Context

Right angles and parallel structures are generally the rule for actual buildings. When mapping building footprints, either for an authoritative agency or in OpenStreetMap (OSM), many factors will affect the accuracy of the result (i.e. right angles and parallel structures are not exactly respected). Initially, this procedure was developed to orthogonalize building footprints made available by Canadian government agencies, in order to import them in OSM (with a compatible license). A description of the concepts behind orthogonalizing building footprints and the algorithm are detailed below.

Concepts Behind Orthogonalizing Building Footprints

First, not all sections of a building are necessarily orthogonal since some architectural features may impose different shapes.

1. Uncertainties on right angles.

For instance, let start with a simple building having a right triangle footprint. Mapped footprints generally differ from the actual building shape (figure 1).

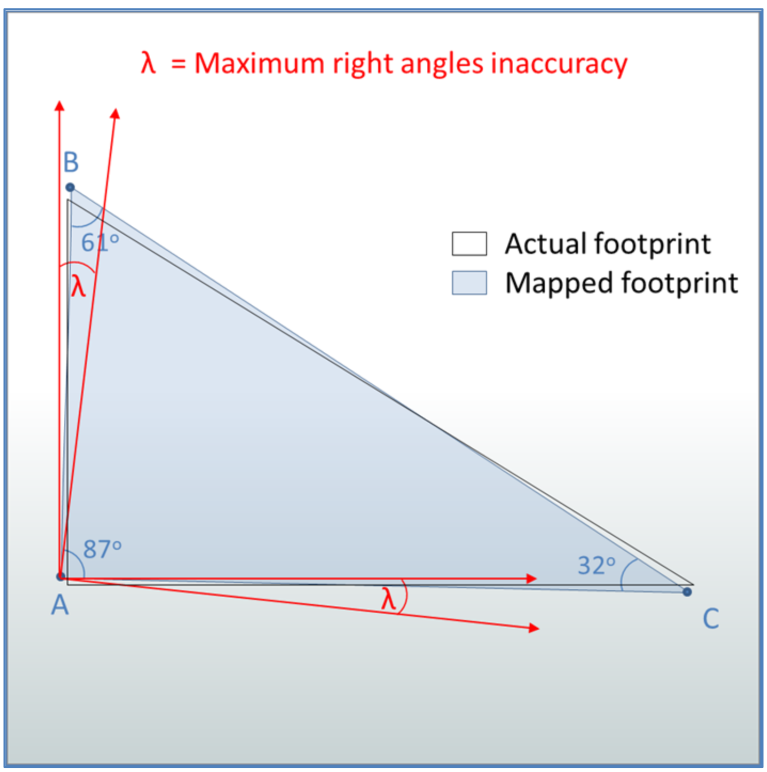


Figure 1 Maximum right angles inaccuracy

Segments AB and AC show inaccuracies which make angle A not orthogonal. At the same time, corners B and C do not have right angles and must not be orthogonalized. In order to differentiate orthogonal sections from those who are not, a parameter must define a maximum mapping error around right angles.

Below the threshold λ, angles must be orthogonalized, above they must not. A threshold λ of 10^{o} defines that all segments linked with an angle of $90^{o} \pm 10^{o}$ must be orthogonalized. In figure 2, angle A (i.e. segments AB and AC) must be orthogonalized while the segment BC must not. The same rule applies for more complex buildings, creating sections of contiguous right angle segments and possibly sections of contiguous non-orthogonal segments.

Orthogonalizing a section of contiguous right angle segments consists of imposing the same slope to all parallel segments and an orthogonal slope to each adjacent segments (i.e. only two angles). When imposing the slope to a segment, segment centre coordinate is used to determine its new location.

2. Orthogonalizing Building Footprints Aggregate

In urban environment, buildings are often contiguous to each other, creating aggregates of buildings. The same rule applies to aggregated building footprints. All contiguous sections of right angle footprints must be imposed an orthogonal slope to each adjacent segments and the same slope to parallel segments. Doing so, adjacent segments that should be collinear may end-up as parallel because of the uncertainty on each segment. It is then necessary to impose a minimum distance ε on parallel segments to ensure that below this threshold, they will end up collinear and share the same coordinate at their edges (Figure 2).

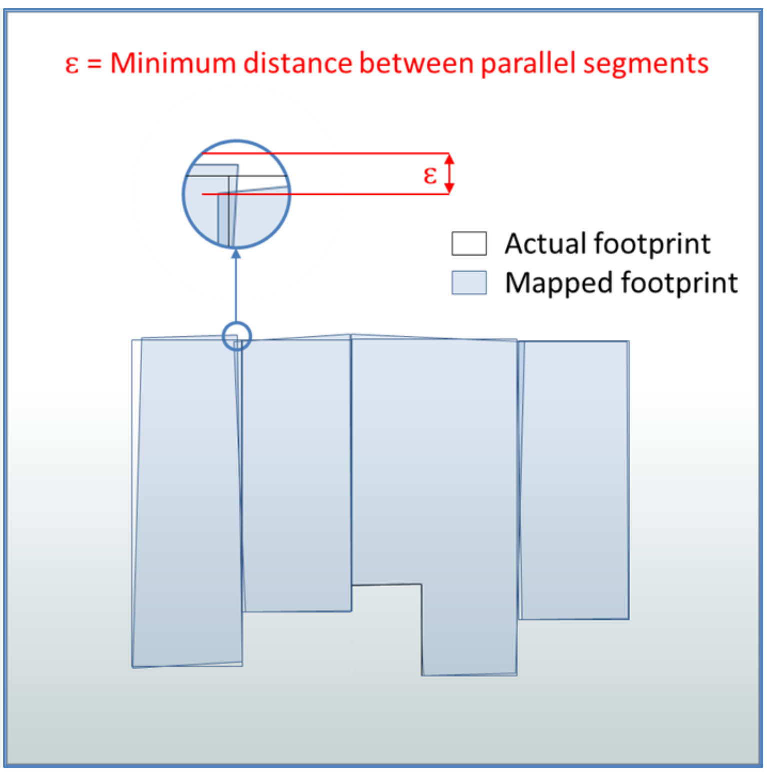


Figure 2 Minimum distance between parallel segments

A line equation of the general form Ax+By+C=0 is built for each line segment using the new common orthogonal slopes and each segment's centre coordinate. The line equations are adjusted to respect ε by rounding C by that value. Rounding C with ε makes collinear the segments of battlement shape buildings when the distance between segments' projection is smaller than ε.

3. Removing Collinear Vertices

Another characteristic of mapped footprints is the existence of unnecessary vertices (nodes) which may slightly alter the shape of the buildings (figure 3). Algorithms exist to remove these vertices based on the resulting line displacement after removing a given vertex (e.g. Douglas-Peucker). However, on building footprints, small segments (i.e. small displacement after removing vertices) may represent important building features and should not be removed. Fortunately, unnecessary vertices generate very small angles. One can define an angle threshold ϴ, below which removing a vertex as not significant impact on building footprint. A threshold ϴ smaller than the stroking angles of round buildings is advisable.

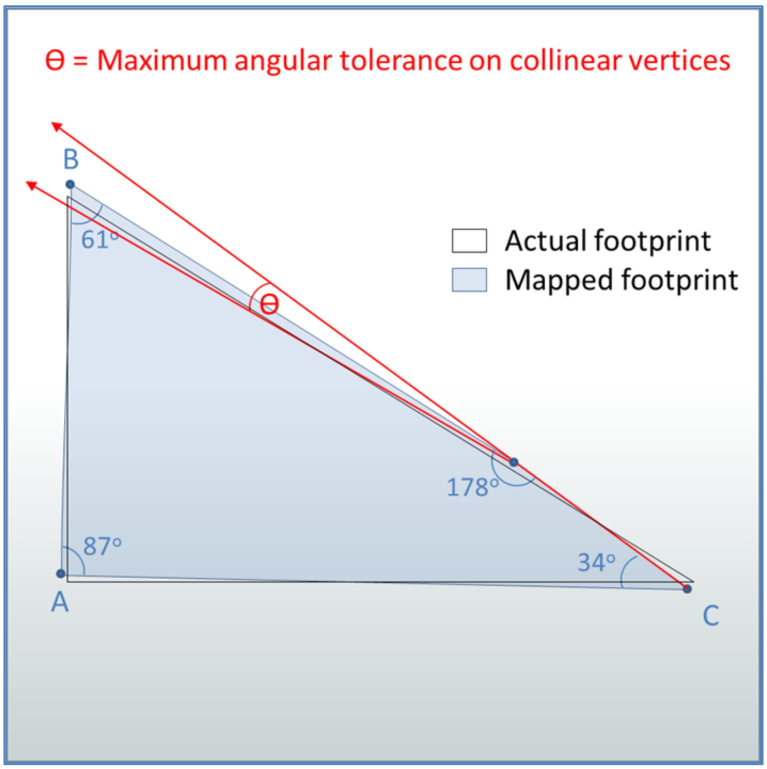


Figure 3 Maximum angular tolerance on collinear vertices

In the context of this procedure, where right angle line segments are expected over orthogonal sections, vertices having an angle smaller than the threshold λ must be removed from these sections because once orthogonalized, they will have a resulting angle of zero (0) degree.

4. Summary

Not all sections of a building are necessarily orthogonal since some architectural features may impose different shapes. The junction of each line segment results in three angle types.

- The collinear or near-collinear vertices and are defined through the angle threshold ϴ (Figure 3). Vertices creating an angle smaller than that threshold are removed.

Orthogonalizing Building Footprint – General Concepts

- Orthogonal or near-orthogonal angles are defined by using a threshold λ (Figure 1) which defines the maximum angular error on right angles. When an angle differs from 90o by less than that threshold, the angle is considered to be orthogonal and the corresponding line segments are modified to ensure a right angle between them.
- Between angles Ɵ and 90±λ are non-orthogonal angles which should not be altered since there are no rules that define their expected values. The one exception is when angles <= λ is adjacent to orthogonal segments. In these cases, they are considered near collinear due to the orthogonalization process described below.

In order to orthogonalized line segments, each line segment equation (Ax+By+C=0) is computed. The distance threshold ε (Figure 2) is used to ensure expected collinear segments are created as such by adjusting the C component values. Once new line equations are defined, the intersection between adjacent segments (new vertex coordinate) is computed using homogeneous coordinates.

Orthogonalizing Building Footprints Algorithm (kind of...)[1]

The following text uses OGC primitive geometry definitions (2D). Building footprints are polygons. A polygon may have inner polygons (holes) in it. A polygon may also be seen as a close linestring (first/last vertices have the same coordinates). Each step of the following algorithm is illustrated in annexes.

1. Features Cleanup & Attributes Setting

Result: Polygons have required geometry and attributes for processing

1.1. Objects identifiers and projection

Result: Each polygons' geometry has planar coordinates and a unique identifier (id.object).

original[*].id.object = uuid() – Generate a unique identifier for each original polygon[2].
copy[*].ring[*].vertex[*].xy = original[*].ring[*].vertex[*].xy – Copy the original geometry.
copy[*].id.object = original[*].id.object – Add the original polygon's uuid to the working copy.
copy[*].* = projection(copy[*], AZMED) – Project polygons in a conform projection[3].

1.2. Polygon Inner/Outer Components Identification

Result: Inner and outer polygons stand alone. Each component has an *id.object* and inner polygons know their outer polygon uuid (*id.partOf*).

For each copy[i].ring[j=1-N] as C.R : – Copy polygon's inner rings [4]
  polygon[*].vertex[*].* = C.R.vertex[*].* – Creates a new polygon from ring[i]
  polygon[*].id.partOf = P.id.object – Copy original polygon's id to the new polygon
  polygon[*].id.object = uuid() – Give a uuid to the new polygon

For each copy[i].ring[0] as C.R : – Copy polygon's outer ring
  polygon[*].vertex[*].* = C.R.vertex[*].* – Creates a new polygon from ring[i]
  polygon[*].id.object = C.id.object – Give a uuid to the new polygon

1.3. Aggregated Buildings Identification

Result: Groups of adjacent buildings (not disjoint) have a same aggregate uuid (id.aggregate). This information is later used to generalize line equations parameters for a same aggregate.

aggregate[*].* = dissolve(polygon[*].*) – Create a polygon around touching/overlapped polygons[5].
aggregate[*].id.aggregate = uuid() – Generate a uuid for each aggregate.

For each polygon[i] :
  polygon[i].id.aggregate = aggregate[contains(polygon[i])].id.object – Copy id from aggregate.

---

[1] The algorithm could be largely improved/simplified/optimized. It is written mixing explanations, SQL, R and FME coding.
[2] The uuid is later used to link orthogonalized geometry and original attributes.
[3] AZMED: Lambert Azimuthal Equidistant Projection is a local coordinate system in metres centred on the bounding box of each feature. Conformal projections such as Lambert Conformal Conic or Transverse Mercator can be used.
[4] Polygon's outer ring is stored in ring[0], inner rings are stored in ring[1-N]
[5] Removes all inner rings. The result is an outer ring polygon that topologically contains all touching polygons.

1.4. Linearize Near-Collinear Vertices

Result: Near-collinear vertices are aligned with each other (angles to adjacent segments are 0). This step helps to keep lines straight in case collinear vertices cannot be removed for topological reasons.

Straighten a temporary copy of polygons

For each polygon[i] as P :
  copy[i].* = P.* – Copy each polygon to a temporary version.
  copy[i].vertices = count(P.vertex[*]) – Count original number of vertices.

For each copy[i].vertex[j] as C.V : – Straighten polygons by removing collinear vertices.
  C.V.angularChange = angularChange(C.V) – Compute angle between incoming line segments [0-180].
  C.V.angularChange <= ϴ ? delete(C.V) : keep(C.V) – Remove collinear vertices[6].

For each copy[i] as C:
  C.vertices = count(C.vertex[*]) ? delete(A) : keep(A) – Keeps only straighten polygons (copy[i].*).

Straighten original polygons' geometry

For each polygon[i].vertex[j] as P.V:
  distance(P.V.xy, copy[*]) <= ε.? P.V.xy = snap(P.V.xy, copy[*]) : P.V.xy= xy – Straighten polygons[7].

1.5. Move Polygon Origin on Most Right Angle Vertex

Result: Polygon's origin (first/last vertex) is set on the polygon's most right angle corner. Although not mandatory, it simplifies later computations for complex cases.

Identify polygon's most right angle corner location

For each polygon[i].vertex[j] as P.V:
  P.V.id.vertex = j – Original index of the vertex.
  P.V.angularChange = angularChange(P.V) – Compute angle between incoming line segments [0-180].
  P.V.delta90 = abs(90-P.V.angularChange) – Compute difference to 90 degrees.

For each polygon[i] as P :
  Sort(P.vertex[*].delta90, increasing) – So the smallest difference appears first.
  P.neworigin = P.vertex[0].id.vertex – Original index in the polygon.
  P.maxIndex = max(P.vertex[*].id.vertex) – The largest index value (i.e. number of vertices – 1).

Reorder polygon's vertices to set new origin

For each *polygon[i]* as P :
  Delete(P.vertex[P.maxIndex].*) – Remove original "last" vertex.

