f) $\alpha = 1.00$

Figure 13: Matching a circle to a half-circle of points with different values of the regularization parameter α.

We can see how regularization helps to get a shape more similar to the target even with half of the points missing.

Illustrations



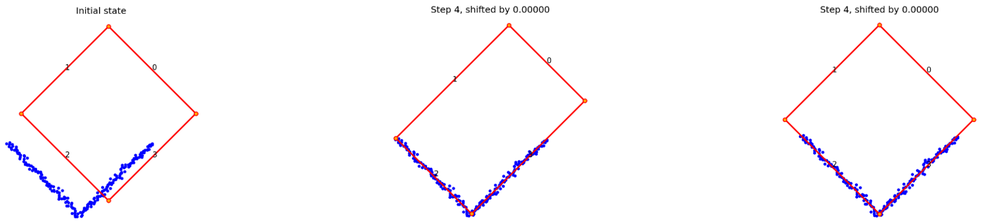
(a) Initial state (b) $\alpha = 0.00$ (no regularization) (c) $\alpha = 0.05$

(d) $\alpha = 0.20$ (e) $\alpha = 0.50$ (f) $\alpha = 1.00$

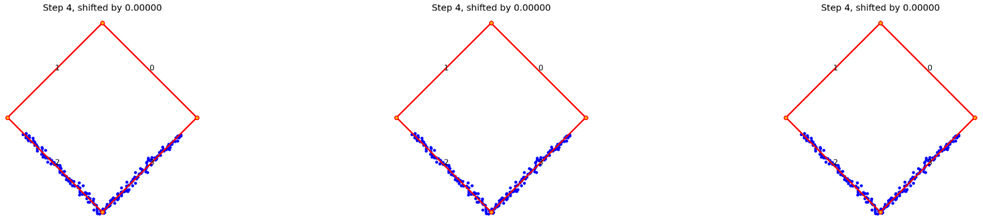
Figure 14: Matching a square to points along two sides of a larger square with different values of the regularization parameter α.

$\alpha > 0$ encourages the shape to be closer to the initial one, which is better than $\alpha = 0$, until a certain point where the regularization is too strong and prevents the shape from extending to capture all the points.

Illustrations



(a) Initial state (b) $\alpha = 0.00$ (no regularization) (c) $\alpha = 0.05$

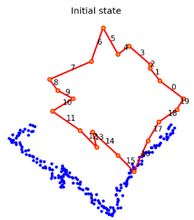


(d) $\alpha = 0.20$ (e) $\alpha = 0.50$ (f) $\alpha = 1.00$

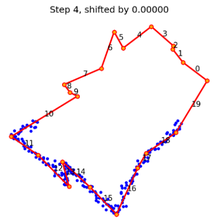
Figure 15: Matching a square to points along two sides of a smaller square with different values of the regularization parameter α.

$\alpha > 0$ encourages the shape to be closer to the initial one, which is again better than $\alpha = 0$, but this time even the smallest value of α already keeps the shape the same size. The shape has no reason to be smaller: it would not give it a better proximity score, and it would give it a worse regularization score, so the algorithm converges to the same shape for all values of $\alpha > 0$.

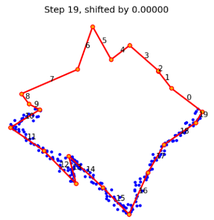
Illustrations



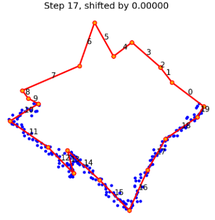
(a) Initial state



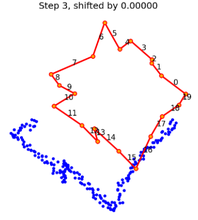
(b) $\alpha = 0.00$ (no regularization)



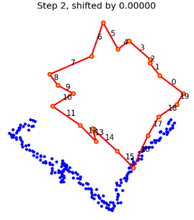
(c) $\alpha = 0.05$



(d) $\alpha = 0.20$



(e) $\alpha = 0.50$

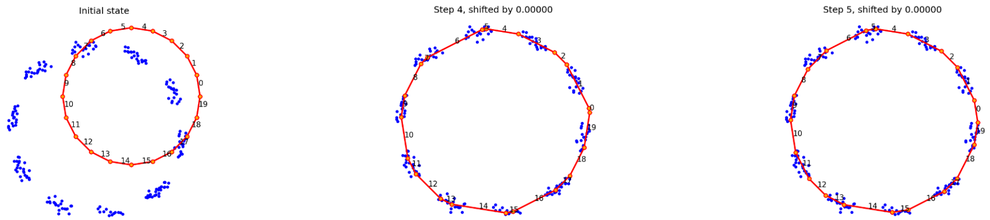


(f) $\alpha = 1.00$

Figure 16: Matching a polygon to half of its shape with points with different values of the regularization parameter α.

