Simplifying Massive Contour Maps

Lars Arge[1], Lasse Deleuran[1], Thomas Mølhave[2], Morten Revsbæk[1], and Jakob Truelsen[3]

[1] MADALGO[†], Department of Computer Science, Aarhus University
[2] Department of Computer Science, Duke University.
[3] SCALGO, Scalable Algorithmics, Denmark

Abstract. We present a simple, efficient and practical algorithm for constructing and subsequently simplifying contour maps from massive high-resolution DEMs, under some practically realistic assumptions on the DEM and contours.

1 Introduction

Motivated by a wide range of applications, there is extensive work in many research communities on modeling, analyzing, and visualizing terrain data. A (3D) digital elevation model (DEM) of a terrain is often represented as a planar triangulation $\mathbf{M}$ with heights associated with the vertices (also known as a triangulated irregular network or simply a TIN). The l-level set of $\mathbf{M}$ is the (2D) segments obtained by intersecting $\mathbf{M}$ with a horizontal plane at height l. A *contour* is a connected component of a level set, and a *contour map* $\mathcal{M}$ is the level sets at a set of heights; refer to Figure 2. Contour maps are widely used to visualize a terrain primarily because they provide an easy way to understand the topography of a terrain from a simple two-dimensional representation.

Early contour maps were created manually, severely limiting the size and resolution of the created maps. However, with the recent advances in mapping technologies, such as laser based LIDAR technology, billions of (x, y, z) points on a terrain, at sub-meter resolution with very high accuracy ($\sim$10-20 cm), can be acquired in a short period of time and with a relatively low cost. The massive size of the data (DEM) and the contour maps created from them creates problems, since tools for processing and visualising terrain data are often not designed to handle data that is larger than main memory. Another problem is that contours generated from high-resolution LIDAR data are very detailed, resulting in a large amount of excessively jagged and spurious contours; refer to Figure 2. This in turn hinders their primary applications, since it becomes difficult to interpret the maps and gain understanding of the topography of the terrain. Therefore we are interested in simplifying contour maps.

Previous work: The inefficiency of most tools when it comes to processing massive terrain data stems from the fact that the data is too large to fit in

[†] Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation

main memory and must reside on slow disks. Thus the transfer of data between disk and main memory is often a bottleneck (see e.g. [9]). To alleviate this bottleneck one needs algorithms designed in the I/O-model of computation [4]. In this model, the machine consists of a main memory of size M and an infinite-size disk. A block of B consecutive elements can be transferred between main memory and disk in one *I/O operation* (or simply *I/O*). Computation can only take place on elements in main memory, and the complexity of an algorithm is measured in terms of the number of I/Os it performs. Over the last two decades, I/O-efficient algorithms and data structures have been developed for several fundamental problems. See recent surveys [5, 21] for a comprehensive review of I/O-efficient algorithms. Here we mention that scanning and sorting N elements takes $O(\text{Scan}(N)) = O(N/B)$ and $O(\text{Sort}(N)) = O(N/B \log_{M/B}(N/B))$ I/Os, respectively. The problem of computing contours and contour maps have previously been studied in the I/O-model [2, 1]. Most relevant for this paper, Agarwal *et al.* [1] presents an optimal I/O-efficient algorithm that computes a contour map (along with the nesting of the individual contours) in $O(\text{Sort}(N)+\text{Scan}(|\mathcal{M}|))$ I/Os, where $|\mathcal{M}|$ is the number of segments in the contour map and N is the number of triangles in the DEM. However, this algorithm is not practical.

While the problem that contour maps generated from high-resolution LIDAR data contain excessively jagged contours, can be alleviated by contour map simplification (while also alleviating some of the scalability problems encountered when processing contour maps), the main issues is of course to guarantee the accuracy of the simplified contour map. There are two fundamental approaches to simplifying a contour map: The DEM $\mathbf{M}$ can be simplified before computing the contour map $\mathcal{M}$, or $\mathcal{M}$ can be simplified directly.