---

[6] Conditional operations: *Condition* ? *true* : *false* (similar to an if then else statement).
[7] Distance between all original polygons' vertices and remaining straighten polygons' is computed (considering rings as linestrings). When measured distance is smaller than ε, original polygons' vertices are snapped to nearest straighten polygons' rings (i.e. projected to the nearest line segment).

For each polygon[i].vertex[j] as P.V :

(PV.id.vertex – polygon[i].newOrigin >= 0)? – The new origin is located before current location…

True: PV.id.vertex.new = PV.id.vertex - polygon[i].newOrigin

False: PV.id.vertex.new = polygon[i].maxIndex - polygon[i].newOrigin + PV.id.vertex

(PV.id.vertex.new = 0)? – Create new "last" vertex of polygon (i.e. duplicate first one)

True: polygon[i].vertex[polygon[i].maxIndex].* = polygon[i].vertex[i].*

For each polygon[i] :

newVertices[*] = sort(polygon[i].vertex[*].id.vertex.new, Ascending).
polygon[i].* = rebuilt(polygon[i].*, newVertices[*]) – Rebuilt polygon from reordered vertices.

2. *Polygon Segmentation on Angular Changes Classification*

Result: Polygons are converted into linestrings which are split into orthogonal / nonorthogonal segments. Topologically related segments are group into sections, each having a unique identifier (*id.section*).

2.1. *Angular Type Identification*

Result: Each vertex knows its angular type (orthogonal or not). Locations of changes between orthogonal/non-orthogonal angles are known.

Compute angular changes and their type in both forward and backward directions[8]

angularTypeDefinition(angularChange): - Angular types definition function
angularType =0 if angularChange in [90-λ, 90+λ] (orthogonal vertex).
angularType =1 if angularChange in [0, ϴ] (collinear vertex).
angularType =2 if angularChange in [ϴ, λ] (collinear vertex if adjacent to orthogonal one).
angularType =3 if angularChange in [λ, 90-λ],[90+λ,180] (non-orthogonal vertex).
angularType =9 as an initial value before attempting to compute the type of angularChange.

For each polygon[i].vertex[j] as P.V :

P.V.angularChange = angularChange(P.V) – Compute angle between incoming line segments [0-180].
P.V.AngularType=9 – Initialize angular type to know a vertex has not been evaluated in one direction.

Initialize type for next loops

angularType.forward=0 – Initialize type for next loop
angularType.backward=0 – Initialize type for next loop
index.forward.defined=0 – Initialize index for next loop
index.backward.defined=0 – Initialize index for next loop

For each *polygon[i].vertex[j]* as P.V *:* – **Forward direction j=[0-N]**

angularType.vertex = angularTypeDefinition(P.V.angularChange)
angularType.change = angularType.vertex-angularType.forward

[8] Angular changes are classified in 4 different angular types (*angularType*). Angular type are either orthogonal [0], non-orthogonal [3] or collinear [1,2]. Collinear vertices [1,2] provide no meaningful information. In these cases, the last significant value of *angularType* [0,3], in both directions, makes it possible to later identify the actual type.

(angularType.change=(3|-3) & angularType.vertex=(0|3)?
True: There is a change in angular type. Extract the location of the change.

index.change = (angularType.change)>0) ? j : index.forward.defined
polygon[i].angularChange[*].xy = polygon[i].vertex[index.change].xy

(angularType.vertex=(0|3)?
True: Current angular type is defined (0|3)– Update current vertex information.
index.forward.defined = j
angularType.forward = angularType.vertex
angularType.measure = angularType.vertex

False: Current angular type is not defined (1|2) – Update angularType.measure
angularType.measure = P.V.AngularType
angularType.measure = min(angularType.forward,angularType.measure)

P.V.angularType = angularType.measure

For each polygon[i].vertex[k] as P.V **– Backward direction k=[N-0]**
angularType.vertex = angularTypeDefinition(P.V.angularChange)
angularType.change = angularType.vertex-angularType.backward

(angularType.change=(3|-3) & angularType.vertex=(0|3)?
True: There is a change in angular type. Extract the location of the change.

index.change = (angularType.change)>0) ? j : index.backward.defined
polygon[i].angularChange[*].xy = polygon[i].vertex[index.change].xy

(angularType.vertex=(0|3)?
True: Current angular type is defined (0|3)– Update current vertex information.
index.backward.defined = j
angularType.backward = angularType.vertex
angularType.measure = angularType.vertex

False: Current angular type is not defined (1|2) – Update angularType.measure
angularType.measure = P.V.AngularType
angularType.measure = min(angularType.backward,angularType.measure)

P.V.angularType = angularType.measure

2.2. Polygon Segmentation

Result: Polygons are split at above changes location (polygon[*].angularChange[*].xy), if any. Each resulting linestring knows its angular type (orthogonal or not). Superposed linestrings have their angular type made identical, prioritizing orthogonal classification.

Segment polygon at each angular change

linestring[*].* = split(polygon[*].*, polygon[*].angularChange[*].xy) – Split polygons at changes location.

For each linestring[i] as L : - Identify the type of angular changes from each linestring (0|3)
L.section.angularType = min(L.vertex[*].angularType

Segment linestrings at intersections

linestring[*].*= intersect(linestring[*].*, overlapped= overlaps) - Split linestrings at intersections[9]

For each linestring[i] as L : - Identify the type of angular changes (0|3)
L.section.angularType = min(L.vertex[*].AngularType)
linestring[*].*= explode(linestring[*].*, by id.object) – Duplicate linestring overlapping segments by id

2.3. Linestrings Angular Type Topology

Result: orthogonal linestrings are grouped into networks of topologically related objects. All linestrings will be assigned a network id (id.section) but each non-orthogonal linestring have its own id.

Shorten linestrings where necessary

shortenLinestring[*]=linestring[*] – makes a temporary copy of linestring

For each shortenLinestring[i] as S : - Shorten (disconnect) extremities that are non-orthogonal (type=3)[10]
segment.vertex.indexF=0 – Index of first linestring coordinate.
segment.vertex.indexL= NumCoords(S)-1 – Number of coordinates in current linestring – 1.

(S.vertex[segment.vertex.indexF].angularType =3)? - Shorten linestring at first coordinate if necessary.
True: S.vertex[segment.vertex.indexF].xy = shorten(S.vertex[*], segment.vertex.indexF , 0.001)[11]

(S.vertex[segment.vertex.indexL].angularType =3)? - Shorten linestring at last coordinate if necessary.
True: S.vertex[segment.vertex.indexL].xy = shorten(S.vertex[*], segment.vertex.indexL , 0.001)

Build topological networks of orthogonal linestrings

shortenLinestring[*].id.section= network(shortenLinestring[*], by section.angularType)[12]

For each linestring[i] : - Transfer shortenLinestring[*].id.section to original linestrings
linestring[i].id.section = shortenLinestring[i].id.section

3. Segments Characteristics

Result: Linestrings are split into individual line segments (2 vertices each) and each segment held its Ax+By+C equation. Line equations are generalized by id.section then by id.aggregate, when the difference between sections and the corresponding aggregate are smaller than λ.

---

[9] Computes intersections between all input features, breaking lines and polygons wherever an intersection occurs. Overlapping segments are reduced to one segment before being output, with a list of overlapped segments ids (id.object).
[10] This way, it ensure that non-orthogonal linestrings will not be included in any network (except its own)
[11] Shorten linestring at given index, by a small amount (0.001), to disconnect identified extremity from the other linestrings.
[12] Build and identify networks of topologically related objects (related orthogonal linestrings will have the same id.section)

3.1. Linestrings Cleaner

Result: Collinear vertices are removed from linestrings according to threshold ϴ for non-orthogonal segments, and threshold λ for orthogonal ones.

For each *linestring[i].vertex[1-(j-1)]* as L.V : – Excludes first and last vertex
  L.V.angularChange = angularChange(L.V) – Compute angle between incoming line segments [0-180].
  ((L.V.angularChange<=λ & L.angularType=1) || (L.V.angularChange<=ϴ & L.angularType=-1))?
    True: Remove LV.*

3.2. Original Segment Characteristics

Result: Linestrings are split into individual line segments (2 vertices each) and all required segment characteristics to computed line equation is available.

segment[*].* = chopper(linestring[*].*, coordinates=2) - Split linestrings into line segments[13].

Compute each segment unitary vector characteristics

For each segment[i] :
  segment[i].length = length(segment[i].vertex[*].xy)

Compute centre line coordinate

  segment[i].cx = (segment[i].vertex[0].x + segment[i].vertex[1].x)/2
  segment[i].cy = (segment[i].vertex[0].y + segment[i].vertex[1].y)/2

Transform to unitary vector (r=1)

  segment[i].dx = (segment[i].vertex[1].x- segment[i].vertex[0].x)/segment[i].length
  segment[i].dy = (segment[i].vertex[1].y- segment[i].vertex[0].y)/segment[i].length

(segment[i].dx <= 2e-32) ? – *initialize for dx=0.0*
    True: segment[i].slope = 2e+32
    False: segment[i].slope = min(segment[i].dy/segment[i].dx, 2e+32)

  segment[i].angle00 = atan(segment[i].slope)/pi()*180.0 – Original angle
  segment[i].angle90 = atan(1/segment[i].slope)/pi()*180.0 – Orthogonal angle

Standardize slope, angle, dx, dy [14](all segments are oriented in the same directions)

(segment[i].slope <= 2e-32)? – initialize for slope =0.0
    True: segment[i].slopeSign = 1
    False: segment[i].slopeSign = segment[i].slope/abs(segment[i].slope)

  segment[i].dx = abs(segment[i].dx)
  segment[i].dy = abs(segment[i].dy)* segment[i].slopeSign

---

[13] Segments have two coordinates
[14] Two parallel segments oriented toward opposite direction will be given the same dx, dy.

3.3. Standardized Orthogonal Segments Characteristics

Result: The slope (actually dx, dy) of each segment that belongs to an orthogonal section is generalized to obtain representative and uniform values (A,B) for each segment equation Ax+By+C=0 over each aggregate or over a section if not possible.[15]

Before further processing, orthogonal/non-orthogonal segments are split apart.

For each segment[i] : (segment[i].angularType = 1)?

False: segment[i].* belongs to an orthogonal section. Segments are directed to sections 3.3.1 to 3.3.3…

3.3.1. Compute Reference Direction (dx, dy) By Section

Result: A reference direction (dx,dy) for current section is associated to each segment[16]

section[*].length.sum = sum(segment[*].length, by segment[*].id.section)
section[*].length.max = max(segment[*].length, by segment[*].id.section)

For each section[i] :

section[i].reference = segment[index(segment[*].length = section[i].length.max)].*

For each segment[i] :

segment[i].reference.dx.section = section[segment[i].id.section].reference.dx
segment[i].reference.dy.section = section[segment[i].id.section].reference.dy
segment[i].section.length = section[segment[i].id.section].length.sum

3.3.2. Compute Reference Direction (dx, dy) By Aggregate

Result: A reference direction (dx,dy) for current aggregate is associated to each segment[17]

aggregate[*].length.sum = sum(segment[*].length, by segment[*].id.aggregate)
aggregate[*].length.max = max(segment[*].length, by segment[*].id.aggregate)

For each aggregate[i] :

aggregate[i].reference = segment[index(segment[*].length = aggregate[i].length.max)].*

For each segment[i] :

segment[i].reference.dx.aggregate = aggregate[segment[i].id.aggregate].reference.dx
segment[i].reference.dy.aggregate = aggregate[segment[i].id.aggregate].reference.dy
segment[i].aggregate.length = aggregate[segment[i].id.aggregate].length.sum

3.3.3. Associate Best Reference Direction (dx, dy) to segments[18]

Result: The most representative reference direction (dx, dy), from either segment's aggregate or section is associated to each segment.

---

[15] Representativeness is based on the longest segment of each section because shorter segments tend to have higher variations of orientations (angles) in their representation.

[16] General orientation/direction of the section is based on section's longest segment to keep building's general shape.

[17] General orientation/direction of the aggregate is based on aggregate's longest segment to keep building's general shape.

[18] Priority is given to aggregate reference (dx,dy), which should typically be the case, unless some sections' orientation differs from the aggregate by more than λ. In this case, section reference (dx,dy) is used instead.

For each segment[i] : – Compute distance between section's and aggregate's directions
    rdx = segment[i].reference.dx.aggregate - segment[i].reference.dx.section
    rdy = segment[i].reference.dy.aggregate - segment[i].reference.dy.section
    segment[i].reference.deltaDirection = sqrt(rdx**2 + rdy**2)

Build a list of references (dx,dy) by sections and aggregates

segment[*].id.aggregate as id.aggregate
segment[*].id.section as id.section
segment[*].reference.deltaDirection as deltaDirection

list[*].*=sort(segment[*].*, by= id.aggregate, deltaDirection, id.section, ascending, noduplicate)[19]

Scan Aggregate, section by section, to group similar segment orientation (dy,dy)

For each list[*].id.aggregate=[i] as L :
    aggregate[*].id.aggregate = L.id.aggregate
    aggregate[*].id.section = L.id.section
    aggregate[*].reference.dx.section = L.reference.dx.section
    aggregate[*].reference.dy.section = L.reference.dy.section
    aggregate[*].reference.dx = L[0].reference.dx.aggregate – Initialize reference.dx
    aggregate[*].reference.dy = L[0].reference.dy.aggregate – Initialize reference.dy
    aggregate[*].reference.id = 0 – Initialize general orientation group identifier.

For each aggregate[i] as A – Iterate over each section of each aggregate[20]
        rdx = A.reference.dx - A.reference.dx.section
        rdy = A.reference.dy - A.reference.dy.section
        reference.deltaDirection = sqrt(rdx**2 + rdy**2)

(reference.deltaDirection >= degToRad(λ))? – Changes in direction must be less than λ (radian)[21]
            True: Reset references
                A.reference.dx = A.reference.dx.section – Reset reference.dx to current section dx value
                A.reference.dy = A.reference.dy.section – Reset reference.dx to current section dx value
                A.reference.id = A.reference.id + 1 – Increment general orientation group identifier

        references[*].* = A.*

Associate reference orientation (dx,dy) to each segment

leftJoin(segment[*].*(A), references[*].*(B) where A.id.section = B.id.section) [22]

For each segment[i] as S:
    rdx = S.reference.dx – S.dx
    rdx = S.reference.dy – S.dy
    S.distance2reference= sqrt(rdx**2 + rdy**2)

[19] Result is a list of sections of each aggregate, ordered by the difference of direction between section and aggregate, from smallest to largest difference (i.e. similar sections appear first and later sections may be considered having different directions). Difference of direction is measured using unitary circle.
[20] The list is ordered from smallest to largest reference.deltaDirection for each section of each aggregate, .
[21] Since sections are ordered by difference of direction, the directions are not the same anymore if >= degToRad(λ).
[22] It creates *segment[*].reference.dx*, *segment[*].reference.dy and segment[*].reference.id.*

(S.distance2reference<@sqrt(2)/2)? – A weighted version of dx,dy are oriented in a same direction[23].

True: Segment and reference have the same orientation, Keep current orientation.

