Validation and Automatic Repair of Planar Partitions Using a Constrained Triangulation

Ken Arroyo Ohori   Hugo Ledoux   Martijn Meijers

This is an author's version of the paper. The authoritative version is:

Validation and automatic repair of planar partitions using a constrained triangulation. Ken Arroyo Ohori, Hugo Ledoux and Martijn Meijers. *Photogrammetrie, Fernerkundung, Geoinformation* 5, October 2012, pp. 613–630. ISSN: 1432-8364.
DOI: 10.1127/1432-8364/2012/0143

Related source code is available at:
`https://github.com/tudelft3d/pprepair`

Planar partitions are frequently used to model, among others, land cover, cadastral parcels and administrative boundaries. In practice, they are often stored as a set of individual polygons to which attributes are attached (e.g. with the Simple Features paradigm), causing different errors and inconsistencies (e.g. gaps, overlaps and disconnected polygons), which are introduced during their creation, manipulation and exchange. These errors severely hamper the use of planar partitions in other software (e.g. due to false assumptions causing erroneous calculations). Existing approaches to validate planar partitions involve first building a planar graph of the polygons and enforcing constraints, then repair is done by snapping vertices and edges of this graph. We argue that these approaches have many shortcomings in terms of complexity, numerical robustness and difficulty of implementation, and do not guarantee valid results. Furthermore, they are semi-automatic, requiring manual user intervention. We propose in this paper a novel method to validate and *automatically* repair planar partitions. It uses a constrained triangulation of the polygons as a base—which by definition is a planar partition—and only simple operations are needed (i.e. labelling of triangles) to both validate and repair. Perhaps the biggest advantage of our method is that we can guarantee that a planar partition is valid after repair. In the paper we describe the details of our method, our implementation, and the experiments we have done with real-world datasets. We show that our implementation scales to big datasets and that it offers better capabilities and overall performance than existing solutions.

1 Introduction

Planar partitions are frequently used in GIS to model concepts such as land cover, the cadastre, or the administrative boundaries of a given country. As shown in Figure 1, a planar partition is a subdivision of a polygonal subset of the plane into non-overlapping polygons. In practice, planar partitions are often represented, and stored in a computer, as a set of individual polygons to which one or more attributes are attached, and the topological relationships between polygons are not explicitly stored (shared boundaries are thus represented and stored twice). The preferred method of practitioners is representing polygons according to the *Simple Features* specification [OGC, 2006], for instance as an ESRI *Shapefile* [ESRI, 1998] or in a database, such as PostGIS[1].

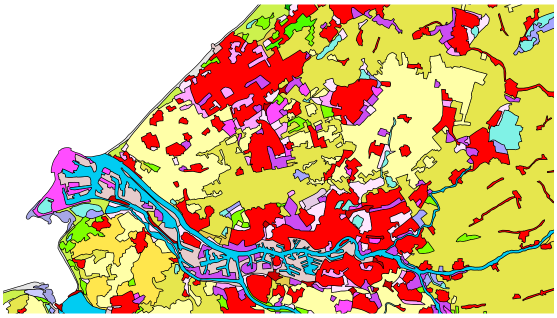


Figure 1: Part of the Corine Land Cover dataset for the region around Delft, The Netherlands.

If a planar partition is stored as a set of individual polygons, then in practice errors, mistakes and inconsistencies will often be introduced when the planar partition is built, updated or exchanged. Examples of common errors are: overlapping polygons, gaps between polygons, and polygons not connected to the others. This can be, among others, due to human error, the use of floating-point arithmetic, or limited precision [Schirra, 1997]. These errors can have catastrophic consequences for practitioners since most software and algorithms using planar partitions as input assume that this input is valid. At best erroneous results are returned, at worst it causes a software failure, often without any warning to the user. Moreover, such problems are often not visible at the scale that the data is usually viewed, exacerbating the problem [Laurini and Milleret-Raffort, 1994].

Solving that issue entails working on two related problems: (1) how to identify errors in a planar partition; and (2) how to repair these errors. As described in Section 2, both problems have been tackled in the past with the creation of a planar graph of the input. The validation, the simpler of the two problems, is usually implemented as a set of topological and geometrical constraints that the planar graph must have. As for the repair, it is usually performed by *snapping* together the vertices and edges of the graph, or by using topological information. As we discuss in Section 2, both approaches have drawbacks for users: the former method is error-prone (topological inconsistencies can be created), and the latter is only semi-automatic (and in practice real-world datasets can easily contain several hundred errors).

We present in this paper a novel method to both validate and *automatically* repair planar partitions stored according to the Simple Features specification. Our method, which is an extension of our preliminary results [Ledoux and Meijers, 2010], uses a constrained triangulation (CT) of the polygons as a support—which is by definition a planar partition—and both the validation and the repair functions are performed with relatively simple operations. These are the labelling of triangles, and standard graph traversal algorithms (such as depth-first search). Since errors are repaired by re-labelling triangles (vertices are never moved), we can guarantee that a given repair operation will preserve the topological consistency of the whole planar partition. We describe in Section 3 how the CT is used, how the polygons are labelled, how the validation is performed, and how we can automatically repair a planar partition. Moreover, we describe six different repair operations that can be used to obtain different output.

We have implemented the method in C++,

[1] http://postgis.refractions.net/

and its most relevant details are discussed in Section 4. Our software takes as input polygons stored according to the Simple Features specification, validates them, repairs them if they contain errors, and returns a new set of polygons that is guaranteed to be a valid planar partition. We also report in that section our experiments with several real-world datasets (some of them rather large), and we compare our method and its implementation to alternatives, both for validation and for repair. Finally, we discuss the advantages and disadvantages of using our method and the conclusions drawn from this in Section 5.

2 Related Work

Since a planar partition is formed by a set of individual polygons, we first discuss what a valid polygon is in our context, and then we review existing methods to validate and repair planar partitions.

2.1 Simple Features and validity of simple polygons

While there are several definitions of what constitutes a valid polygon (see van Oosterom et al. [2004] for an extensive discussion), we use in the following the international standard Simple Features [OGC, 2006], with the addition of the ISO 19107 [ISO, 2003] polygon orientation rules. Simple Features defines a polygon as follows: "A Polygon is a planar Surface defined by 1 exterior boundary and 0 or more interior boundaries. Each interior boundary defines a hole in the Polygon.". In the specification, six assertions are given that together define a valid polygon. Essential for a valid polygon is that the boundaries of the polygon must define one connected area. Additionally, a polygon can contain holes. We say that the exterior boundary of the polygon is the *outer ring*, and a hole is an *inner ring*. These holes can be filled by one or more polygons, which can recursively contain holes, which are filled by other polygons. Observe also that holes are allowed to interact with each other and the outer boundary under certain conditions, e.g. they are allowed to touch at one point, as long as the interior of the polygon stays one connected area. Each polygon is stored independently from other polygons, and it is not possible to store topological relationships between the polygons. The ISO 19107 specification [ISO, 2003] is more ambiguously defined, but it does establish orientation rules (counterclockwise for the outer ring, clockwise for the inner ones), which we use in our output.