- $\alpha > 0$ forces the shape to be closer to the initial one, but it takes many more steps to converge, because the more edges that don't have points take longer to balance the regularization term between each other.
- If α is too high, it prevents the shape from matching the points, because the proximity score is not good enough to balance the regularization term, which is the case here for $\alpha \geq 0.50$.

Illustrations



(a) Initial state (b) $\alpha = 0.00$ (no regularization) (c) $\alpha = 0.05$

(d) $\alpha = 0.20$ (e) $\alpha = 0.50$ (f) $\alpha = 1.00$

Figure 17: Matching a circle to points along half of its boundary with different values of the regularization parameter α.

At the cost of more iterations, **a larger α outputs a more regular shape**, closer in proportions to the initial one.

Last results

Computation of roofprints from BD TOPO

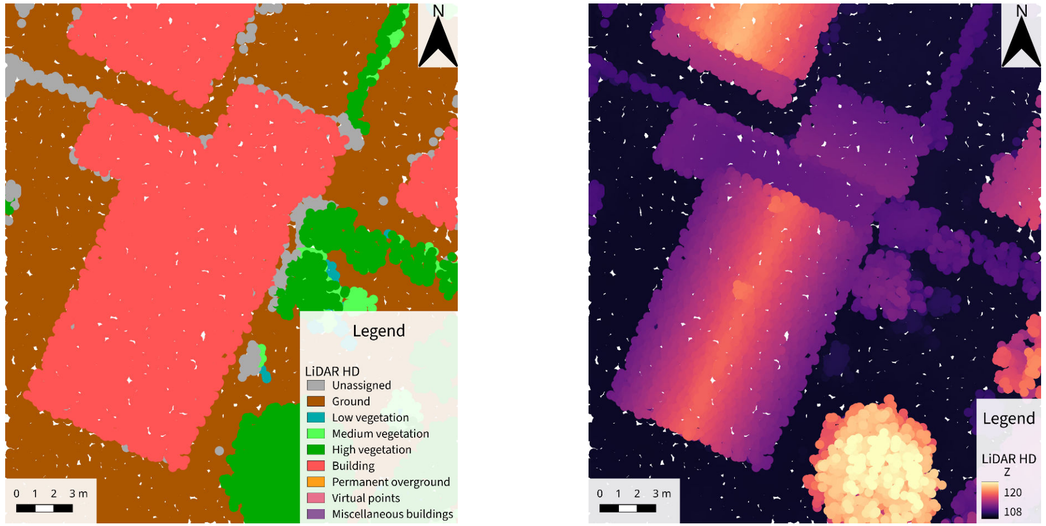


Figure 18: The LiDAR HD data.

- We can see **three different building parts** that are connected, one with a pitched roof and the two others with flat roofs.
- The points are well classified overall, but there are a number of points **classified as Unassigned (grey)**, especially at the boundaries between the building and the vegetation.

Computation of roofprints from BD TOPO

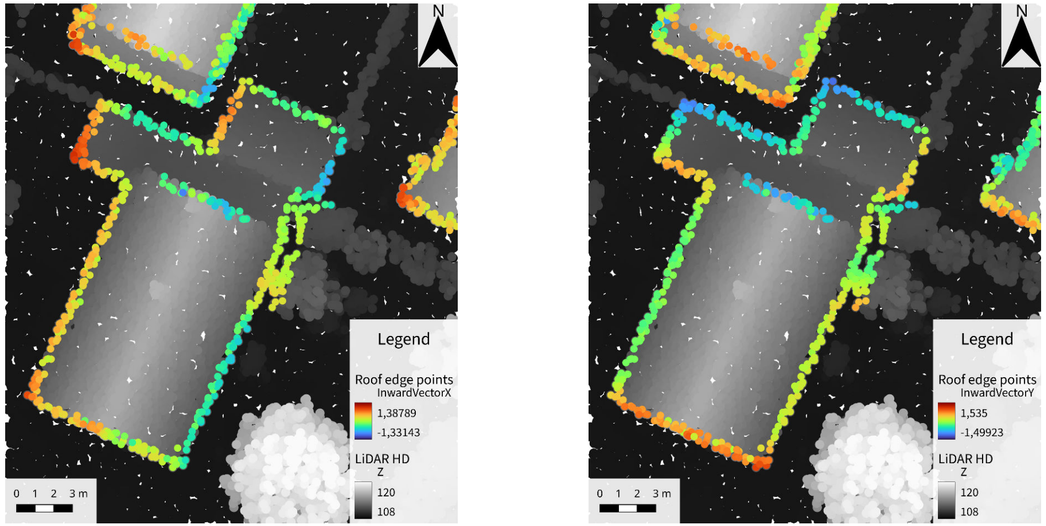


Figure 19: The detected roof edge points coloured by their inward vector.