S.dx.weighted = S.dx * S.length
S.dy.weighted = S.dy * S.length

False: Segment and reference don't have the same orientation, use orthogonal version instead.

S.dx.weighted = S.dy * S.length *(-1)
S.dy.weighted = S.dx * S.length

Compute a weighted average dx,dy to standardize line equation of each segment[24]

segment[*].dx.weighted.sum = sum(Segment[*].dx.weighted, by id.aggregate, reference.id)
segment[*].dy.weighted.sum = sum(Segment[*].dy.weighted, by id.aggregate, reference.id)
segment[*].length.sum = sum(Segment[*].length, by id.aggregate, reference.id)

For each segment[i].* as S

S.standardized.dx = S.dx.weighted.sum/S.length.sum
S.standardized.dy = S.dy.weighted.sum/S.length.sum
S.standardized.slope = (S.standardized.dx==0)? 2e+32 : S.standardized.dy/S.standardized.dy
S.standardized.angle00 = atan(S.standardized.slope)/pi()*180
S.standardized.angle90 = atan(-1/S.standardized.slope)/pi()*180

(S.distance2reference<@sqrt(2)/2)? – Change standardized angle into its original orientation

True: S.standardized.angle = S.standardized.angle00 - Orthogonal segment, same direction
False: S.standardized.angle = S.standardized.angle90 - Orthogonal segment, orthogonal direction

True: segment[i].standardized.angle = segment[i].angle00 – Non-orthogonal segment (from 3.3)

3.4. Standard Segment Equations (ABC)

Result: The expected equation (Ax+By+C) of each segment (orthogonal or not) is defined.

Compute expected standard equations of segments

For each segment[i].* as S : - Includes all segments, orthogonal or not.

S.equation.theta = S.standardized.angle
S.equation.dx = cos(S.equation.theta)/180.0*pi()
S.equation.dy = sin(S.equation.theta)/180.0*pi()

S.equation.A = round(S.equation.dy)*(-1), 1e-15)[25]
S.equation.B = round(S.equation.dx), 1e-15)
S.equation.C = round((S.equation.A * S.cx + S.equation.B * S.cy)*(-1), 1e-15)

---

[23] Lines' equations are built using weighted averages of dx, dy after having oriented all of them in reference's direction.

[24] Standardized line equations are built using average dx, dy, weighted by segments' length for each general orientation group identifier (reference.id).

[25] Rounding of A,B and C to nearest 1e-15 value is made to ensure proper processing of limit cases around 0.0

Generalize new C values to insure segments are collinear within the threshold ε

For each segment[*].* where segment[*].id.aggregate=i and segment[*].equation.theta=j as G
    group[*].id.aggregate=i
    group[*].equation.theta=j
    group[*].equation.C = sort(group[*].equation.C, ascending)
    group.reference = group[0].equation.C
    group.Id = 0

For each group[i].equation.C - Group similar C values through same group id
(abs(group.reference - group[i].equation.C) > ε)? – C value is outside the reference group
        True: A new group of C values must be created
            group.reference = group[i].equation.C
            group.Id = group.Id + 1
    group[i].id = group.Id

Compute average C value by group id

group[*].meanC = mean(group[*].equation.C, by group[*].id)

Assign new C values to each segment's equation and cleanup everything

leftJoin (segment[*] (A), group[*] (G), where A.equation.C=G.equation.C & A.equation.theta=G.equation.theta)

For each segment[i] :
segment[i].A = segment[i].equation.A
segment[i].B = segment[i].equation.B
segment[i].C = segment[i].meanC – New C value for line segments
segment[*].* = clean(Segment[*], keep=id.aggregate, id.object, id.section, orthogonalAngle, A, B, C, cx, cy)[26]

4. Coordinates Calculations

Result: New coordinate values are known for every vertex. New coordinates insure a) orthogonal angles between segments for orthogonal sections, b) collinear segments when standard topology requires it, and c) unchanged angles between segments for non-orthogonal sections.

4.1. Transfer Equations to Node

Result: Each node (vertex) knows the equations of its adjacent segments, the angular change (angle) and the angular type (orthogonal or not) which is associated to each polygon it belongs to (id.object).

Build a list of nodes from all segments and group them by distinct coordinates [27]

For each segment[i].vertex[j] – convert each segment into nodes (2)
    node[*].xy = segment[i]vertex[j].xy.
    node[*].id.object = segment[i].id.object
    node[*].angularChange = segment[i].vertex[j].angularChange

[26] Remove every variable but those mentioned in the call, for aesthetic only.
[27] All nodes are grouped together by distinct coordinates. When combined, each resulting node obtains a unique identifier and a list of all variables' values from the combined nodes.

nodes[*].* = match(node[*], by xy, id = id.node, list=(id.object,angularChange)) – Group nodes by xy.
nodes[*].* = nodup(node[*].id.object) – keeps one occurrence of node[*].id.object by node[*]

Get adjacent segments equation for each id.object[28]

nodes[*].* = touch(nodes[*].* (A), segment[*].* (B) where A.id.object=B.id.object & A.xy=B.xy)[29]

4.2. Topological Nodes

Result: Each node knows if it may be removed (*node.collinearCleaning* [true,false]) and if it belongs to an orthogonal section (*node.orthogonalSection* [true,false]).

For each nodes[i] :

Check if the nodes may eventually be removed

nodes[i].angularChange.min = min(nodes[i].node[*].angularChange)
nodes[i].angularChange.max = max(nodes[i].node[*].angularChange)
nodes[i].node.collinearCleaning = (nodes[i].angularChange.min!=nodes[i].angularChange.max)? false : true

Check if the nodes belongs to an orthogonal section

nodes[i].orthogonalSection = min(nodes[i].node[*].orthogonalAngle)=0)?[30] true : false

4.3. New Coordinates Calculation

Result: For each node belonging to an orthogonal section, intersection of adjacent lines equation is computed using homogeneous coordinates. Since equations have been settled to be orthogonal, the resulting node coordinate makes adjacent segments orthogonal. For the others, coordinates remain unchanged.

Orthogonal nodes coordinate computation

(nodes[*].orthogonalSection=true)?

True: Compute new coordinates location to create orthogonal segments

For each nodes[i] as N:

Keep/generate appropriate orthogonal line equations

N.node[*].* = Nodup(N.node[*].C) – keeps one equation by C value
(N.node[1].* = NULL | N.node[1].* = N.node[0].*)? – One equation is missing or both are similar

True: Generate the missing orthogonal equation

N.node[1].xy = N.node[0].xy
N.node[1].A = N.node[0].B*(-1)
N.node[1].B = N.node[0].A
Cx = N.node[1].A * N.node[1].x

---

[28] The structure of nodes[*] is: id. MatchedNodes, xy, segments[*].id.object, segments[*].angularType, segments[*]. A, segments[*]. B, segments[*]. C, node[*].id.object, node[*].angularChange. Segments (*segments[*].**) are paired by *id.object.*

[29] The node receives the line equation parameters (A,B,C) of all adjacent line segments (2-N).

[30] Since the domain of orthogonalAngle is [true,false], max(orthogonalAngle) is either true or false (i.e. true > false)

Cy = N.node[1].B * N.node[1].y
N.node[1].C = Cx + Cy

Generate new coordinates using Homogeneous Coordinates

N.X = NN[0].B*NN[1].C – NN[1]. B* NN[0].C
N.Y = NN[1].A*NN[0].C – NN[0]. A* NN[1].C
N.W = NN[0].A*NN[1].B – NN[1]. A* NN[0].B
N.x.new = N.X/N.W (or 0 if N.W=0)
N.y.new = N.Y/N.W (or 0 if N.W=0)

False: Non-orthogonal nodes keep their original locations

For each nodes[i] as N:
N.x.new = N.x
N.y.new = N.y

5. New Polygons Creation

Result: All polygons are rebuilt using new coordinates. All corners that were originally within λ^{o} from a right angle now show 90^{o} corners. All nodes that were near collinear (within Θ^{o} or λ^{o}) are either removed or, when they were required to keep initial polygons' topology, are exactly collinear. All polygons' segments that were near collinear (within ε m) are now exactly collinear.

5.1. Original Nodes Retriever

Result: The vertices of the original polygons are retrieved in their original order.

For each original polygon[i].vertex[j] – convert each coordinate into nodes
originalNode[j].xy = polygon[i]vertex[j].xy
originalNode[j].id = j
originalNode [j].id.object = polygon[i].id.object
originalNode [j].id.aggregate = polygon[i].id.aggregate

5.2. Collinear Vertices Remover

Result: The original polygons' vertices are matched with vertices' new coordinates without collinear vertices.

Duplicate nodes by *id.object* to recreate processed polygons' vertices

For each nodes[i].node[j].*
node[i].* = nodes[i].node[j].*

Remove collinear vertices in orthogonal sections using λ as threshold

(node[i].orthogonalSection=true & node[i].collinearCleaning=true & abs(node[i].angularChange)<=λ)?
True: remove *node[i].**

Remove collinear vertices in orthogonal sections using ϴ as threshold

(node[j].orthogonalSection=false & node[j].collinearCleaning=true & abs(node[j].angularChange)<=ϴ)?
True: remove *node[j].**

Identify all nodes that have been removed and retrieve new coordinates

LeftJoin(node[*].* (A), originalNode[*].* (B), where A.id.object = B.id.object AND A.xy = B.xy)

5.3. Object Reconstruction

Result: The original polygons are rebuilt by using the new coordinates location.

polygon[*] = BuildPolygon(xy=node[*].xy.new, by node[*].id.object, node[*].id)

Build multipolygons from their components

For each polygon[i]:
    (polygon[i].id.partOf exist)? – The polygon is a hole in a larger polygon
        True: polygon[i].id.object = polygon[i].id.partOf – inner/outer rings of the polygon set to the same id

polygon[*] = BuildMultipolygon(polygon[*], by id.objec) – build the holes of the largest polygon

5.4. Object Process Classification

Result: Add an attribute identifying if the polygon was completely orthogonalized or not.

For each polygon[i].vertex[j]:
    polygon[i].sumAngles=0
    polygon[i].sumVertices=0
    Compute angle between incoming line segments [0-180] (polygon[i].vertex[j].angularChange)

    (polygon[i].vertex[j].angularChange != 0)?
        True: sum angular changes and count the number of angular changes not equal to 0

            polygon[i].sumAngles = polygon[i].sumAngles + abs(polygon[i].vertex[j].angularChange)
            polygon[i].sumVertices = polygon[i]. sumVertex + 1

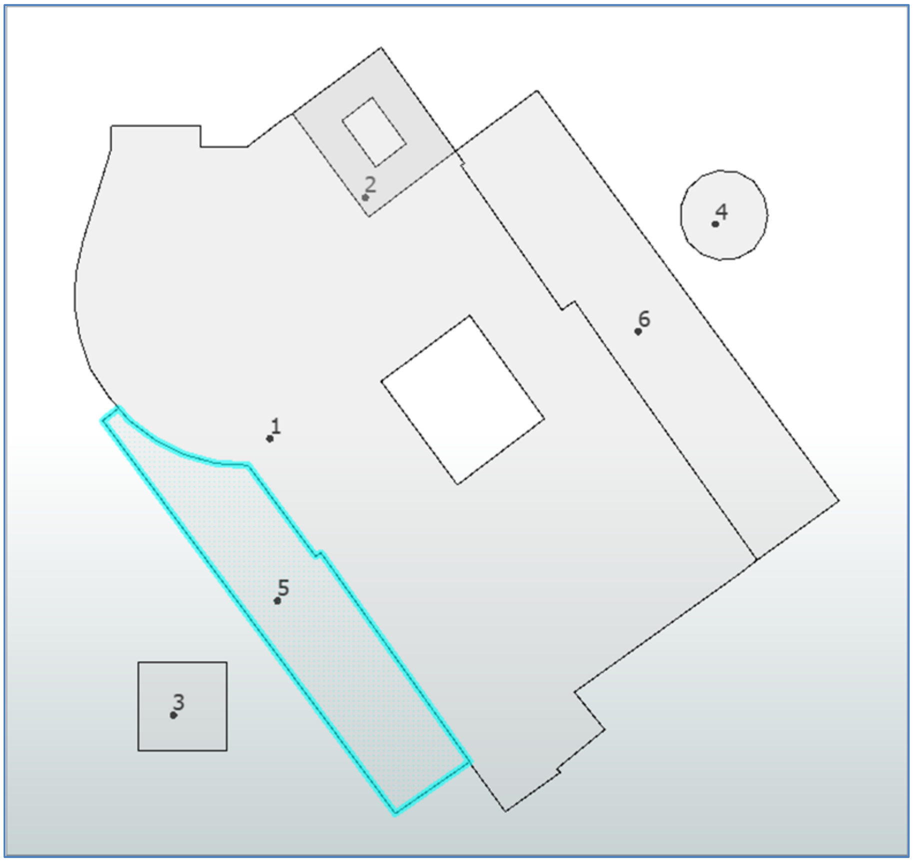
For each polygon[i].*:
    (polygon[i].sumAngles/polygon[i].sumVertices = 90)?
        True: Polygon[i].orthogonalbuilding = true
        False: Polygon[i].orthogonalbuilding = false

5.5. Attributes Retrieval & Projection

Result: All orthogonalized polygons are back to their original projection and attributes

LeftJoin(polygon[*].* (A) ,original[*].* (B), where A.id.object = B.id.object)
    remove polygon[*].id.object
    Reproject(Polygon[*].* to LL-WGS84)

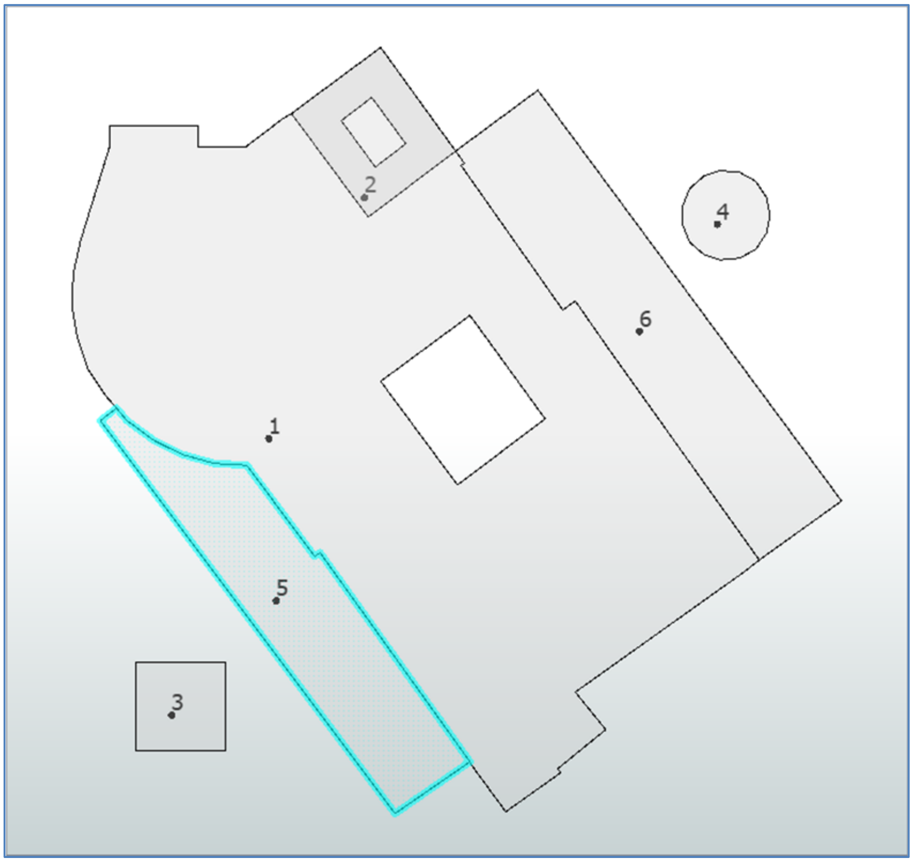
ANNEX 0: Benchmark data file in OSM format