The validation of a single polygon is possible with different libraries, GEOS[2] and JTS[3] being two widely used open-source examples.

The repair of single polygons is a less documented topic than their validation. Different software vendors offer tools to help identify and semi-automatically repair broken polygons. Examples are `ST_MakeValid()` from PostGIS and the constraints in 1Spatial Radius Topology. The method we present in this paper has been adapted to automatically repair common errors in individual polygons, e.g. wrong ring orientation, or holes that split the interior of a polygon (see Ledoux et al. [2012]). However, we focus in this paper on the validation and repair of planar partitions only and we assume that the input polygons are individually valid.

2.2 Validation of a Planar Partition Using a Planar Graph

Assuming that individual polygons have been deemed to be valid, it is possible to test the validity of a planar partition by identifying the two types of invalid configurations: overlaps and gaps.

If individual polygons are checked without building a planar graph (or an indexing structure), finding overlaps involves

[2] Geometry Engine Open Source: http://trac.osgeo.org/geos/

[3] Java Topology Suite: http://www.vividsolutions.com/jts/jtshome.htm

checking whether any possible pair of polygons overlap. This is a computationally expensive operation to make (quadratic behaviour), even when heuristics to speed up the process are used [Badawy and Aref, 1999; Kirkpatrick et al., 2002]. Additionally, robustness issues are significant in polygon intersection tests [Hoffmann et al., 1988]. Finding the potential gaps in a planar partition is even more problematic. For this, computing the union of the entire set of polygons is required, which is also computationally expensive [Margalit and Knott, 1989; Rivero and Feito, 2000].

The validation process can be sped up by first building a planar graph of the input polygons, which is afterwards checked for consistency. It should first be noticed that while different approaches are available to construct a planar graph [Shamos and Hoey, 1976; van Roessel, 1991], it is still sometimes difficult, especially if the polygon contains holes. The graph of the boundary can then be unconnected and extra machinery is necessary to represent the knowledge of holes in the graph structure. The fact that holes are also allowed to touch complicates the task of validation even further, since holes cannot be assumed to form an unconnected planar graph.

Based on this graph, Plümer and Gröger [1997] specify a list of minimal mathematical axioms that can be used to check the validity of a planar partition: no dangling edges, no zero-length edges, planarity, no holes, no self-intersections, no overlaps, and having a connected graph. It is important to note that Plümer and Gröger base their axioms on concepts from graph theory, but they also highlight the fact that a graph-based approach alone is not enough: the graph has to be augmented with geometrical knowledge (each vertex has geometry attached, i.e. the coordinates of points have to be stored). Validation is thus underpinned by both geometrical and topological concepts and systems thus have to deal with those two concepts at the same time. The method we propose in this paper —using a constrained triangulation— permits us to do exactly this: to embed both geometry and topology in the same structure.

2.3 Repair Using Point and Edge Snapping and Splitting

The most common method for planar partition repair is based on the assumption that polygons *approximately* match each other at their common boundaries. This implies that they should be within a certain distance of each other along those edges. If, additionally, all parts further apart than this value are known not to be common boundaries, it is possible to "snap" together polygons that are closer to each other than this threshold, while keeping the rest untouched. This method of planar partition repair is available in many GIS packages, including ArcGIS[4], FME[5], GRASS[6] and Radius Topology[7].

Since thresholds are central to this method, it is of utmost importance to select a good threshold value, something that is completely different in each dataset. For planar partition repair to be successful using this method, such a threshold should be chosen in a careful manner, and always comply with a few conditions. These have been summarised as follows:

1. Adjacent polygons should not be further apart than this threshold along any part of their common boundaries (shown as the maximum threshold in Figure 2(a)). Otherwise, gaps cannot be fixed.

2. Adjacent polygons should not overlap each other in areas which are further inwards than this threshold from their common boundaries (shown as the maximum threshold in Figure 2(b)). Otherwise, overlaps cannot be fixed.

3. None of the vertices of a polygon should be closer to each other than this threshold, including non consecutive vertices (shown as the minimum thresholds in Figure 2). Otherwise, they

[4] http://www.esri.com/software/arcgis/index.html

[5] http://www.safe.com/fme/

[6] http://grass.osgeo.org/

[7] http://www.1spatial.com/software/radius_topology/

might be snapped together, creating repeated vertices, disjoint regions, or various topological problems.

4. None of the vertices of a polygon should be closer than this threshold to any non incident edge. Otherwise, they might be snapped together, creating disjoint regions or various topological problems.

This threshold value is usually manually determined, either by trial and error, or by analysing certain properties of the dataset(s) involved (e.g. point spacing, precision, or map scale). However, it is often hard to find an optimal threshold for a certain dataset, since ensuring that it works well for every part of a dataset is unrealistic. Moreover, sometimes such a threshold does not even exist (e.g. because point spacing in some places might be smaller than the width of the gaps and overlaps present).

Since the aforementioned conditions are frequently not met or are not checked beforehand, and it is still necessary to perform repair of a dataset, snapping is often performed nevertheless, possibly creating invalid polygons and/or planar partitions, or significantly changing the topology of the existing features. Two examples of this phenomenon are shown in Figures 3 and 4.

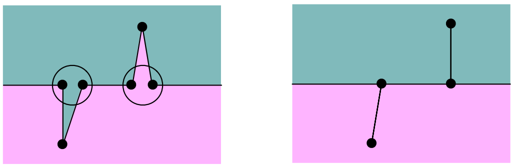

(a) Before snapping
(b) After snapping

Figure 3: Spikes and punctures can be created by snapping, since the bases of these elongated forms (encircled) might be narrower than the threshold, but its length not.

While these examples show that snapping is not problem-free, it is important to note that commercial GIS packages often implement more complex snapping options (such as point-to-edge, edge-to-edge, or using a reference dataset). These options can help to solve a problematic case, but can also have undesired consequences, such as changing the topology of the polygons. Another problem is that post-processing operations to clean resulting polygons might be required (e.g. disposing of polygons with small areas, removing redundant lines, thresholds for minimum angles, etc.), which could again create invalid configurations, requiring iterative validation or repair processes.