There has been a lot of work on simplifying DEMs; refer e.g. to [8, 14] and the references therein. However, most often the simple approaches do not provide a guarantee on the simplification accuracy, while the more advanced approaches are not I/O-efficient and therefore do not scale to large data sets. Developing I/O-efficient DEM simplification algorithms with simplification guarantees have shown to be a considerable challenge, although an $O(\text{Sort}(N))$ I/O (topological persistence based) algorithm for removing "insignificant" features from a DEM (resulting in small contours) have recently been developed [3].

Simplifying the contour map $\mathcal{M}$ directly is very similar to simplifying a set of polygons (or polygonal lines) in the plane. Polygonal line simplification is a well studied problem; refer e.g. to [16] for a comprehensive survey. However, there are a number of important differences between contour maps and polygonal line simplification. Most noticeably, simplifying a contour line in the plane using a polygonal line simplification algorithms will, even if it guarantees simplification accuracy in the plane, not provide a z-accuracy guarantee. Furthermore, simplifying the contours individually may lead to intersections between the simplified contours. Finally, when simplifying contour maps its very important to preserve the relationships between the contours (the *homotopic* relationship), that is, maintain the nesting of the contours in the map. Note that intersections are au-

tomatically avoided and homotopy preserved when simplifying the DEM before computing the contour map.

One polygonal line simplification algorithm that is often favored for its simplicity and high subjective and objective quality on real life data is Douglass and Peucker's line simplification algorithm [12]. The algorithm simplifies a contour by removing segment endpoints from the contour while ensuring that the distance between the original and the simplified contour is within a distance parameter ε_{xy} (but it does not guarantee that it removes the optimal number of endpoints under this constraint). Modifications of this algorithm have been developed, that removes self-intersections in the output [19], as well as ensures homotopy relative to a set of obstacles (polygons) [11, 7]. However, these modified algorithms are complicated and/or not I/O-efficient (and do also not consider z-accuracy).

Our results: In this paper we present a simple, efficient and practical algorithm for constructing and subsequently simplifying contour maps from massive high-resolution DEMs, under some practically realistic assumptions on the DEM and contours. The algorithm guarantees that the contours in the simplified contour map are homotopic to the unsimplified contours, non-intersecting, and within a distance of ε_{xy} of the unsimplified contours in the xy plane. Furthermore, it guarantees that for any point p on a contour in the l-level-set of the simplified contour map, the difference between l and the elevation of p in $\mathbf{M}$ (the z-error) is less than ε_z. We also present experimental results that show a significant improvement in the quality of the simplified contours along with a major (about 90%) reduction in size.

Overall, our algorithm has three main components. Given the levels $\ell_1, \ldots, \ell_d$, the *first component*, described in Section 3, computes the segments in the contour map $\mathcal{M}$ containing a level-set for each input level. The component also computes level-sets for each levels $\ell_i \pm \varepsilon_z$, $1 \leqslant i \leqslant d$. The contours generated from these extra levels will be used to ensure that the z-error is bounded by ε_z. We call these contours *constraint* contours and mark the contours in $\mathcal{M}$ that are not constraint contours. The component also orders the segments around each contour and computes how the contours are nested. It uses $O(\mathrm{Sort}(|\mathcal{M}|))$ I/Os under the practically realistic assumptions that each contour, as well as the contour segments intersected by any horizontal line, fit in memory. This is asymptotically slightly worse than the theoretically optimal but complicated algorithm by Agarwal *et al.* [1]. The *second component*, described in Section 4, computes, for each of the marked contours P, the set $\mathcal{P}$ of contours that need to be considered when simplifying P. Intuitively, $\mathcal{P}$ contains all (marked as well as constraint) contours that can be reached from P without crossing any other contour of $\mathcal{M}$. Although each contour can be involved in many sets $\mathcal{P}$, the entire algorithm uses only $O(\mathrm{Sort}(|\mathcal{M}|))$ I/Os. The *third component*, described in Section 5, simplifies each marked contour P within $\mathcal{P}$. This is done using a modified version of Douglas and Peucker's simplification algorithm [12]. As with Douglas Peucker's algorithm it guarantees that the simplified contour $\mathcal{P}'$ is within distance ε_{xy} of $\mathcal{P}$, but it also guarantees that P' is homotopic to the original contour P (with respect to $\mathcal{P}$) and that P' does not have self intersections. The

existence of the constraint contours in $\mathcal{P}$ together with the homotopy guarantee ensures the z-error is within ε_z. Under the practically realistic assumptions that each contour P along with $\mathcal{P}$ fits in internal memory, the algorithm does not use any extra I/Os.