Original OSM data file[31]

variables	values	coordinates (LL-84)		
id.object	*5*	0: 0.12332, 0.12346		
changeset	999	1: 0.12333, 0.12345		
Id	3	2: 0.12334, 0.12344		
tag{0}.k	building	3: 0.12335, 0.12343		
tag{0}.v	yes	4: 0.12337, 0.12343		
timestamp	2019-06-19T01:53…	5: 0.12339, 0.12343		
uid	101184	6: 0.12342, 0.12338		
user	jfd553	7: 0.12342, 0.12338		
version	1	8: 0.12350, 0.12328		
visible	TRUE	9: 0.12346, 0.12325		
		10: 0.12331, 0.12345		
		11: 0.12332, 0.12346		

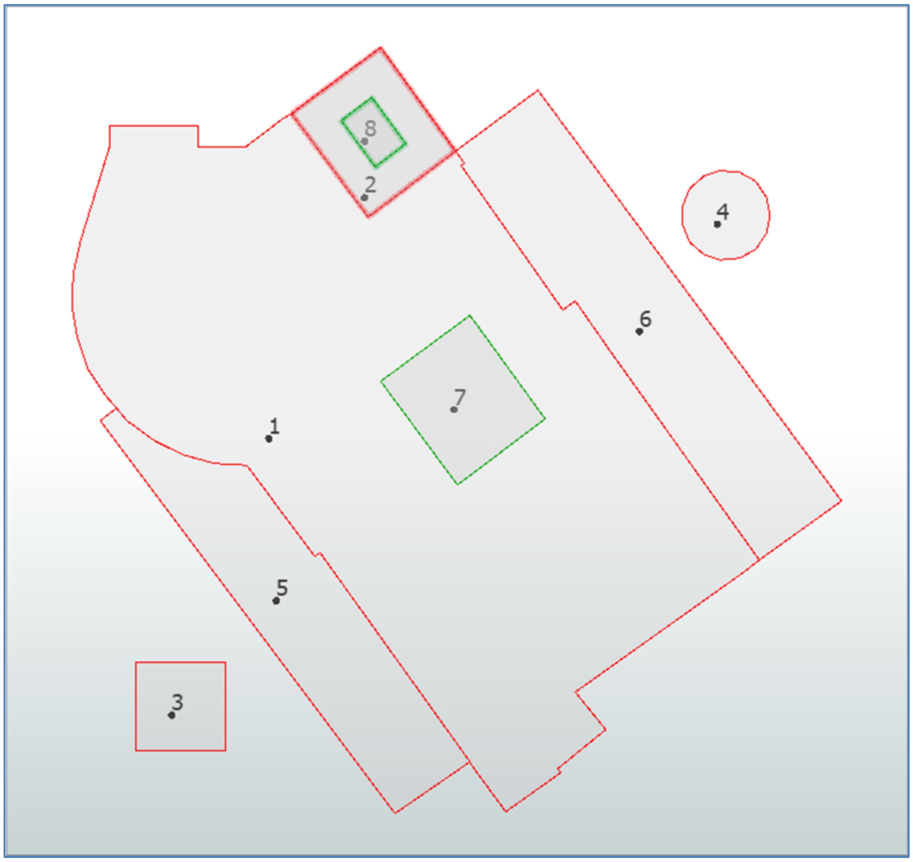
[31] Numbers show buildings' uuid (id.object)



1.1 Objects identifiers and projection -Results

variables[32]	values	coordinates (AZMED)		
Id.object	5	0: -19.04672, 1.24949		
		1: -18.45673, 0.55287		
		2: -16.95392, -0.55287		
		3: -15.35092, -1.34901		
		4: -13.54755, -1.84659		
		5: -11.75531, 1.94611		
		6: -7.94819, -7.05464		
		7: -7.64763, -6.84455		
		8: 0.74584, 18.55436		
		9: -3.45090, -21.45141		
		10: -19.94841, 0.55287		
		11: -19.04672, 1.24949		

[32] Variables names may slightly differ from algorithm description

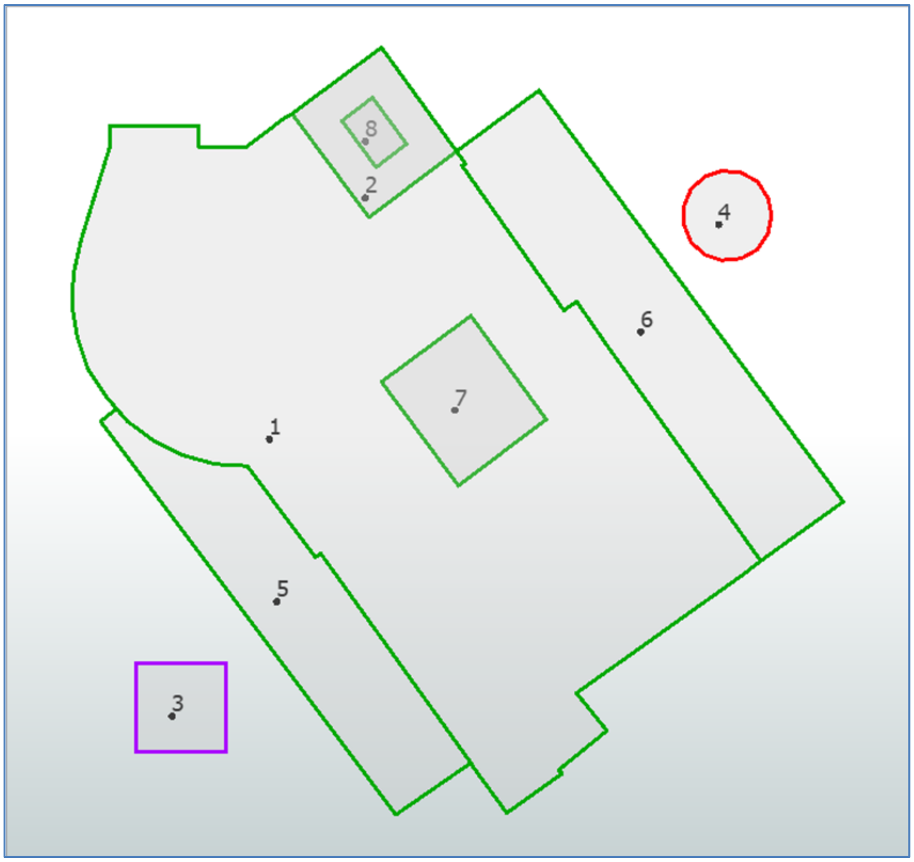


1.2.Holes Extraction -Results [33]

variables	values	coordinates (AZMED)		
Id.object	2	0: -0.05566, 15.64626		
		1: -4.25239, 21.45141		
		2: -9.25063, 17.74717		
		3: -4.95371, 11.95308		
		4: -0.05566, 15.64626		
Id.object	8	0: -4.55296, 14.75061		
id.partOf	2	1: -6.45652, 17.34910		
		2: -4.75333, 18.65388		
		3: -2.84977, 16.05539		
		4: -4.55296, 14.75061		

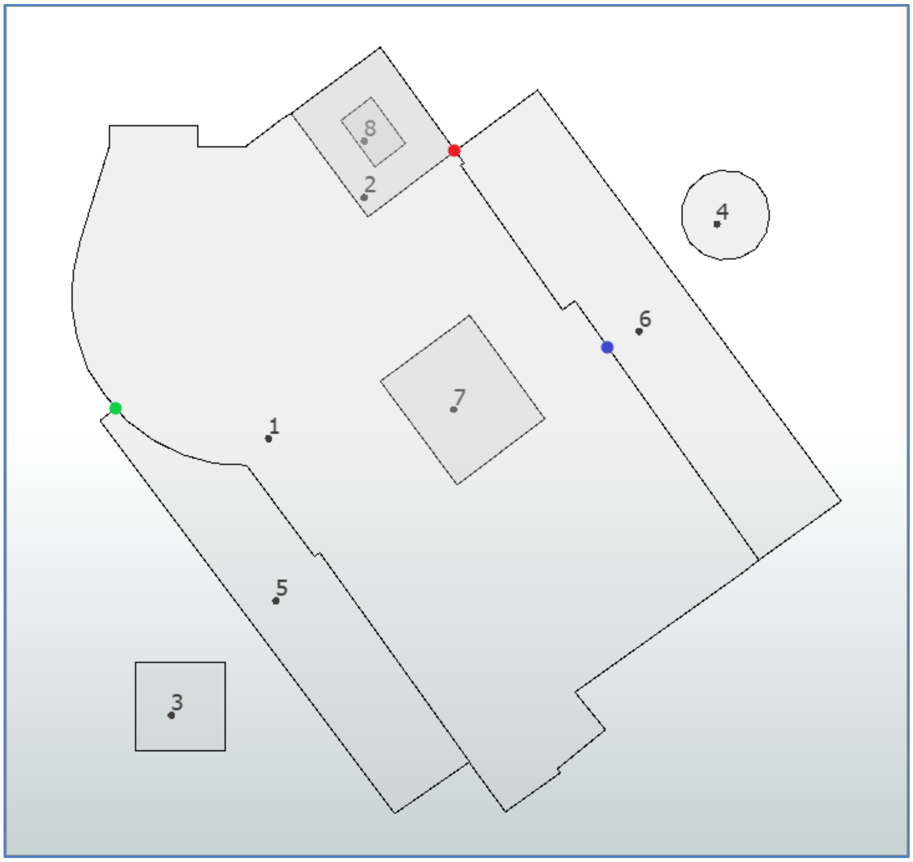
[33] Components of multipolygons are split apart to create autonomous polygons. Original polygons and multipolygon outer rings are displayed in red, inner components in green. Above selected polygons were originally a multipolygon (id.object=2). The variable id.partOf=2 is added to inner components to keep the link between outer and inner components.

ANNEX 1: Features Cleanup & Attributes Setting



1.3.Aggregate Identification -Results

variables	values			
id.aggregate	0			
Id.object	3			
id.aggregate	1			
Id.object	1			
Id.object	2			
Id.object	5			
Id.object	6			
Id.object	7			
Id.object	8			
id.aggregate	2			
Id.object	4			

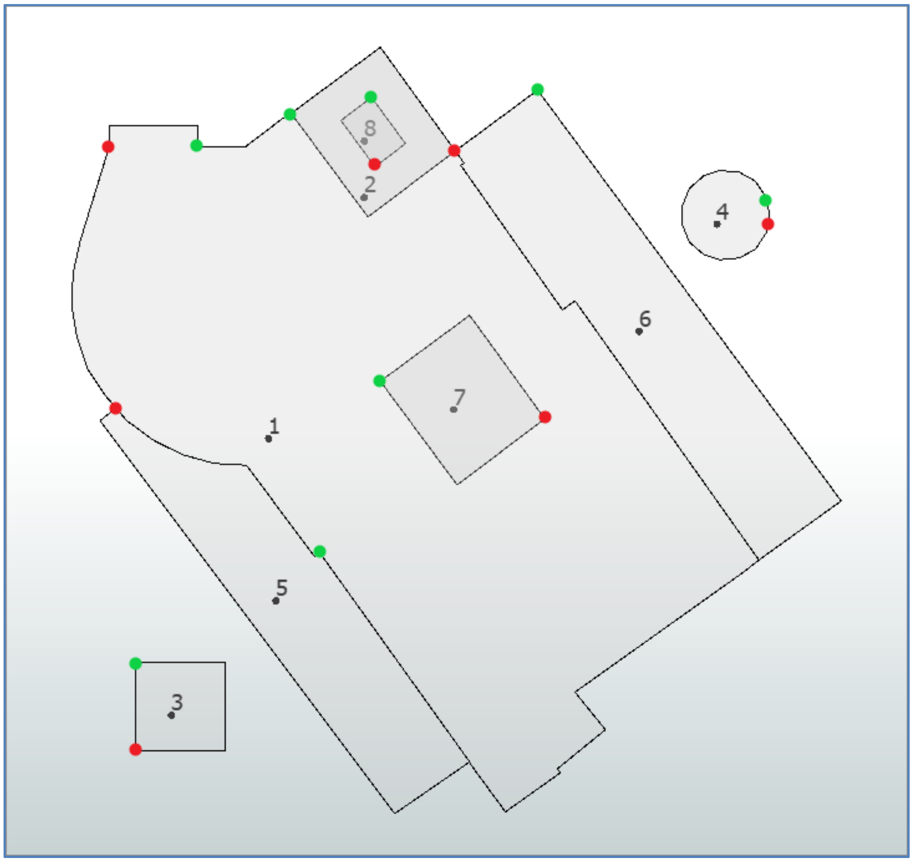


1.4.Linearize Near-Collinear Vertices -Results[34]

variables	values	Coordinates (AZMED)	Angle[35]	Comment
id.object	1	8: -0.05566, 15.64626	0.144579	Before
id.object	2	0: -0.05566, 15.64626	-88.84772	Before
id.object	6	0: -0.05566, 15.64626	89.25607	Before
id.object	1	13: 8.44913, 4.65518	-0.73643	Before
id.object	6	4: 8.44913, 4.65518	0.736432	Before
id.object	1	30: -19.04672, 1.24949	0.528968	Before
id.object	5	0: -19.04672, 1.24949	87.42618	Before
id.object	1	8: -0.05723, 15.64513	0.00000	After
id.object	2	0: -0.05723, 15.64513	-88.83188	After
id.object	6	8: -0.05723, 15.64513	89.38539	After
id.object	1	13: 8.47639, 4.67452	0.00000	After
id.object	6	4: 8.47639, 4.67452	0.00000	After
id.object	1	30: -19.04994, 1.24674	0.00000	After
id.object	5	0: -19.04994, 1.24674	87.15008	After

[34] In this case, four vertices are identified as near-collinear (3 from polygon 1 and 1 from polygon 6). Linearization not only changes these vertices but all overlapped vertices from other polygons