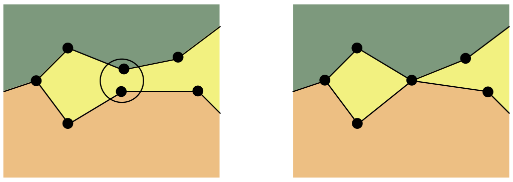

(a) Before snapping
(b) After snapping

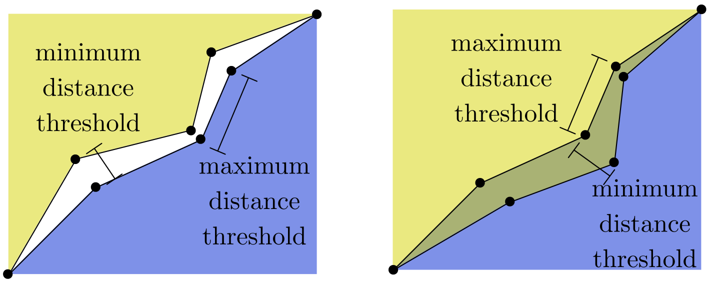
Figure 4: Polygons can be split by snapping, since some parts of them might be narrower than the threshold (encircled). While this result does not create an invalid planar partition, it can change the number of polygons present and their topological relations, and can therefore be undesirable.

2.4 Repair Using Topological Information

A different approach for planar partition repair, based on topological information, is available in some software.

GRASS also creates a graph, using edges as a base structure instead of triangles, and could be used to (manually) detect overlaps and gaps based on the number of labels using the functions `v.what` and `d.what.vect`. However, there is no simple automated procedure to get the number of labels at a certain point, which makes it very cumbersome and time consuming to use GRASS for this purpose.

Meanwhile, ArcGIS provides a more complete solution, using a method similar in some ways to the one developed and described in Section 3. It involves using the



(a) Gaps between polygons (b) Overlapping polygons

Figure 2: Defining a threshold for vertex, edge and face snapping. The threshold to use should be larger than the largest minimum distance between the matching boundaries, and smaller than the minimum distance between vertices.

Geodatabase feature of the software with some combined validation rules (e.g. must not overlap and must not have gaps). However, fixing every problematic area in an appropriate manner requires extensive user intervention (see Figure 5), since the best choice for each case depends on the specific configuration of the error.

Since both the aforementioned programs do not offer an automated process to correctly solve this problem (and GRASS lacks the ability to visualise problem areas), they are not really comparable to our solution. A planar partition can easily contain tens of thousands of polygons, possibly generating thousands of errors, which need to be checked and repaired semi-automatically.

3 Validation and Automatic Repair Using a Constrained Triangulation

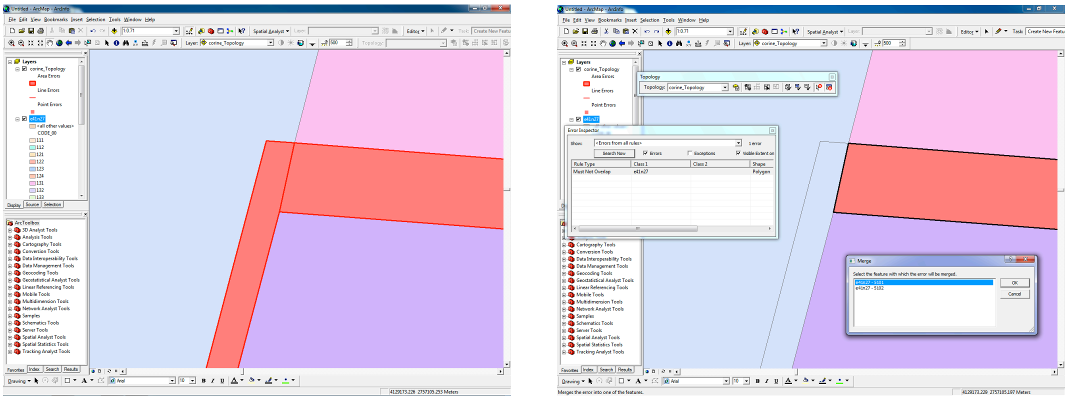
Our approach to validation and automatic repair of planar partitions uses a constrained triangulation (CT) as a supporting structure because it has many good properties, including the following:

- It is by definition a planar partition. Therefore, as long as we keep the information about which polygon each triangle belongs to, the reconstructed polygons will be either a valid planar partition, or multiple ones.
- It can be built quickly, in $O(n \log n)$ with a variety of approaches [Guibas and Stolfi, 1985; Mücke et al., 1999; Clarkson et al., 1992][8].
- Changes to the triangulation (e.g. adding a new constrained edge) are local, and therefore fast.
- Constrained edges can usually be added in constant time, being only significantly slower (and more complex) when there is an intersection with an existing constrained edge [Shewchuk, 1997b].
- Implementation-wise, several stable and fast triangulation libraries exist, including CGAL [CGAL, 2011], Triangle [Shewchuk, 1997a] and GTS [GTS, 2006].

The general workflow of our approach to both validate and repair a planar partition is as follows:

1. the CT of the input segments forming the polygons is constructed;
2. each triangle in the CT is labelled with the label of the polygon inside which it is located;
3. problems are detected by identifying triangles having no or multiple labels,

[8] The actual computational complexity can be $O(n \log n + k)$, with k being the number of edge-edge intersections, which could conceivably even be n^2. However, $k \ll n$ for most GIS datasets.



(a) Viewing a topology error in ArcGIS.

(b) Assigning an overlapping region to one of the polygons involved.

Error Inspector

Show: <Errors from all rules> 1 error

Search Now ☑ Errors ☐ Exceptions ☑ Visible Extent on

Rule Type	Class 1	Class 2	Shape
Must Not Overlap	e41n27		Polygon

Merge

Select the feature with which the error will be merged.

e41n27 - 5101
e41n27 - 5102

OK
Cancel

Merges the error into one of the features.

4129173.229 2757105.197 Meters

Figure 5: Planar partition repair in ArcGIS. The user is expected to zoom in to a particular error, analyse the situation (e.g. by looking at the properties from the surrounding polygons), and make a decision to assign the problematic region to a certain polygon. More than 11 000 errors were detected in this tile of the Corine dataset.

and by verifying the connectivity between triangles;

4. repairing of the problems is made by relabelling triangles to ensure that each triangle has exactly one label;
5. extracting the polygons from the triangulation (polygons modelled with the Simple Features specification).

As mentioned previously, for this workflow we assume that each input polygon is individually valid. We describe in the following section the concepts needed and we give a detailed description of the different steps.