- The detected roof edge points are shown here, coloured by their **inward vector** (the direction in which the building is located from the edge).
- We can see that the inward vectors are **generally well oriented** towards the building, except in areas where the building is **very close to vegetation** at the same height.

Computation of roofprints from BD TOPO

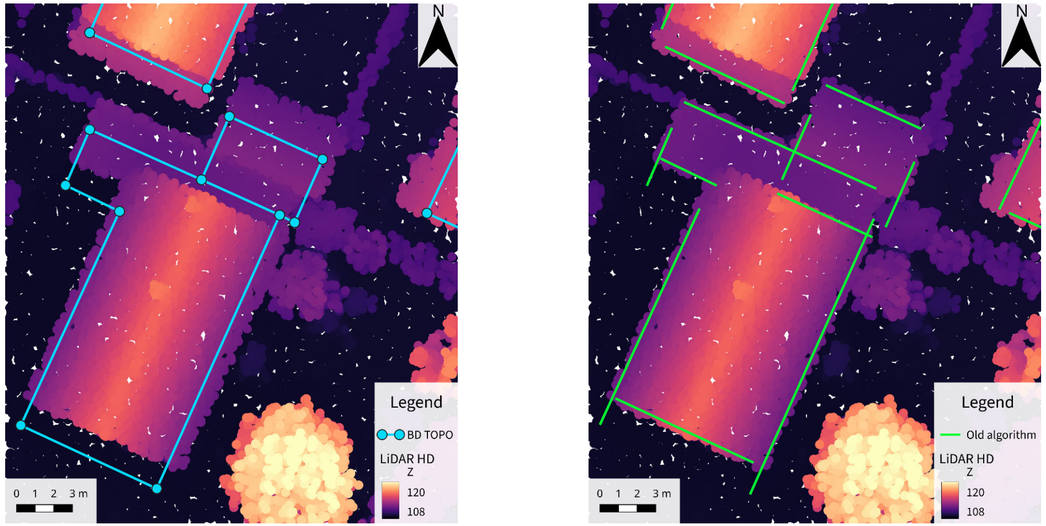


Figure 20: Comparison of BD TOPO outlines and roofprints computed with the old algorithm.

- Interestingly, this building unit is actually divided into **two parts in the BD TOPO**, even though we could expect 1 or 3 as well.
- In the old algorithm, we processed edges individually, resulting in assessing the segments that are displayed on the right.
- The results give a better alignment, but there is still an issue to fix: the bottom left edge of the small top right building was **matched incorrectly**. This could be fixed by **matching the whole initial line once** instead of keeping it split between buildings.

Computation of roofprints from BD TOPO

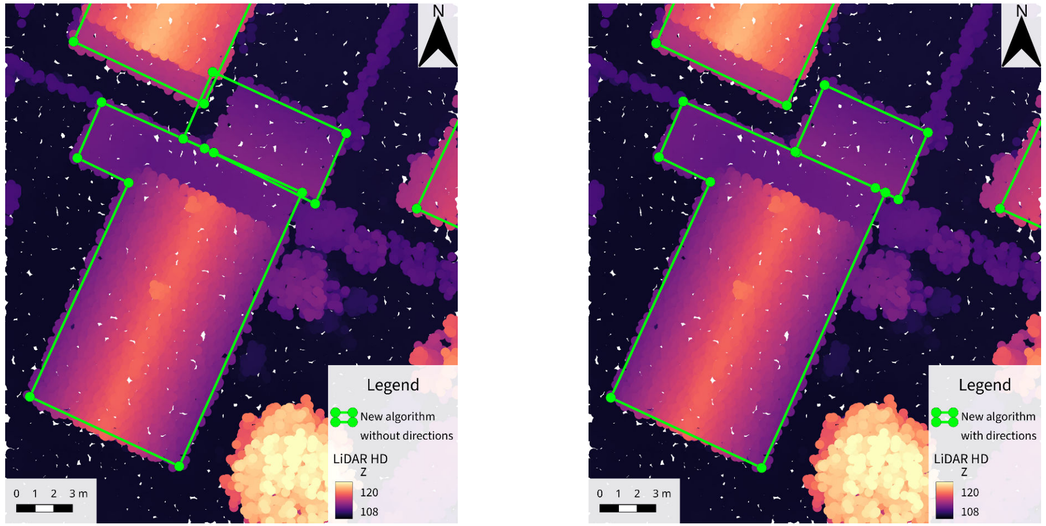


Figure 21: Comparison of the roofprints computed with the new algorithm without and with inward vectors.

- The new algorithm allows **fix the issue** for the bottom left edge of the small top right building, thanks to its alignment with the almost collinear edge of the big building.
- However, for the same building without using the inward vectors, one edge ends up **matching the neighbouring building** instead. This is fixed with the inward vectors, which prevent the points from the other building from counting in the score of the edge.