[35] AngularChange as described in the algorythm

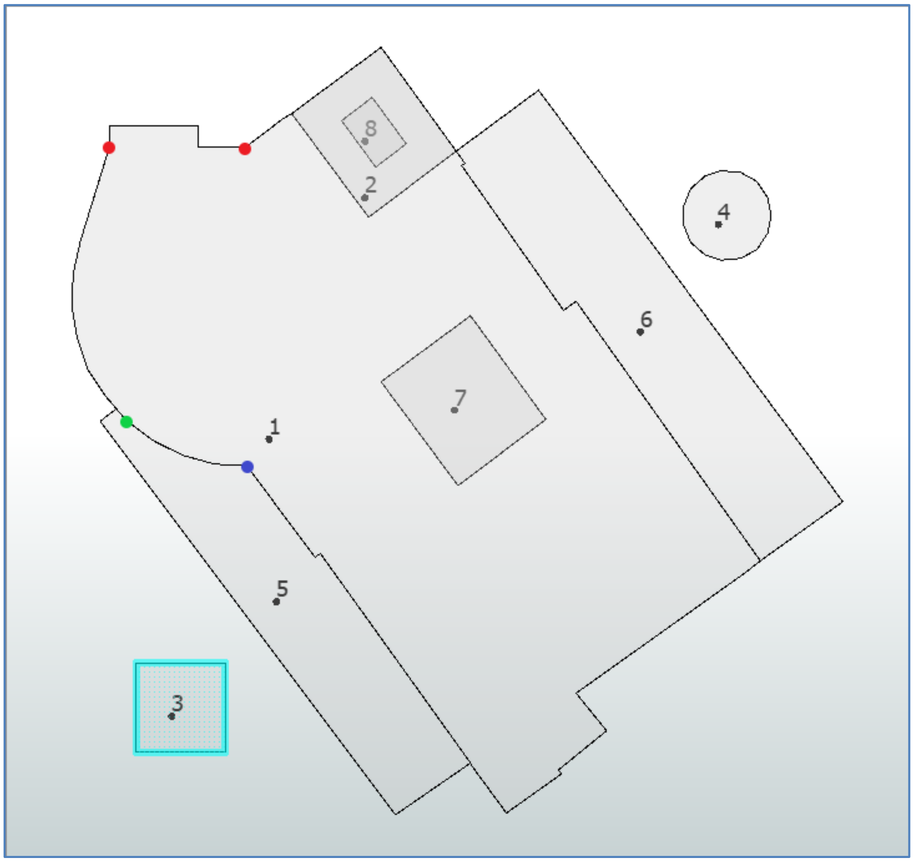


1.5.Move Polygon Origin to Most Right Angle Vertex -Results[36]

variables	values	Coordinates (AZMED)	Angle	
id.object	1	0: -19.44747, 15.84529	-17.142228	
id.object	1	0: -14.44924, 15.84529	-90.000000	
id.object	2	0: -0.05723, 15.64513	-88.831883	
id.object	2	0: -9.25063, 17.74717	-90.018286	
id.object	3	0: -17.95579, -17.94621	90.000000	
id.object	3	0: -17.95579, -12.94825	90.000000	
id.object	4	0: 17.54391, 12.05260	-22.646405	
id.object	4	0: 16.75354, 13.95447	-28.056538	
id.object	5	0: -19.04994, 1.24674	87.150078	
id.object	5	0: -7.64763, -6.84455	89.320790	
id.object	6	0: -0.05723, 15.64513	89.385394	
id.object	6	0: 4.55296, 19.05195	89.996167	
id.object	7	0: 4.95371, 0.65239	-88.951917	
id.object	7	0: -4.25239, 2.75330	-90.047937	
id.object	8	0: -4.55296, 14.75061	91.229942	
id.object	8	0: -4.55296, 14.75061	91.229942	

[36] Red dots show locations of polygons' origin before processing, green ones show new polygons' origin after processing.

ANNEX 2: Polygon Segmentation On Angular Changes Classification

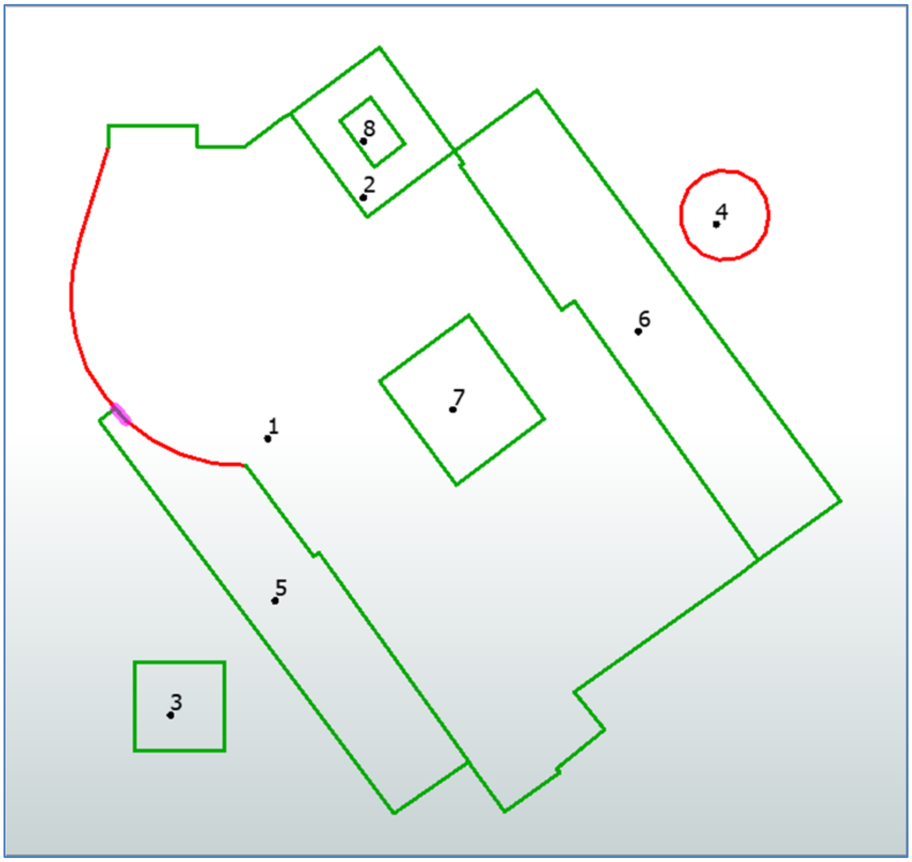


2.1 Angular Type Identification -Results

variables	values	Coordinates (AZMED)	Angle	Type[37]
Object.id	1	-19.44747, 15.84529	-17.14222	3
Object.id	1	11.85549, 15.84529	-37.30963	3
Object.id	1	-11.75531, -1.94610	50.12661	3
Object.id	5	-11.75531, -1.94610	50.12661	3
Object.id	5	-18.45672, 0.55287	-13.12692	3
id.aggregate	0	0: -17.95579, -12.94825	89.99995	0
id.object	3	1: -12.94643, -12.94825	89.99993	0
		2: -12.94643, -17.94621	90.00007	0
		3: -17.95579, -17.94621	90.00008	0
		4: -17.95579, -12.94825	89.99995	0

[37] AngularType as defined in section 2.1

ANNEX 2: Polygon Segmentation On Angular Changes Classification

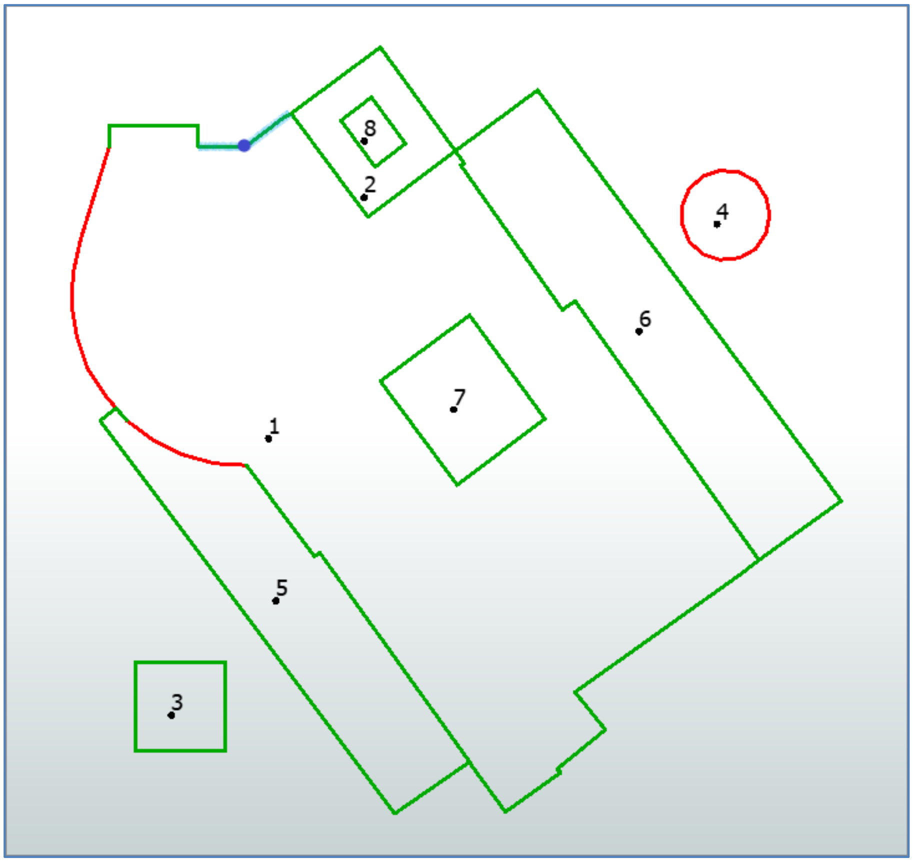


2.2 Polygon Segmentation -Results[38]

variables	values	Coordinates (AZMED)	Angle	Type
id.aggregate	1	0: -19.04993, 1.24674	0.00000	0
id.object	1	1: -18.45672, 0.55287	-13.12692	3
section.angularType	0			
id.aggregate	1	0: -19.04993, 1.24674	87.15007	0
id.object	5	1: -18.45672, 0.55287	-13.12692	3
section.angularType	0			

[38] Green segments will be orthogonalized, red segments won't be touch. In above selected segments, the segment from polygon 1 is set to be orthogonalized (section.angularType=0) because it overlaps with polygon 5 which meet orthogonal criteria at this location.

ANNEX 2: Polygon Segmentation On Angular Changes Classification

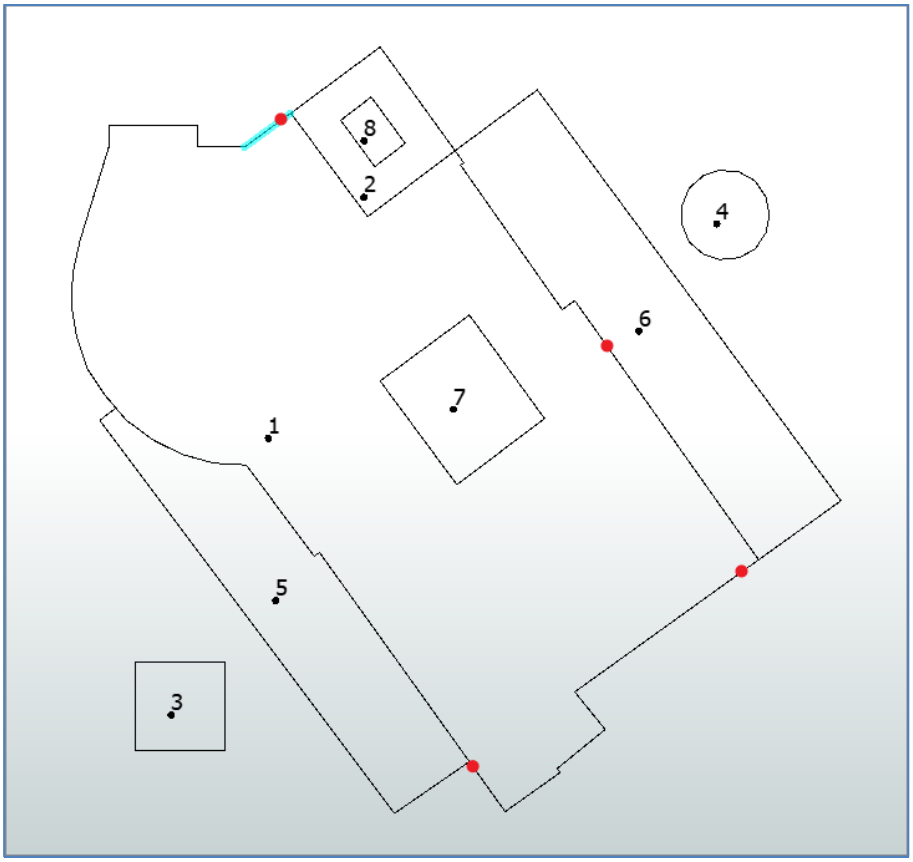


2.3 Linestrings Angular Type Topology -Results[39]

variables	values	Coordinates (AZMED)	Angle	Type
id.aggregate	1	0: -14.44923, 15.84529	-90.00002	0
id.object	1	1: -11.85549, 15.84529	-37.30963	3
id.section	1			
section.angularType	0			
id.aggregate	1	0: -11.85549, 15.84529	-37.30963	3
id.object	1	1: -9.75156, 17.44862	6.51533	0
id.section	2	-9.25062, 17.74717	-5.74819	0
section.angularType	0			

[39] Selected segments (adjacent to blue dot) do not have the same id.section because their junction (blue dot) does not have a right angle according to thresholds λ.

ANNEX 3: Segments Characteristics

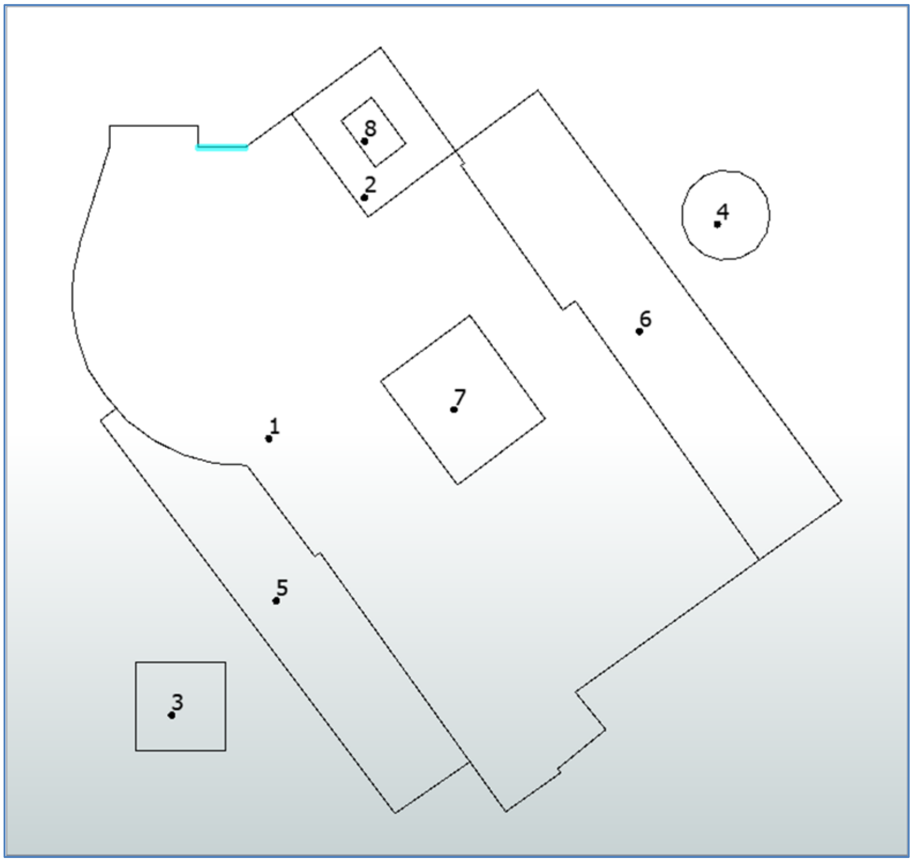


3.1 Linestrings Cleaner-Results[40]

variables	values	Coordinates (AZMED)	Angle	Type
id.object	1	1: -9.75156, 17.44862	6.515337	2
id.object	1	6: 0.84603, -18.75340	9.505838	2
id.object	1	1: 16.05223, -7.95029	2.569014	2
id.object	1	1: 8.47639, 4.67452	0.000000	1
id.object	6	1: 8.47639, 4.67452	0.000000	1
id.aggregate	1	0: -11.85550, 15.84529	-37.309636	3
id.object	1	1: -9.25063, 17.74717	-5.748197	0
id.section	2			
section.angularType	0			

[40] Collinear vertices are twofold. Vertices for which angular change is <= ϴ are removed from all segments. Vertices for which angular change is > ϴ but smaller than λ are removed only in orthogonal sections.