3.1 Triangulation of a polygon and constrained triangulation

A triangulation subdivides an area into non-overlapping triangles. Using a constrained triangulation, every line segment that defines the boundary of a polygon, is ensured to appear as an edge in the triangulation. It is known that any polygon (also with holes) can be triangulated without adding extra vertices [de Berg et al., 2008; Shewchuk, 1997a]. Figure 6 shows an example.

In our approach, the triangulation is performed by constructing incrementally a CT of all the segments representing the boundaries (outer + inner) of each polygon. If the set of input polygons forms a planar partition, then each segment will be inserted twice (except those forming the outer boundary of the set of input polygons). This is usually not a problem for triangulation libraries because they ignore points and segments at the same location (as is the case with the solution we use, see Section 4). When segments are found to intersect, they are split with a new point created at the intersection. This is the only situation in which the generation of new points is required.

Notice that our approach requires only a constrained triangulation, and not one that fulfils the Delaunay criterion [Shewchuk,

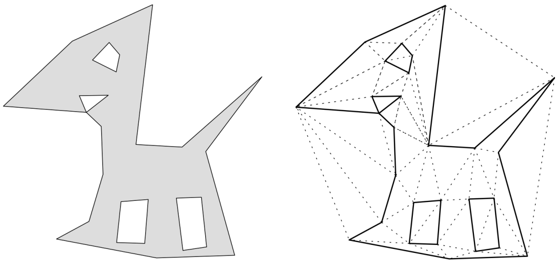


Figure 6: (**a**) A polygon with 4 holes. (**b**) The constrained triangulation of the segments of this polygon.

1997a]. However, having well-shaped triangles is useful for repair purposes and does not significantly increase the processing time. Therefore our implementation actually constructs and uses a constrained *Delaunay* triangulation.

3.2 Labelling the triangles of a Planar Partition

Labelling a triangle means assigning it label(s) for the polygon(s) that it belongs to (if two input polygons overlap, each triangle in the overlapping region should have the labels of the two polygons).

In our previous work on validation [Ledoux and Meijers, 2010], we used the centroid of a polygon to start the labelling process, but this method is prone to errors (if for instance the calculated centroid is outside or on the boundary of the polygon) and does not permit us to differentiate gaps from overlaps.

To solve these problems, we store information about the constrained edges of the CT. Since it is known that the input rings are closed and have a known orientation (according to the ISO 19107 orientation rules), it is also known on which side of a certain line segment the interior of the polygon lies. This property is used for robust labelling of each polygon. Triangles adjacent to the outer ring of the polygons are labelled first, and this is later expanded to triangles further in the interior of the polygon, recursively labelling adjacent unlabelled faces as long as no constrained edges are crossed. After this operation, all triangles that are part of any polygon are labelled, with overlapping regions having multiply labelled triangles. However, holes are then indistinguishable from triangles outside the planar partition, since both have zero labels. Therefore, a special label is created for all the triangles outside the planar partition, referred to as the “universe” label, which are labelled by recursively labelling adjacent triangles from any triangle known to lie in the exterior of the planar partition. We exploit the concept of the “far-away point” to achieve this [Liu and Snoeyink, 2006]; which is used by several implementations and is also known as the “big triangle” [Facello, 1995].

3.3 Validation

If the set of input polygons forms a planar partition then all the triangles will be labelled with one and only one label. The problems are easily detected:

- **Gaps** are detected by finding triangles without any labels.
- **Overlaps** are detected by finding triangles with two or more labels.
- **Disjoint regions** are detected by identifying regions separated by the “universe” label. This is done by starting at a given triangle, and doing a breadth-first search on the dual of the triangulation, without visiting the triangles labelled as “universe”. If all the triangles can be reached, then no polygons of the planar partition are disjoint.

3.4 Repair Operations

The greatest benefits of using a labelled triangulation for planar partition repair stem from the fact that while repair operations are performed, the validity of the planar partition is always kept, together with the integrity of the data. Unlike with snapping, vertices are not required to be moved during the process, and unlike snapping, repair is performed using *local* criteria, instead of global ones (in snapping, the threshold is usually fixed for the whole dataset). This comes as a contrast to other methods, where care needs to be taken to ensure that the (geometric or topological) validity is kept.

Figure 7 shows the standard steps required in a repair operation. In order to avoid order dependency when repairing, the repair operation itself is always performed after all the choices for which label to assign have been made.

In particular, we propose six different repair operations that can be used to fix gaps

Table 1: The repair operations currently implemented in our software. The types of operations are defined in the map algebra classification [Tomlin, 1994].

Repair operation	Type	Criteria
Triangle by priority list	Varies	Label that has the highest priority according to a predefined priority list
Triangle by number of neighbours	Focal	The label present in the largest number of adjacent faces, overlaps included.
Triangle by absolute majority	Focal	Label present in two or more valid adjacent faces
Triangle by longest boundary	Focal	Label present along the longest portion of the boundary of the adjacent faces
Regions by longest boundary	Focal of zonal	Label present along the longest portion of the boundary of the adjacent faces
Regions by random neighbour	Focal of zonal	Random label from the adjacent faces

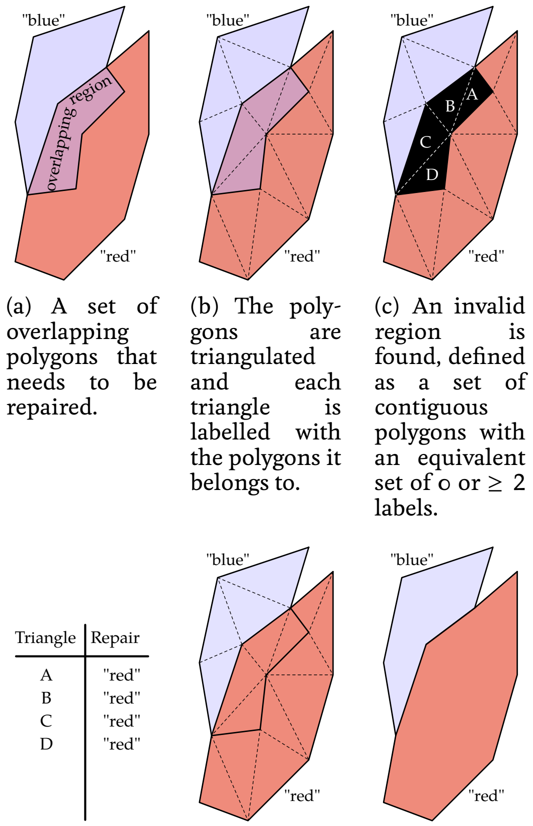
and overlaps. These are shown in Table 1 and all imply re-labelling triangles. Four of them use triangles as a base (i.e. the label assigned is based only on that of its three neighbouring triangles), which is fast and modifies the area of each input polygon the least. Despite their simplicity, they offer substantial control over the results. For instance, the first two operations only differ from each other in their handling of overlapping faces; but *triangle by number of neighbours* is better for large overlapping regions, while *triangle by absolute majority* is better in fixing small problems. Having well-shaped (Delaunay) triangles is most useful for triangle-based repair functions. Two of them use regions of adjacent triangles with equivalent sets of labels, which is slower than a triangle-based method but yields results that can be cartographically more pleasing. An interesting repair operation for practicioners is the one in which a *priority of labels* is used, i.e. in case of gaps/overlaps the labels in the triangle (overlap) or in the adjacent polygons (gap) are ordered according to a user-defined priority, and the highest priority is assigned to the problematic triangles. Notice that these repair operations can be used one after the other (in a hierarchical manner), for instance if first the repair according to the longest boundary is used but one zone has two or more boundaries with exactly the same length, then the deadlock can be solved by choosing one randomly. A sample of the results obtained with different repair operators is also shown in Figure 8.