Further improvements

Point cloud semantic segmentation

Planned

- Look into using the **height** of points and normals to tell apart points on façades and roof edges.
- Look into training a deep learning model to **classify points** into more specific classes (such as roof and façade)

Not planned

- Consider **neighbours in other directions** than scan order (such as perpendicularly).
- Identify **lines in the point cloud topology** with Wu Teng's method:
  - Could be used to estimate the **normals locally** in cases where segments are found to estimate if points are on façades or on roofs.
  - Could be used to identify the breaks of roof planes (out of scope for now).

- To create a dataset to train a deep learning model, we were thinking of using the **AHN** and the **3DBAG**, which are the Dutch equivalents of the LiDAR HD and the BD TOPO in 3D, and split the points between the different classes based on the **3DBAG building models.**

Roofprint generation

Planned

- Experiment with the order of edge processing. For now it is **longer edges first**, but we could try random order.
- Start with a single 2D translation for all edges, to see if it can improve the results.
- (Maybe) run the **whole pipeline multiple times** with different conditions (different orders, different initial shifts, etc.) and pick the best result.

Not planned

- Use the **repartition of points** on the edge in their score to prioritize a less points with a uniform repartition over a lot of points clustered on one side of the edge.
- Use **combinatorial optimization** to pick the best set of edges with multiple solutions for difficult edges.
- Allow for small **rotations** of the edges.
- Look at edges in **3D** instead of 2D.

Further improvements — Roofprint generation

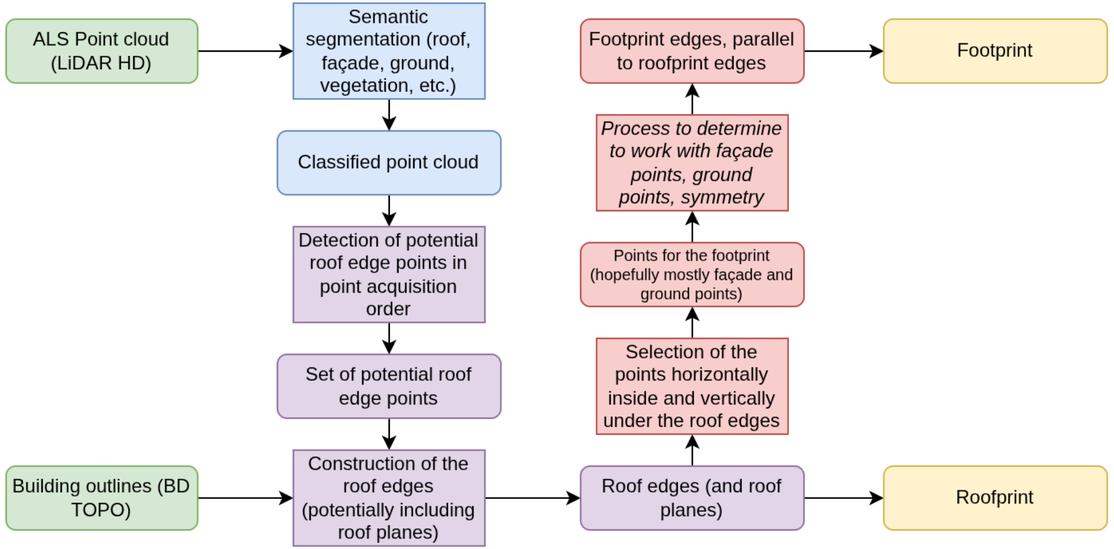
- Different orders and especially changing orders could help untangling some situations.
- **Different conditions** could really lead to **different results**, so it could be interesting, but it may be **difficult to pick the best result**.
- **Reward uniform distribution** of points along the edge with a histogram-like metric.
- Small rotations of the edges could drastically improve the results in some cases, but it makes preserving the topology more complex and goes against our assumption that angles are great in the initial outlines.
- Looking at edges in 3D could help a lot to separate points on roof edges from points on façades or vegetation. But is much more complex because it adds two new dimensions: the base height and the angle of the edge. Moreover, what is one straight edge in 2D may be multiple segments in 3D.

Footprint generation

Planned

- Find criteria to differentiate between roof edges and façade points (including training a deep learning model if needed).
- Use the **height variations in acquisition order** with different thresholds.
- Use **rays directions** to count points on potential façades **positively** and **negatively**.
- **Adapt the matching criterion** to the façades:
  - Not easy to compute the **inward vector** (potentially with the ground points).
  - Care even more about **distribution of points along the Z axis** to avoid matching to roof edges or vegetation.

Rest of the pipeline



Thank you for your attention!

Any questions?

alexandre.bry@ign.fr