ANNEX 3: Segments Characteristics

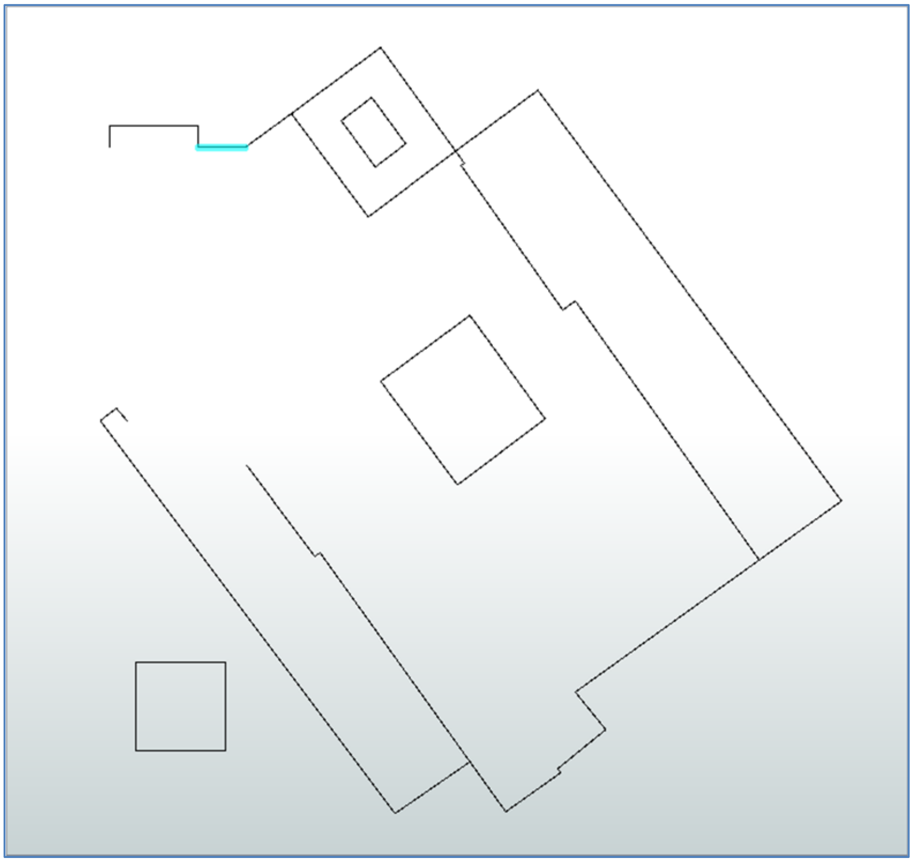


3.2 Original Segment Characteristics -Results[41]

variables	values	Coordinates (AZMED)	Angle	Type
id.aggregate	1	0: -14.44924, 15.84529	-90.00002	0
id.object	1	1: -11.85550, 15.84529	-37.30964	3
id.section	1			
section.angularType	0			
segment.angle00	-2.54651E-07			
segment.angle90	89.99999975			
segment.cx	-13.15236744			
segment.cy	15.84529449			
segment.dx	1			
segment.dy	-4.4445E-09			
segment.length	2.593738142			
segment.slope	-4.4445E-09			
segment.slopeSign	-1			

[41] Red variables result from operations described in current section, others already existed.

ANNEX 3: Segments Characteristics

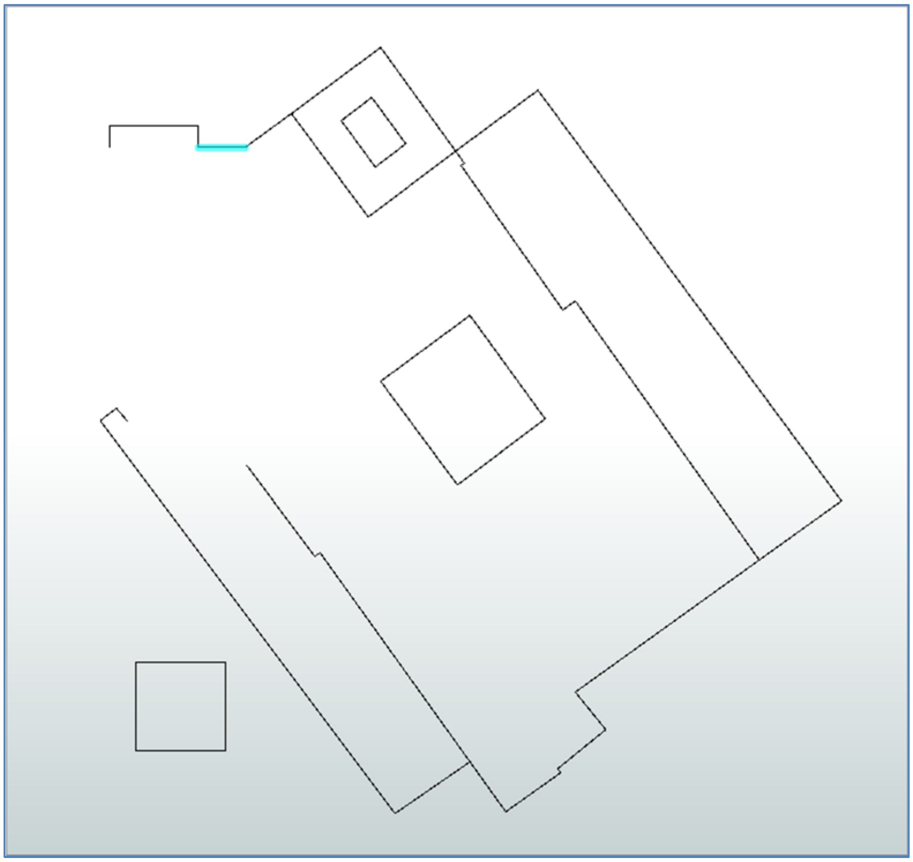


3.3.1 Compute Reference Direction By Section-Results[42]

variables	values	Coordinates (AZMED)	Angle	Type
id.aggregate	1	0: -14.44924, 15.84529	-90.00002	0
id.object	1	1: -11.85550, 15.84529	-37.30964	3
id.section	1			
reference.dx.section	1			
reference.dy.section	-5.72794E-09			
section.angularType	0			
section.length.max	4.998233584			
section.length.sum	10.00249105			
segment.angle00	-2.54651E-07			
segment.angle90	89.99999975			
segment.cx	-13.15236744			
segment.cy	15.84529449			
segment.dx	1			
segment.dy	-4.4445E-09			
segment.length	2.593738142			
segment.slope	-4.4445E-09			
segment.slopeSign	-1			

[42] Red variables result from operations described in current section, others already existed.

ANNEX 3: Segments Characteristics

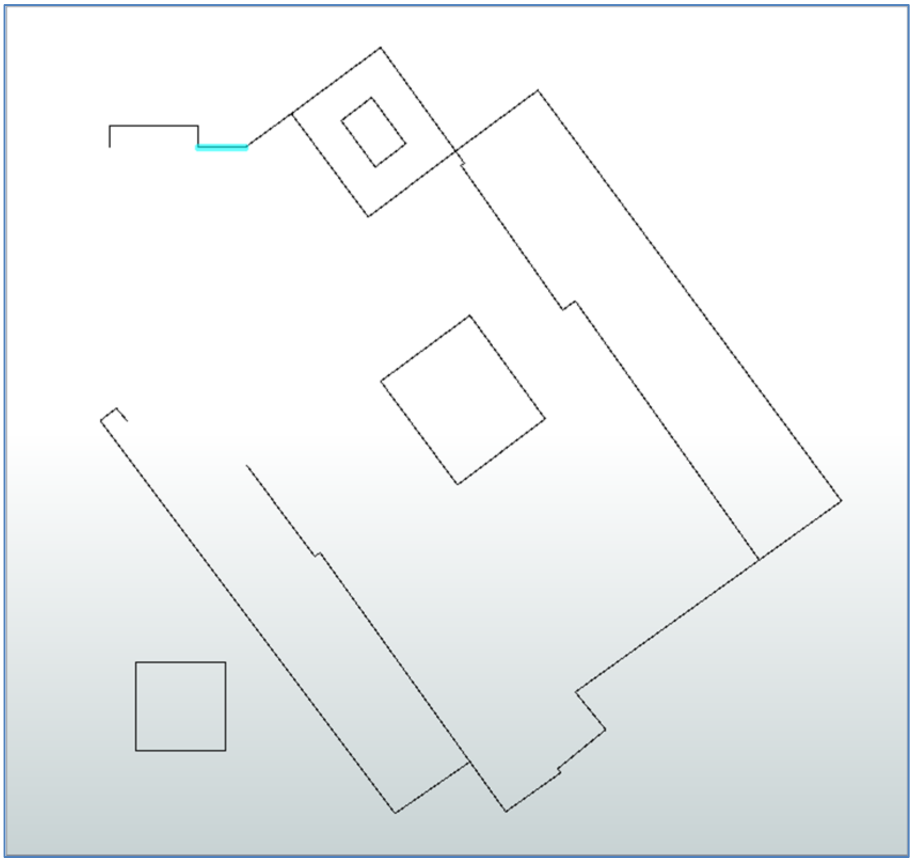


3.3.2. Compute Reference direction By Aggregate-Results[43]

variables	values	Coordinates (AZMED)	Angle	Type
aggregate.length.max	246.881148	0: -14.44924, 15.84529	-90.00002	0
id.aggregate	1	1: -11.85550, 15.84529	-37.30964	3
id.object	1			
id.section	1			
reference.dx.aggregate	0.594365133			
reference.dx.section	1			
reference.dy.aggregate	-0.804195305			
reference.dy.section	-5.72794E-09			
section.angularType	0			
section.length.max	4.998233584			
section.length.sum	10.00249105			
segment.angle00	-2.54651E-07			
segment.angle90	89.99999975			
segment.cx	-13.15236744			
segment.cy	15.84529449			
segment.dx	1			
…	…			

[43] Red variables result from operations described in current section, others already existed.

ANNEX 3: Segments Characteristics

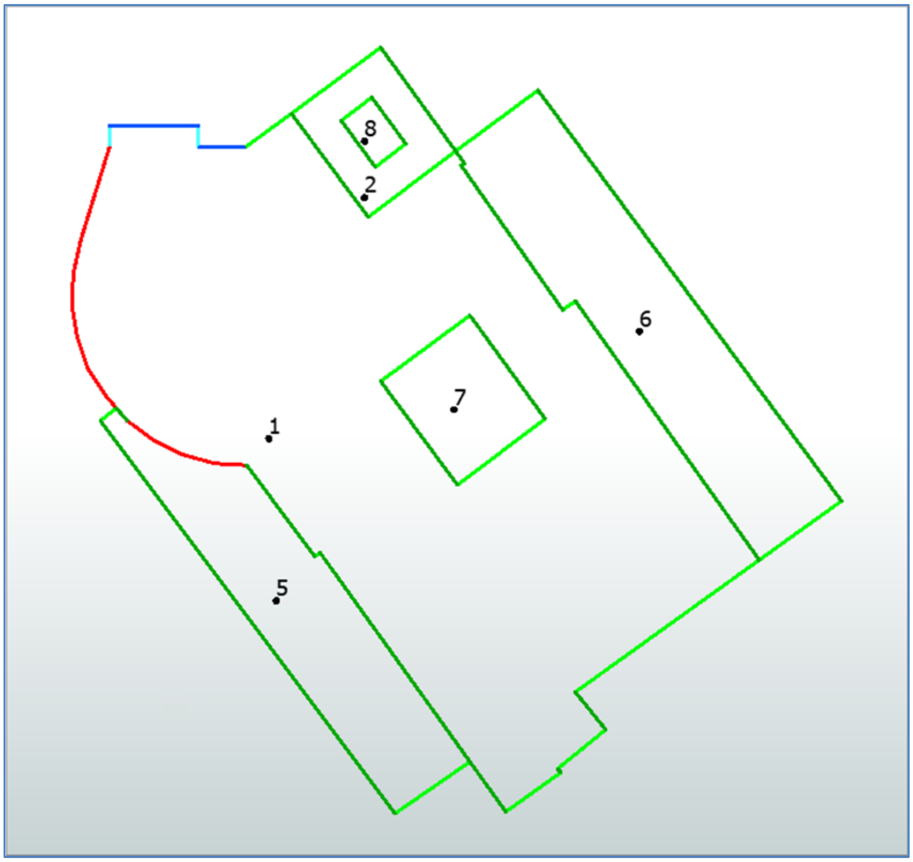


3.3.3 Associate Reference Characteristics to Segment -Results[44]

variables	values	Coordinates (AZMED)	Angle	Type
id.aggregate	1	0: -14.44924, 15.84529	-90.00002	0
id.object	1	1: -11.85550, 15.84529	-37.30964	3
id.section	1			
reference.dx	1			
reference.dy	-5.72794E-09			
reference.id	2			
segment.standardized.angle	-2.91327E-07			
segment.standardized.angle00	-2.91327E-07			
segment.standardized.angle90	89.99999971			
segment.standardized.dx	0.5180162			
segment.standardized.dy	-2.63392E-09			
segment.standardized.slope	-5.08462E-09			
segment.distance2reference	1.28344E-09			
segment.dx.weighted	2.593738142			
segment.dx.weighted.sum	5.181452401			
segment.dy.weighted	-1.15279E-08			
...	...			

[44] Red variables result from operations described in current section, others already existed.