More repair operations based on extensions to the idea of labelling triangles/regions can be further developed. For instance, triangles could be split to subdivide an area with problems (as in Bader and Weibel [1997]), or sliver triangles/regions could be discarded (and then filled during repair).

3.5 Extraction of Polygons From a Triangulation

Starting from a labelled triangulation, it is possible to reconstruct the individual polygons of a planar partition to conform to valid polygons according to the ISO 19107 and the Simple Features specifications, which allows users to incorporate automatic validation and repair in their workflow.

We do this operation polygon by polygon. We start at an unprocessed triangle and visit all the connected triangles having the same label (to reconstruct the polygon), marking the triangles as processed as we visit them. Note that since all these triangles are connected, the outer and inner boundaries of a polygon are all simple (non self-



Triangle	Repair
A	"red"
B	"red"
C	"red"
D	"red"

(a) A set of overlapping polygons that needs to be repaired.

(b) The polygons are triangulated and each triangle is labelled with the polygons it belongs to.

(c) An invalid region is found, defined as a set of contiguous polygons with an equivalent set of 0 or ≥ 2 labels.

(d) The neighbours of the region are analysed to see how it should be re-labelled, and the result is saved onto a list.

(e) The triangles are relabelled according to the list.

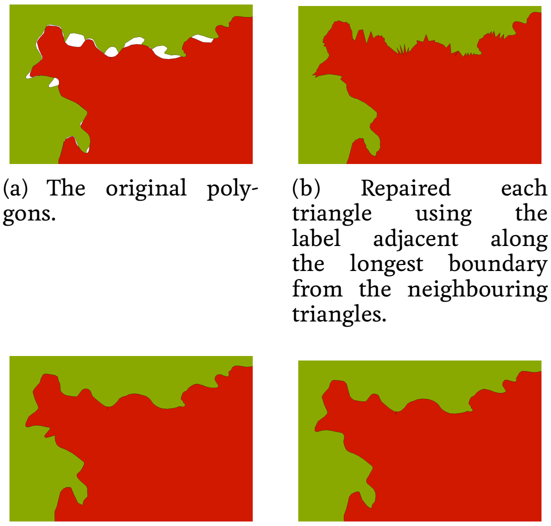
(f) The output polygons, which now have no gaps or overlaps.

Figure 7: The steps in a generic repair operation.

intersecting). We repeat this operation until all the triangles have been processed.

For each polygon, we have to recover not only its outer boundary, but also its inner boundaries, which are not connected. Observe that we cannot simply follow the original constrained edges as these do not have any meaning after a planar partitions was repaired; the boundaries of the repaired polygons are instead formed by edges incident to two triangles having different labels.

For each polygon, we walk at a triangle, and move on to triangles having the same label.



(a) The original polygons.

(b) Repaired each triangle using the label adjacent along the longest boundary from the neighbouring triangles.

(c) Repaired each region using a random label from the neighbouring triangles.

(d) Repaired each region using the label adjacent along the longest boundary from the neighbouring triangles.

Figure 8: Different repair operations used in the two polygons for the Arribes del Duero Natural Park in Spain (red/darker grey) and the International Douro Natural Park in Portugal (green/lighter grey). All of them can be considered best by a certain criterion, like preserving the area ratio between the two polygons (b), smoothness of the boundary (d), or a balance between the two (c).

As the process goes, a single polyline that runs along all the boundaries of the polygon is generated. This involves a depth-first search (clockwise) that recursively reaches until the boundary of a polygon, returning a long chain of edges in a procedure similar to following the boundary edge by edge. The procedure is shown step by step in Figure 9. The polyline created with this method has “bridges”, which allow us to keep all inner boundaries (holes) connected with the outer one, in a manner that keeps the interior connected as well. These help to preserve connectivity and the relations between different (outer and inner) boundaries, but are removed later in the process (to conform to the Simple Features specification). Also, its orientation conveys the in-

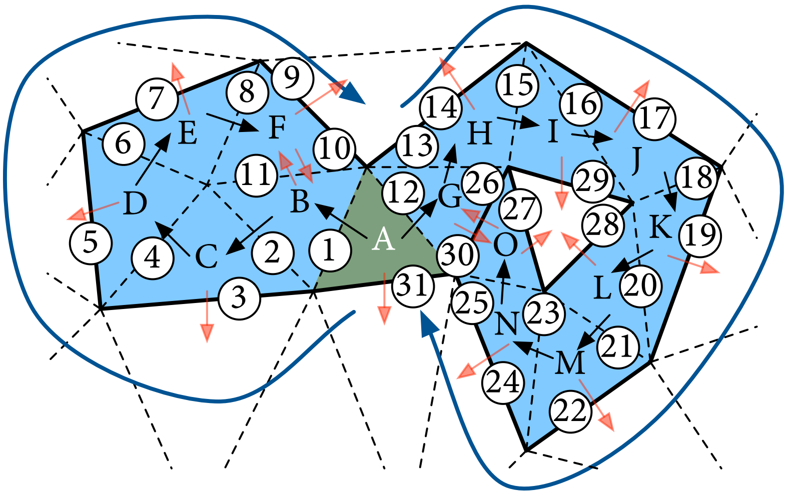


Figure 9: The traversal order when navigating through triangles in a clockwise manner. Starting at $\triangle A$ and the edge between $\triangle A$ and $\triangle B$, the operations occur according to the encircled numbers. Black arrows denote when an unprocessed triangle is found, red arrows when it is not. Notice how the traversal is performed clockwise (shown in dark blue arrows) for both cases, despite starting from different sides.

formation of whether a section of it is part of an inner or an outer boundary.