The details on the implementation of our algorithm is given in section 6 along with experimental results on a terrain data set of Denmark with over 12 billion points. We e.g. construct a contour map for the entire country with a granularity of 0.5m (i.e. with a level-set for every 0.5m) and simplify it with $\varepsilon_{xy} = 5$m and $\varepsilon_z = 0.2$m in 49 hours. While the original contour have about 4.8 billion segments in about 7 million contours (occupying about 53 gigabytes just for the xyz coordinates) the simplified contour map only has about 9.2% of the original points (and taking up only 5 gigabytes). Besides the obvious decrease in storage requirements, the generated contour map is much more visually pleasing than the original map.

2 Preliminaries

We consider the terrain $\mathbf{M}$ to be represented as a triangular irregular network, which is a planar triangulation whose vertices v are associated with heights $h(v)$. The height of a point interior to a triangle is determined by linearly interpolating from the triangle vertex heights. In this way, the terrain is represented as a continuous piecewise linear surface, and we can use $h(p)$ to refer to the height of any point p in $\mathbb{R}^2$.

Paths and polygons: Let $p_1, \ldots, p_n$, be a sequence of $n > 1$ points in $\mathbb{R}^2$. The *path* Q defined by these points is the set of line segments defined by pairs of consecutive points in the sequence. The points $p_1, \ldots, p_n$ are called the *vertices* of Q. A *simple path* is a path where only consecutive segments intersect, and only at the endpoints. Given integers $1 \leqslant i < j \leqslant n$, the *sub-path* Q_{ij} is the path defined by the vertices $p_i, p_{i+1}, \ldots, p_j$. We abuse notation slightly by using Q to denote both the sequence of vertices, and the path itself. We define the size of Q as its number of segments, i.e. $|Q| = n - 1$. A path Q' is a *simplification* of a path Q if $Q' \subseteq Q$ and the vertices of Q' appear in the same order as in Q.

A *polygon* (*simple polygon*) is a path (simple path) P where $p_1 = p_n$. A simple polygon P partitions $\mathbb{R}^2 \setminus P$ into two open sets — a bounded one called *inside* of P and denoted by P^{i}, and an unbounded one called *outside* of P and denoted by P^{o}. We define a *family of polygons* to be a set of non-intersecting and vertex-disjoint simple polygons. Consider two simple polygons P_1 and P_2 in a family of polygons A. P_1 and P_2 are called *neighbors* if no other polygon $P \in A$ separates them, i.e., one of P_1 and P_2 is contained in P^{i} and the other in P^{o}. If P_1 is neighbor to P_2 and $P_1 \subset P_2^{\mathrm{i}}$, then P_1 is called a *child* of P_2, and P_2 is called the *parent* of P_1, we will refer to the parent of P as $\hat{P}$. The *topology* of a family of polygons $\mathcal{M}$ describes how the polygons are nested i.e. the parent/child relationship between polygons.

(a) (b)

Fig. 1. (a) Polygonal domain $\mathcal{P}$ (solid lines) of P (dashed lines). (b) The valid simplification P' of P within $\mathcal{P}$ and a polygon Q not homotopic to P.

Given a polygon P in A, the *polygonal domain of* P, denoted $\mathcal{P}$, consists of the neighbors $P_1 \dots P_k$ of P in A, see Figure 1 . We define the size of $\mathcal{P}$ to be $|\mathcal{P}| = |P| + \sum_i |P_i|$.

Intuitively, two paths Q and Q' are *homotopic* with regards to a polygonal domain $\mathcal{P}$ if one can be continuously transformed into the other without intersecting any of the polygons of $\mathcal{P}$. We refer to Figure 1 for an illustration of this. The exact definition can be found in Appendix A. Path Q' is *strongly homotopic* to Q if Q' is a simplification of Q and if every segment $q'_i q'_{i+1}$ in Q' is homotopic to the corresponding sub-path $Q_{k,l}$ where $q'_i = q_k$ and $q'_j = q_l$. It follows that Q and Q' are also homotopic, but the reverse implication does not necessarily hold.