ANNEX 3: Segments Characteristics

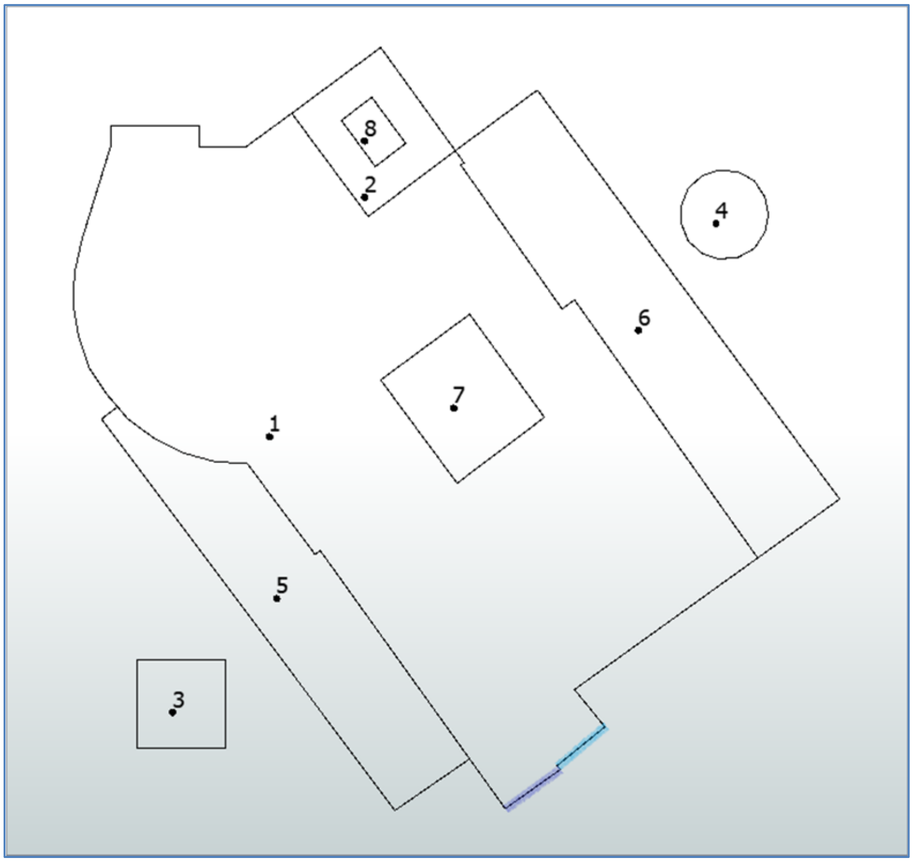


3.3 Standardized Orthogonal Segments Characteristics-Results for id.aggregate=1[45]

variables	values			
id.section	1			
id.object	1			
segment.standardized.angle	-2.91327347368e-7			
id.section	1			
id.object	1			
segment.standardized.angle	89.999999708672			
id.section	2			
id.object	*			
segment.standardized.angle	-54.0359031425963			
id.section	2			
id.object	*			
segment.standardized.angle	35.9640968574037			
id.section	6 or 8			
id.object	1 or 5			
segment.standardized.angle	*			

[45] Standardized angles associated to orthogonal segments (blue and green) show that line equations will have right angles.

ANNEX 3: Segments Characteristics

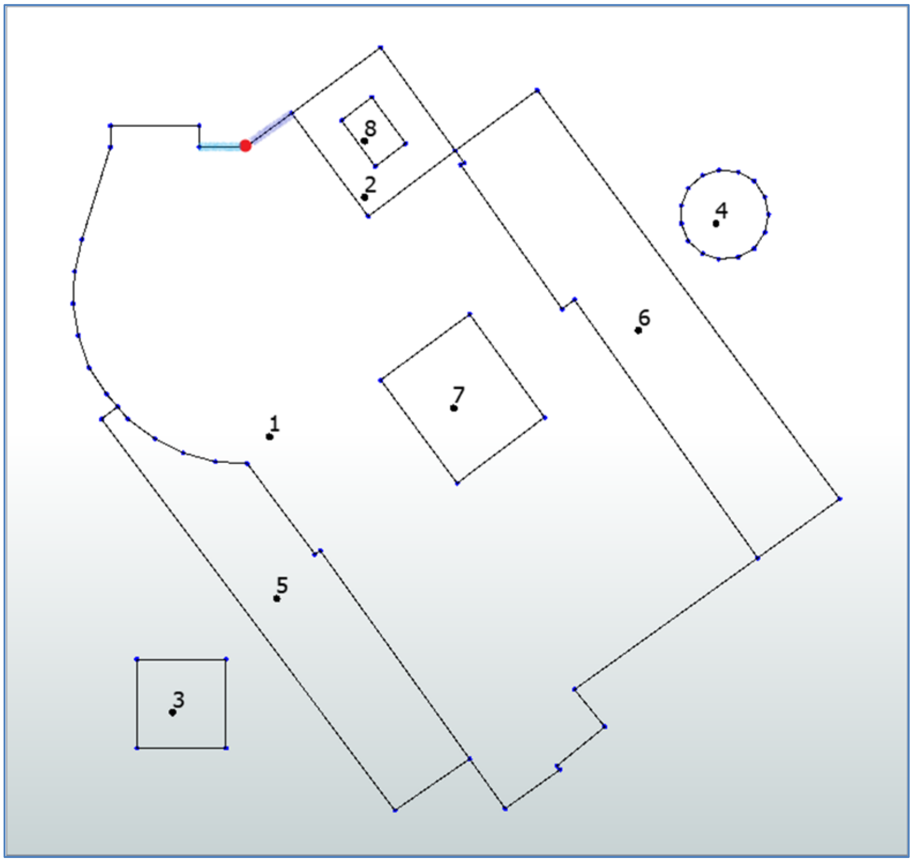


3.4 Standard Segment Equations (ABC) -Results[46]

variables	values			
id.aggregate	1	0: 8.34894, -16.75200	89.904658	0
id.object	1	1: 5.64389, -18.95243	-96.065735	0
id.section	2			
standardEquation.theta	35.96409686			
A	-0.587278184			
B	0.809385158			
C	**18.55816127**			
id.aggregate	1	0: 5.84426, -19.15147	99.778122	0
id.object	1	1: 2.74959, -21.35189	89.189149	0
id.section	2			
standardEquation.theta	35.96409686			
A	-0.587278184			
B	0.809385158			
C	**18.91489769**			

[46] Having used a threshold ε of 0.4m (instead of 0.1m) would have resulted in an identical C value for both equations (i.e. converting both lines into one). The small segment between both lines would have been transformed in collinear point and eventually removed as such.

ANNEX 4: Coordinates Calculation

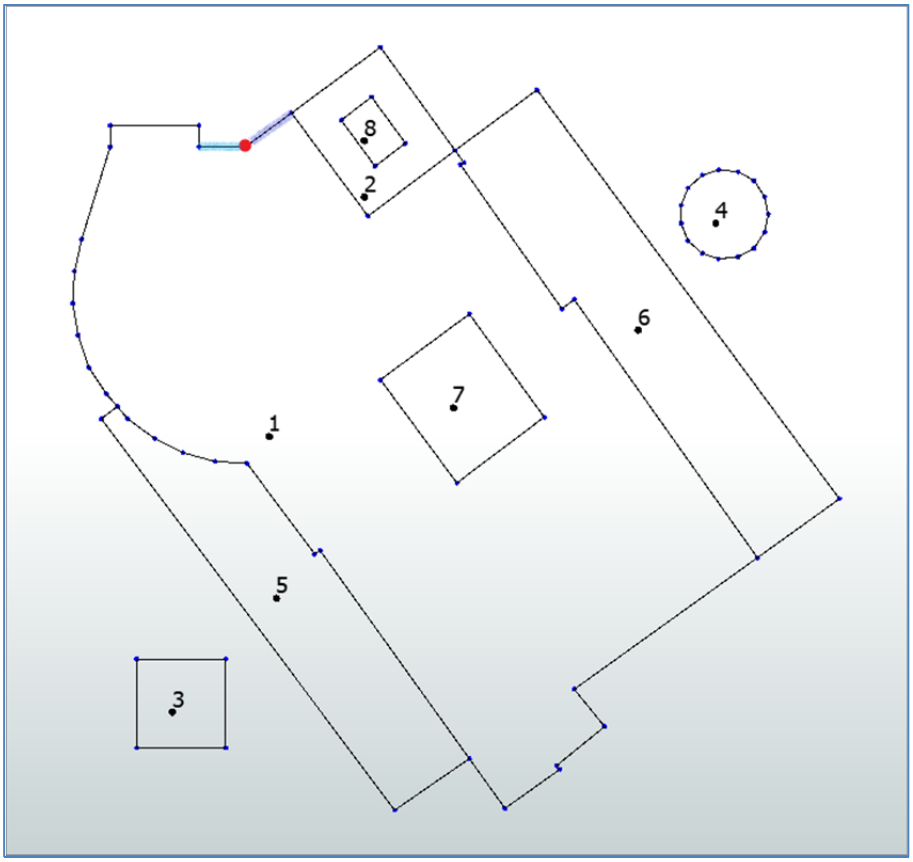


4.1 Transfer Equations to Node -Results[47]

variables	values	Coordinates (AZMED)	Angle	Type
id.Nodes	1	0: -11.85550, 15.84529	-37.30964	3
id.object	1			
Segment{0}.id.object	1			
Segment{0}.orthogonalAngle	0			
Segment{0}.A	-0.587278184			
Segment{0}.B	0.809385158			
Segment{0}.C	-19.81632922			
Segment{1}.id.object	1			
Segment{1}.orthogonalAngle	0			
Segment{1}.A	5.08462E-09			
Segment{1}.B	1			
Segment{1}.C	-15.84529443			
Nodes{0}.angularChange	37.30963586			
Nodes{0}.id.object	1			

[47] Equations from adjacent segments (2-N) are transferred to their common node (red dot), for each node/segments.

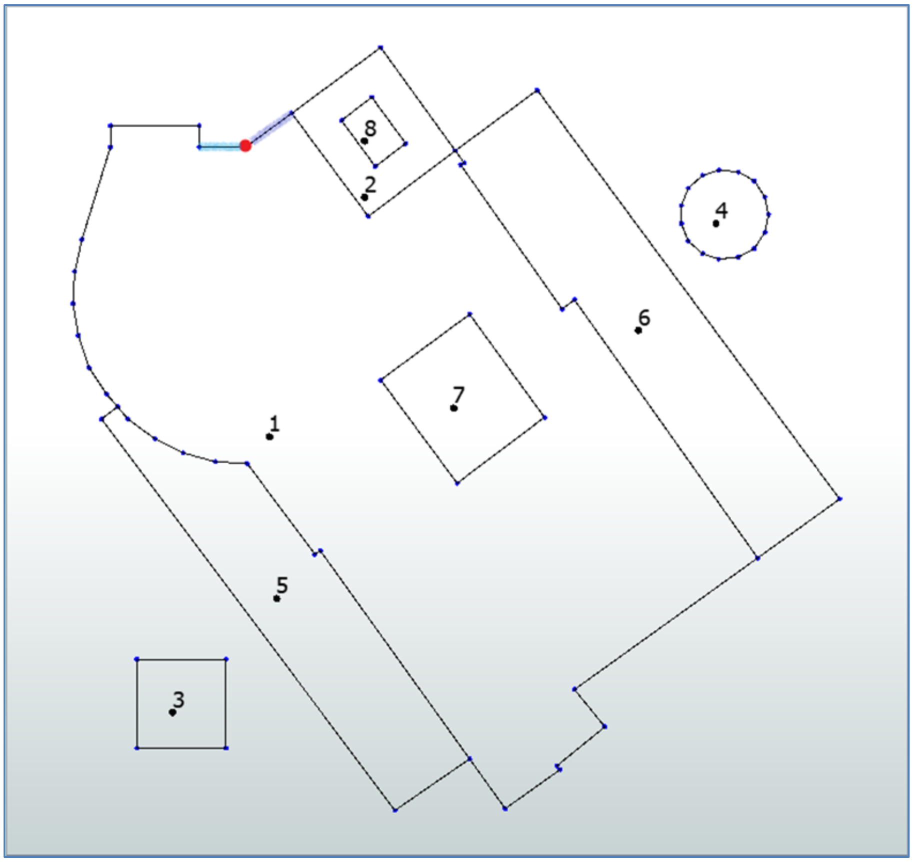
ANNEX 4: Coordinates Calculation



4.2 Topological Nodes -Results

variables	values	Coordinates (AZMED)	Angle	Type
id.Nodes	1	0: -11.85550, 15.84529	-37.30964	3
id.object	1			
node.collinearCleaning	TRUE			
node.orthogonalSection	TRUE			
Segment{0}.id.object	1			
Segment{0}.orthogonalAngle	0			
Segment{0}.A	-0.587278184			
Segment{0}.B	0.809385158			
Segment{0}.C	-19.81632922			
Segment{1}.id.object	1			
Segment{1}.orthogonalAngle	0			
Segment{1}.A	5.08462E-09			
Segment{1}.B	1			
Segment{1}.C	-15.84529443			
Nodes{0}.angularChange	37.30963586			
Nodes{0}.id.object	1			

ANNEX 4: Coordinates Calculation

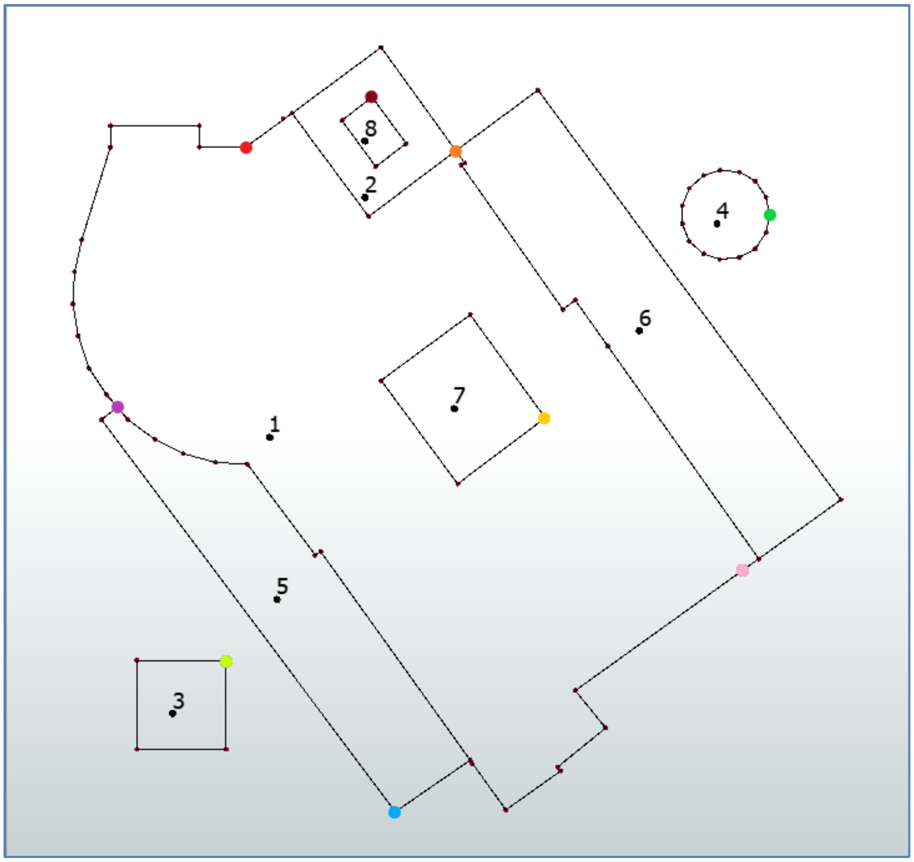


4.3 New Coordinates Calculation-Results[48]

variables	values	Coordinates (AZMED)	Angle	Type
node.collinearCleaning	TRUE	0: -11.85550, 15.84529	-37.30964	3
node.newX	-11.90472117			
node.newY	15.84529449			
node.orthogonalSection	TRUE			
Nodes{0}.angularChange	37.30963586			
Nodes{0}.id.object	1			

[48] New coordinates (node.newX, node.newY) is the intersection of adjacent line equations as computed in section 3, and transferred to node in section 4.