This polyline is processed with a stack-based algorithm that generates separate closed rings for the outer boundary and each of the inner boundaries, collapsing the "bridges" that were generated. In order to do this, the polyline is cut at the places where more than two edges join, and these are joined in the correct order by keeping track of (yet) unclosed rings. When a new segment is being processed, it can be one of three options: one that completes a ring, one that is part of a bridge, or one that is not yet closed. Closed rings are stored and bridges are removed, while unclosed rings are saved in the stack until they can be popped to form a closed ring together with a new segment. This is shown in Figure 10.

4 Implementation and Experiments

4.1 Implementation

An implementation of the algorithms described in Section 3 was written in the C++ programming language, using external libraries for some functionality. The developed software is called **pprepair**, and is open source and freely available at `http://tudelft3d.github.com/pprepair/`. C++ was selected in order to have plenty of control with regards to low level details and to achieve good performance, which makes it possible to compare it with existing solutions. The libraries used are: the OGR Simple Features Library[9] (which allows input and output from a large variety of data formats common in GIS); and CGAL[10] (which has support for many robust spatial data structures and the operations based on them; we use its constrained triangulation module).

4.2 Experiments with real-world planar partition datasets

We have made experiments with four freely available real-world datasets, i.e. we have validated and automatically repaired them with our implementation; the overview of these datasets is shown in Figure 11 and

[9] `http://www.gdal.org/ogr/`

[10] Computational Geometry Algorithms Library: `http://www.cgal.org/`

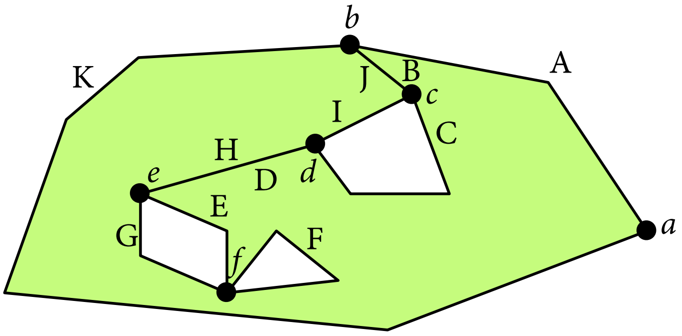
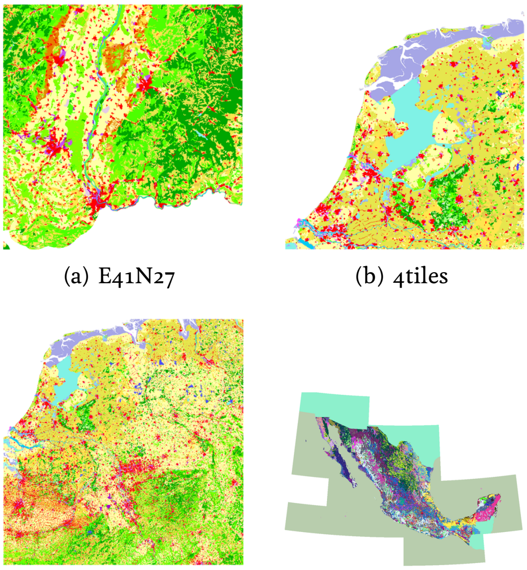


Figure 10: Processing the polyline shown starting from vertex a and moving towards polyline $\overline{A}$, the following operations occur: (1) $\overline{A}$ unclosed→push, (2) $\overline{B}$ unclosed→push, (3) $\overline{C}$ unclosed→push, (4) $\overline{D}$ unclosed→push, (5) $\overline{E}$ unclosed→push, (6) $\overline{F}$ closed→**store**, (7) pop→ $\overline{EG}$ closed→**store**, (8) pop→ $\overline{DH}$ degenerate→erase, (9) pop→ $\overline{CI}$ closed→**store**, (10) pop→ $\overline{BJ}$ degenerate→erase, (11) pop→ $\overline{AK}$ closed→**store**.


(a) E41N27

(b) 4tiles

(c) 16tiles

(d) Mexico

Figure 11: Overview of the four datasets.

their properties in Table 2. The datasets are the following:

E41N27 Corine 2000[11] tile E41N27, which contains a shifted polygon (by about 10 cm), creating many small gaps and overlaps in the dataset. The snapping threshold has been set at 1 m.

4tiles Corine 2000 tiles E39N32, E39N33, E40N32 and E40N33, which are known to have long and thin overlapping regions (< 1 mm) with each other. The snapping threshold has been set at 1 cm.

16tiles 16 adjacent Corine 2000 tiles: E39N30, E39N31, E39N32, E39N33, E40N30, E40N31, E40N32, E40N33, E41N30, E41N31, E41N32, E41N33, E42N30, E42N31, E42N32, E42E33. Some have gaps between one another, some overlap, but match within a few centimetres. The snapping threshold has been set at 10 cm.

Mexico 1:1 000 000 scale land cover dataset from INEGI consisting of over 26 000 polygons. Interestingly, it is mostly already valid (according to the Shapefile specification), but contains some very large polygons, with tens of thousands of vertices.

As a comparison, we have also tried to perform the same operations with other available software. While the capabilities for planar partition repair among the software tested vary considerably, with full topological repair only available in ArcGIS (using manual operations only), it is also important to consider how different repair implementations scale to large datasets. For

[11] Corine is a land cover dataset for 32 European countries. It is freely available at `http://www.eea.europa.eu/data-and-maps/data/corine-land-cover-2000-clc2000-seamless-vector-database`.

Table 2: Properties of the datasets used for the experiments.

	# polygons	# pts	# pts largest polygon	avg # pts per polygon
E41N27	14 969	496 303	26 740	33.7
4tiles	4 984	365 702	16 961	74.7
16tiles	63 868	6 622 133	95 112	103.7
Mexico	26 866	4 181 354	117 736	155.6

this, a few performance tests were made in our implementation, and three planar partition repair tools that perform this process using snapping and splitting. The testing methodology for each tool is as follows:

ArcGIS In ArcCatalog, a multiple feature dataset is created in a Geodatabase, set with XY resolution values equal to the snapping threshold. The features are imported into it and the merge and dissolve operations are used to merge adjacent polygons with the same ID. Topology is then generated to check that the planar partition is valid. Everything is finally exported to a single Shapefile. The individual parts of the process are timed and the total is recorded. Memory usage is calculated as the difference between the just loaded ArcCatalog application and its maximum memory usage throughout the process.

FME A reader is created for each input file, which serve as input to a Snapper transformer; features with the same IDs are then dissolved, and finally they are output into a new Shapefile writer. The topology generator is used to be able to tell whether the result is a valid planar partition. Results are timed and the maximum memory allocation of the `fme.exe` process is recorded.