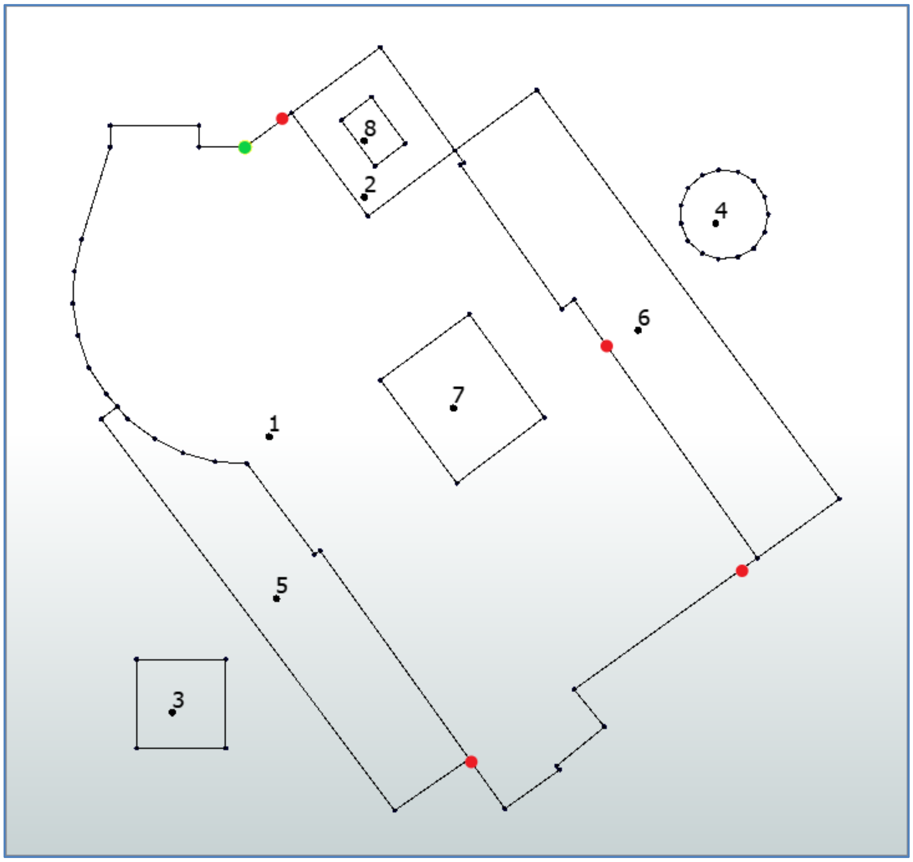
ANNEX 5: New Polygons Creation



5.1 Original Nodes Retriever -Results

variables	values	Coordinates (AZMED)	Angle	Type
id.object	1	0: -11.85550, 15.84529	-37.30964	3
id.object	1	0: -11.85550, 15.84529	0.0000000	1
id.object	1	5: -0.05723, 15.64513	0.0000000	1
id.object	2	2: -0.05723, 15.64513	-88.83188	0
id.object	6	8: -0.05723, 15.64513	89.3853937	0
id.object	3	1: -12.94643, -12.94825	89.999999	0
id.object	4	17: 17.54391, 12.05260	-22.646405	3
id.object	5	2: -3.45090, -21.45141	87.7572914	0
id.object	7	2: 4.95371, 0.65239	-88.95192	0
id.object	8	0: -4.75333, 18.65388	91.229942	0

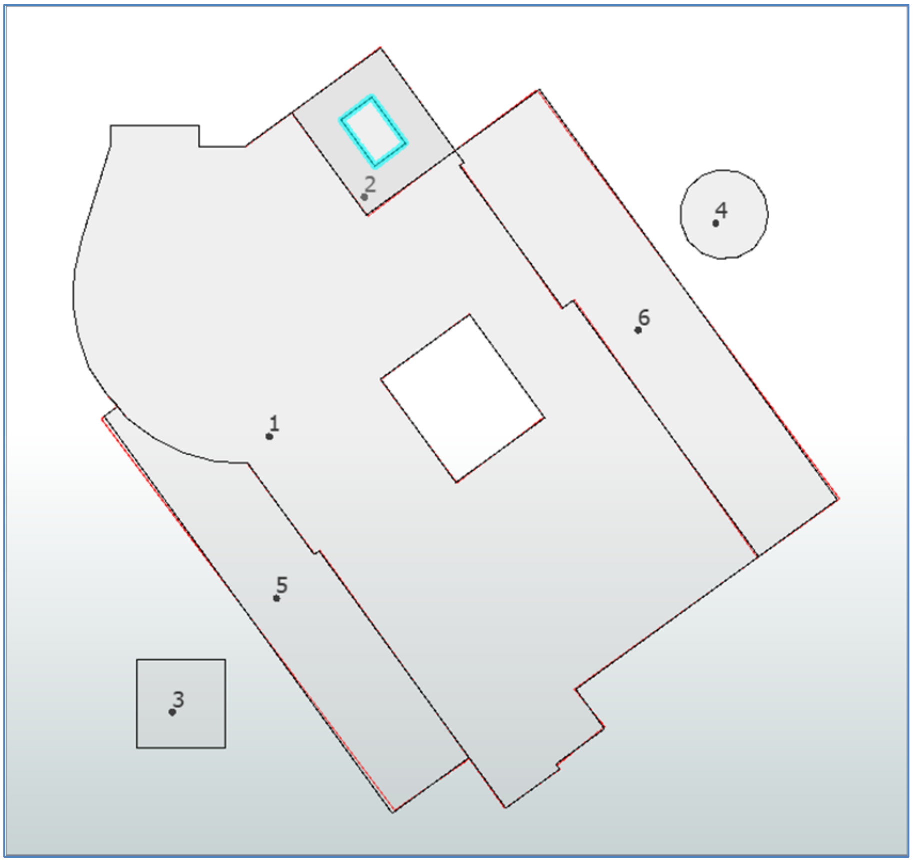
ANNEX 5: New Polygons Creation



5.2 Collinear Vertices Remover-Results[49]

variables	values	Coordinates (AZMED)	Angle	Type
id.object	1	1: -9.75156, 17.44862	6.515337	2
id.object	1	6: 0.84603, -18.75340	9.505838	2
id.object	1	1: 16.05223, -7.95029	2.569014	2
id.object	1	1: 8.47639, 4.67452	0.000000	1
id.object	6	1: 8.47639, 4.67452	0.000000	1
angularChange	37.30963586	**0: -11.85550, 15.84529**	-37.309636	3
id.aggregate	1			
id.MatchedNodes	54			
id.node	1			
id.object	1			
node.collinearCleaning	TRUE			
node.newX	**-11.90472117**			
node.newY	**15.84529449**			
node.orthogonalSection	TRUE			

[49] Original nodes that do not joint with current ones include those removed in section 3.1 and those that may have been identified in section 4.2. Old and new coordinates can be compared here.

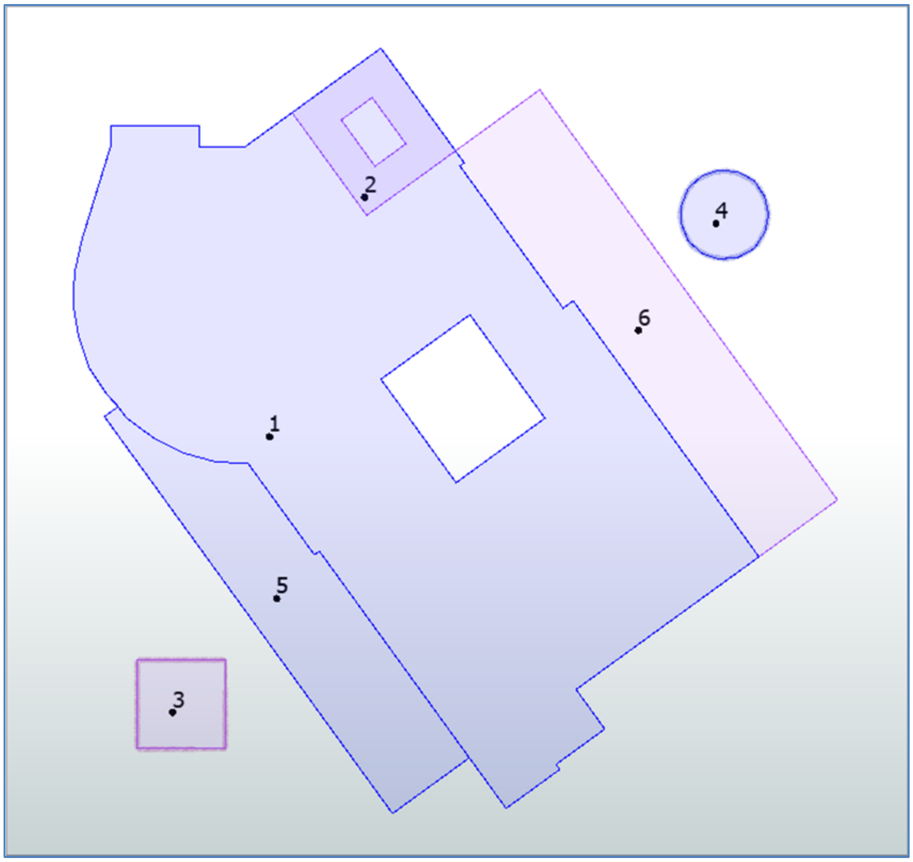


5.3 Objects Reconstruction-Results[50]

variables	values	Coordinates (AZMED)	Angle	Type
Id.object	2	0: -4.75333, 18.65388	91.229942	
		1: -2.84977, 16.05539	88.770058	
		2: -4.55296, 14.75061	91.229942	
		3: -6.45652, 17.34910	88.770058	
		4: -4.75333, 18.65388	91.229942	
Id.object	2	0: -4.73100, 18.63560	90.000000	
		1: -2.83932, 16.02849	90.000000	
		2: -4.57529, 14.76889	90.000000	
		3: -6.46697, 17.37600	90.000000	
		4: -4.73100, 18.63560	90.000000	

[50] Inner ring of polygon 2. Original inner ring coordinates and angular changes are in red, orthogonalized inner ring coordinates and angular changes are in green.

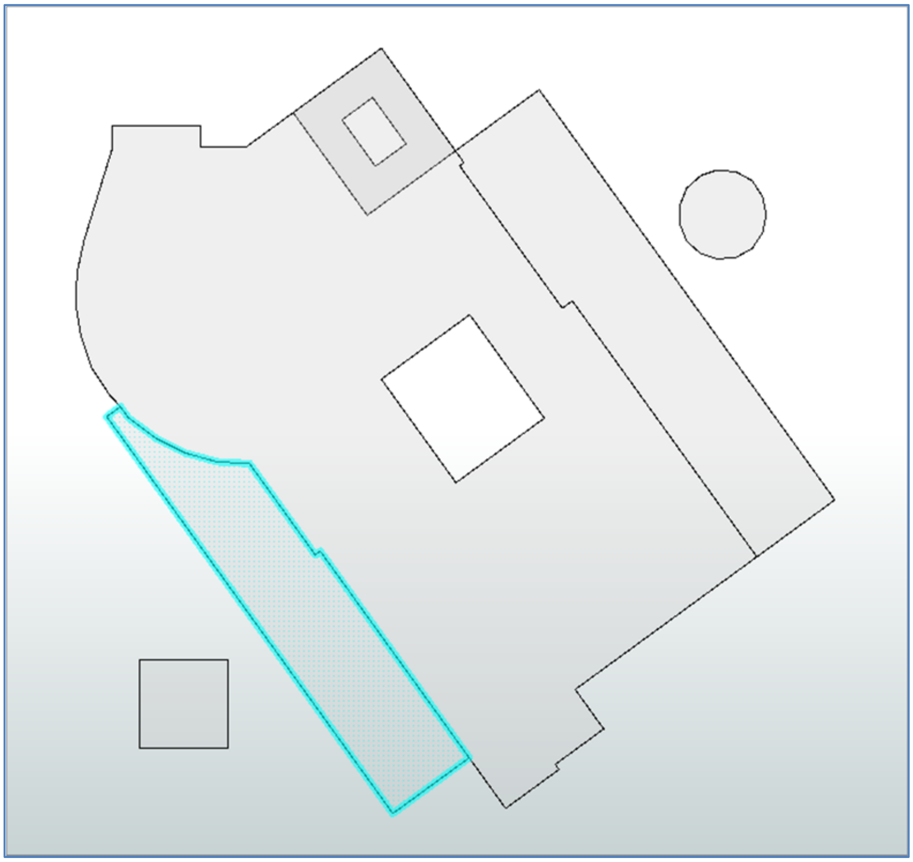
ANNEX 5: New Polygons Creation



5.4 Object Process Classification -Results

variables	values	Coordinates (AZMED)	Angle	Type
Id.object	3	0: -17.95579, -12.94825	90.000000	
orthogonalbuilding	TRUE	1: -12.94643, -12.94825	90.000000	
		2: -12.94643, -17.94621	90.000000	
		3: -17.95579, -17.94621	90.000000	
		4: -17.95579, -12.94825	90.000000	
Id.object	4	0: 16.75354, 13.95447	-28.056538	
orthogonalbuilding	FALSE	1: 15.85186, 14.45206	-23.731716	
		2: 14.74980, 14.55158	-23.479700	
		3: 13.84811, 14.25302	-23.122741	
		4: 13.04661, 13.54535	-26.623022	
		5: 12.64586, 12.55018	-21.934392	
		...	...	
		12: 17.34354, 11.04637	-22.111805	
		13: 17.54391, 12.05260	-22.646405	
		14: 17.34354, 13.04777	-21.667744	
		15: 16.75354, 13.95447	-28.056538	

ANNEX 5: New Polygons Creation



5.5 Reprojection & Attributes Retriever from OSM data file

variables	values	coordinates (LL-84)		
changeset	999	0: 16.75354, 13.95447		
Id	3	1: 15.85186, 14.45206		
tag{0}.k	building	0: 0.12342, 0.12338		
tag{0}.v	yes	1: 0.12350, 0.12328		
timestamp	2019-06-19T01:53…	2: 0.12346, 0.12325		
uid	101184	3: 0.12331, 0.12345		
user	jfd553	4: 0.12332, 0.12346		
version	1	5: 0.12332, 0.12345		
visible	TRUE	6: 0.12334, 0.12344		
		7: 0.12335, 0.12343		
		8: 0.12337, 0.12343		
		9: 0.12339, 0.12343		