GRASS Input files are imported with `v.in.ogr`, with all polygon cleaning operations performed and snapping set to the correct values. Boundaries between features with the same IDs are then dissolved using `v.dissolve`. Files are then exported with `v.out.ogr`. Times reported by GRASS are added together to give the total, while memory usage by the `v.in.ogr.exe`, `v.dissolve.exe` and `v.out.ogr.exe` are monitored and their maximum is recorded.

pprepair Files are read and put into the triangulation, the triangulation is labelled, the repair is performed with the longest boundary first (see Table 1), with ambiguous cases resolved with a random choice. Polygons are then extracted from the triangulation and output to a single Shapefile. The entire process is timed, and the maximum amount of memory used is recorded.

Notice that we are somewhat comparing apples with oranges here since our implementation is able to repair more cases than other methods, keeps topological consistency, and does not require finding out the appropriate threshold value (if it exists). More importantly, it is able to directly state whether the result is a planar partition, unlike the three other solutions (with these snapping is performed but that does not guarantee that the output is a valid planar partition). The results of the experiments are shown in Table 3.

We have made the experiments in order to have an idea of the processing time and the memory usage involved. Only cases that are acceptably solved by snapping and splitting have been considered, since there is no other comparable automated topological repair tool among those studied that would also work for more complex cases, such as those that require a snapping threshold too high to be practical (e.g. horizontal conflation of independently generated datasets [Yuan and Tao, 1999; Davis, 2003]).

Table 3: Planar partition repair comparison using large datasets.

	pprepair		ArcGIS		FME		GRASS	
	memory	time	memory	time	memory	time	memory	time
E41N27	145 MB	19s	145 MB	1m3s	158 MB	31s	59 MB	3m9s
4tiles	116 MB	17s	113 MB	37s	105 MB	31s	49 MB	53s
16tiles	1.45 GB	4m47s	crashes	-	636 MB	15m48s	crashes	-
Mexico	1.01 GB	3m31s	216 MB	>1d	264 MB	2m45s	408 MB	11m38s

All tests have been run 5 times in a machine just booted and the results averaged to account for the small variations that occurred, except in the case where the execution was cancelled after one full day, due to time limitations, or when it causes the program to crash. The hardware is a 2.66 GHz Core 2 Duo MacBook Pro with 4 GB of RAM. ArcGIS 9.3, FME 2010 SP1 and GRASS 6.4 were run in Windows 7, while pprepair was run in Mac OS X 10.7.1.

As Table 3 shows, our approach uses somewhat more memory than other solutions. This is explained by the extra (unconstrained) edges that are added to the input over when triangulating them. It should however be noticed that both ArcGIS and GRASS crashed with the biggest dataset 16tiles. Our implementation is the fastest of the four tested, being for instance around three times faster than FME for the biggest datasets. Only for the Mexico dataset is our implementation slower than FME. This is (probably) explained by the fact that its polygons already form a valid planar partition, and therefore very few snapping operations have been performed by FME; the planar graph of the input was basically built, and then the polygons saved back to disk. We believe that ArcGIS and GRASS struggled with the dataset because it contains several very large polygons (with more than 10 000 vertices). Our implementation took 3m31s, but, as Table 4 shows, it took CGAL 3m02s to simply triangulate the input edges. This table also demonstrates the efficiency of our approach: for the four datasets, around 85% of the time of our approach was used—by the CGAL library—to read the input from disk and triangulate the input edges.

Table 4: Timed steps of the planar partition repair procedure, rounded down to the nearest second. The percentage is the triangulation time over the total time.

	E41N27	4tiles	16tiles	Mexico
Triangulate	0:17	0:15	4:00	3:02
Label	0:01	0:01	0:27	0:16
Repair	0:00	0:00	0:00	0:00
Reconstruct	0:01	0:01	0:11	0:07
Output	0:00	0:00	0:10	0:06
Total	0:19	0:17	4:47	3:31
triangulate %	89%	88 %	83%	86%

During the implementation, we took several engineering decisions to optimise our program. One of them was to favour disk space over computation time, as we wanted to be able to process large datasets. This is why we always reconstruct polygons, even if they not modified by the repair process. Although possible, it would require us to keep in memory the original polygons (which would significantly increase the memory consumption) and to keep track of which labels have been modified. And, as Table 4 shows, the reconstruction is very efficient as it only takes around 3-4% of the total time for the 16tiles and Mexico datasets.

5 Discussion and Conclusions

We have presented a new method for repairing polygonal area partitions, ensuring that the output partition is valid, i.e. all individual polygons are conforming to the ISO 19107 [ISO, 2003] and Simple Features [OGC, 2006] specifications and no gaps nor overlaps are present between any pair of polygons. The novelty of our method lies in the fact that repair is performed according to user defined criteria, but then takes place without any human intervention. Automatic repair is becoming increasingly an important topic due to data integration, e.g. data collected for the different themes of the European INSPIRE Initiative will finally have to fit together and data could eventually be matched fully automatically by dedicated web service components [INSPIRE, 2009]. Our approach could be at the heart of such a web service.

The proposed approach excels in automated repair at the cost of increased memory usage compared to a pure graph-based approach—this difference is mainly caused by the unconstrained edges introduced by the triangulation. While it would be possible to use our repair rules together with a (primal/dual) graph-based approach, these additional edges in the triangulation give fine-grained control over the repair operations, and ensures that the graph is connected, which facilitates the reconstruction of polygons.

We have implemented our algorithm over a numerically robust triangulator (CGAL) and since repair operations are expressed solely in terms of re-labelling of triangles (no geometric computation is involved), the approach is also fully robust. Since, during our experiments, most of the time was used to compute the constrained triangulation, another library could also be tested to improve the implementation.

For the future, we plan to:

- Improve the scalability of the approach and process and repair datasets with more than 10 million polygons. It is known that using divide-and-conquer techniques triangulation algorithms can handle big datasets [Amenta et al., 2003; Blandford et al., 2005]. We will investigate whether it is possible to automatically repair each divided part individually and 'glue' the repaired parts together.
- Investigate snap rounding [Hobby, 1999; de Berg et al., 2007] as a preprocessing step — or embedded directly in the triangulation — to guarantee that repaired planar partitions have no vertices that are closer than a certain ε threshold. However, snap rounding may change the topology of the input, but the output will nevertheless be a valid partition (as the topology will be repaired). Apart from topological changes, snap rounding can also lead to removed polygon parts that are too small to be preserved based on the chosen ε.
- Add more advanced repair operations to our repair toolkit, e.g. repair could take place based on splitting a collection of triangles.
- Extend our work to include the third dimension to validate and repair 3D city models using a constrained tetrahedralisation [Si, 2008]. Notice that the tetrahedralisation of a given polyhedron does not always exist, and thus extra (Steiner) points might need to be added. The main concepts of our approach, (re)labelling and reconstruction, extend naturally to 3D. However, appropriate repair operations for 3D city models would need to be defined, and some application-specific constraints (e.g. right angles at corners) are not trivial to implement in our current approach.

References

Nina Amenta, Sunghee Choi, and Günter Rote. Incremental constructions con BRIO. In *Proceedings of the 19th Annual Symposium on Computational Geometry*, pages 211–219. ACM Press, 2003.

Wael M. Badawy and Walid G. Aref. On local heuristics to speed up polygon-polygon intersection tests. In *Proceedings of the 7th ACM International Symposium on Advances in Geographic Information Systems*, pages 97–102. ACM, 1999.

M. Bader and R. Weibel. Detecting and resolving size and proximity conflicts in the generalization of polygonal maps. In *Proceedings of the 18th International Cartographic Conference. Stockholm*, pages 1525–1532, 1997.

Daniel K. Blandford, Guy E. Blelloch, David E. Cardoze, and Clemens Kadow. Compact representations of simplicial meshes in two and three dimensions. *International Journal of Computational Geometry and Applications*, 15(1):3–24, 2005.

CGAL. *CGAL 3.8 User and Reference Manual*. CGAL Editorial Board, 2011.

Kenneth L. Clarkson, Kurt Mehlhorn, and Raimund Seidel. Four results on randomized incremental constructions. In Alain Finkel and Matthias Jantzen, editors, *Proceedings of the 9th Annual Symposium on Theoretical Aspects of Computer Science*, volume 577 of *Lecture Notes in Computer Science*, pages 461–474. Springer Berlin / Heidelberg, 1992.

Martin Davis. Java conflation suite. Technical report, Vivid Solutions, 2003. Available at `http://www.vividsolutions.com/jcs/`.

Mark de Berg, Dan Halperin, and Mark Overmars. An intersection-sensitive algorithm for snap rounding. *Computational Geometry*, 36(3):159 - 165, 2007. ISSN 0925-7721.

Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 3 edition, 2008.

ESRI. Shapefile technical description. White paper, ESRI, July 1998.

Michael A. Facello. Implementation of a randomized algorithm for Delaunay and regular triangulations in three dimensions. *Computer Aided Geometric Design*, 12 (4):349–370, 1995.

GTS. *GTS Library Reference Manual*, 2006. URL `http://gts.sourceforge.net/reference/book1.html`.

Leonidas Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi. *ACM Transactions on Graphics*, 4(2):74–123, 1985.

John D. Hobby. Practical segment intersection with finite precision output. *Computational Geometry*, 13(4):199 - 214, 1999. ISSN 0925-7721.

Christoph M. Hoffmann, John E. Hopcroft, and Michael S. Karasick. Towards implementing robust geometric computations. In *Proceedings of the 4th Annual Symposium on Computational Geometry*, pages 106–117. ACM, 1988.

INSPIRE. D2.5: Generic conceptual model, version 3.2. Technical report, Drafting Team "Data Specifications", 2009. URL `http://inspire.jrc.ec.europa.eu/documents/Data_Specifications/D2.5_v3.2.pdf`.

ISO. 19107: Geographic information - spatial schema, March 2003.

David Kirkpatrick, Jack Snoeyink, and Bettina Speckmann. Kinetic collision detection for simple polygons. *International Journal of Computational Geometry and Applications*, 12(1–2):3–27, 2002.

Robert Laurini and Françoise Milleret-Raffort. Topological reorganization of inconsistent geographical databases: A step towards their certification. *Computers & Graphics*, 18(6):803–813, December 1994.

Hugo Ledoux and Martijn Meijers. Validation of planar partitions using constrained triangulations. In *Proceedings Joint International Conference on Theory, Data Handling and Modelling in GeoSpatial Information Science*, pages 51–55, Hong Kong, May 2010.

Hugo Ledoux, Ken Arroyo Ohori, and Martijn Meijers. Automatically repairing invalid polygons with a constrained triangulation. In *Proceedings of the 15th AGILE International Conference on Geographic Information Science*, 2012.

Yuanxin Liu and Jack Snoeyink. Faraway point: A sentinel point for Delaunay computation. *International Journal of Computational Geometry and Applications*, 18(4): 343–355, November 2006.

Avraham Margalit and Gary D. Knott. An algorithm for computing the union, intersection or difference of two polygons. *Computers & Graphics*, 13(2):167–183, 1989.

Ernst P. Mücke, Isaac Saias, and Binhai Zhu. Fast randomized point location without preprocessing in two- and three-dimensional Delaunay triangulations. *Computational Geometry—Theory and Applications*, 12:63–83, 1999.

OGC. *OpenGIS Implementation Specification for Geographic Information - Simple Feature Access - Part 1: Common Architecture*, 1.2.0 edition, October 2006.

Lutz Plümer and Gerhard Gröger. Achieving integrity in geographic information systems—maps and nested maps. *GeoInformatica*, 1(4):345–367, 1997.

M. Rivero and F.R. Feito. Boolean operations on general planar polygons. *Computers & Graphics*, 24(6):881–896, December 2000.

Stefan Schirra. *Precision and Robustness in Geometric Computations*, volume Algorithmic Foundations of Geographic Information Systems of *Lecture Notes in Computer Science*, chapter 9, pages 255–287. Springer Berlin / Heidelberg, 1997.

Michael Ian Shamos and Dan Hoey. Geometric intersection problems. In *FOCS*, pages 208–215. IEEE, 1976.

Jonathan Richard Shewchuk. *Delaunay Refinement Mesh Generation*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburg, USA, 1997a.

Jonathan Richard Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18:305–363, 1997b.

Hang Si. *Three Dimensional Boundary Conforming Delaunay Mesh Generation*. PhD thesis, Technische Universität Berlin, 2008.

C. Dana Tomlin. Map algebra: One perspective. *Landscape and Urban Planning*, 30(1-2):3–12, October 1994.

Peter van Oosterom, Wilko Quak, and Theo Tijssen. About invalid, valid and clean polygons. In Peter F. Fisher, editor, *Developments in Spatial Data Handling—11th International Symposium on Spatial Data Handling*, pages 1–16. Springer, 2004.

Jan W. van Roessel. A new approach to plane-sweep overlay: Topological structuring and line-segment classification. *Cartography and Geographic Information Science*, 18(1):49–67, January 1991.

Shuxin Yuan and Chuang Tao. Development of conflation components. *Geoinformatics and Socioinformatics*, pages 1–13, June 1999.