Given two indices $1 \leqslant i, j \leqslant n$ we define the distance $d(p, i, j)$ between any point $p \in \mathbb{R}^2$ and line segment $p_i p_j$ as the distance from p perpendicular to the line defined by $p_i p_j$. We define the error $\varepsilon(i, j)$ to be the maximum distance between the vertices of P_{ij} and the line segment $p_i p_j$, i.e. $\varepsilon(i, j) = \max\{d(i, j, p_i), d(i, j, p_{i+1}), \dots, d(i, j, p_k)\}$. Let P' be a simplification of P. Given a *simplification threshold* ε, we say that P' is a *valid simplification* of P if it is a simple polygon homotopic to P in $\mathcal{P}$ and $\varepsilon(i, j) < \varepsilon$ for any segment $p_i p_j$ of P'.

Contours and contour maps: For a given terrain $\mathbf{M}$ and a *level* $\ell \in \mathbb{R}$, the *ℓ-level set* of $\mathbf{M}$, denoted by $\mathbf{M}_\ell$, is defined as $h^{-1}(\ell) = \{x \in \mathbb{R}^2 \mid h(x) = \ell\}$. A *contour* of a terrain $\mathbf{M}$ is a connected component of a level set of $\mathbf{M}$. Given a list of levels $\ell_1 < \ldots < \ell_d \in \mathbb{R}$, the *contour map* $\mathcal{M}$ of $\mathbf{M}$ is defined as the union of the level-sets $\mathbf{M}_{\ell_1}, \dots, \mathbf{M}_{\ell_d}$. For simplicity, we assume that no vertex of $\mathbf{M}$ has height $\ell_1, \dots, \ell_d$ and we assume that $\mathbf{M}$ is given such that M_{ℓ_1} consists of only a single boundary contour $\mathcal{U}$. This implies that the collection of contours in the contour map form a family of polygons and that each polygon in the family, except $\mathcal{U}$, has a parent. It allows us to represent the topology of $\mathcal{M}$ as a tree $\mathcal{T} = (V, E)$ where the vertices V is the family of polygons and where E contains an edge from each polygon $P \neq \mathcal{U}$ to its parent polygon $\hat{P}$. The root of $\mathcal{T}$ is $\mathcal{U}$. We will refer to $\mathcal{T}$ as the *topology tree* of $\mathcal{M}$.

3 Building the Contour Map

In this section we describe our practical and I/O-efficient algorithm for constructing the contour map $\mathcal{M}$ of the terrain $\mathbf{M}$ and the topology tree $\mathcal{T}$ of $\mathcal{M}$, given a list of regular levels $\ell_1 < \ldots < \ell_d \in \mathbb{R}$. We will represent $\mathcal{M}$ as a sequence

of line segments such that the clockwise ordered segments in each polygon P of $\mathcal{M}$ appear consecutively in the sequence, and $\mathcal{T}$ by a sequence of edges (P_2,P_1) indicating that P_2 is the parent of P_1; all segments in $\mathcal{M}$ of polygon P will be augmented with (a label for) P and the BFS number of P in $\mathcal{T}$. We will use that the segments in any polygon P in $\mathcal{M}$, as well as the segments in $\mathcal{M}$ intersecting any horizontal line, fit in memory.

Computing contour map $\mathcal{M}$: To construct the line segments in $\mathcal{M}$, we first scan over all the triangles of $\mathbf{M}$. For each triangle f we consider each level ℓ_i within the elevation range of the three vertices of f and construct a line segment corresponding to the intersection of z_{ℓ_i} and f. To augment each segment with a polygon label, we then view the edges as defining a planar graph such that each polygon is a maximally connected component in this graph. We find these connected components practically I/O-efficiently using an algorithm by Arge et al. [6], and then we use the connected component labels assigned to the segments by this algorithm as the polygon label. Next we sort the segments by their label. Then, since the segments of any one polygon fit in memory, we can in a simple scan load the segments of each polygon P into memory in turn and sort them in clock-wise order around the boundary of P.

Computing the topology tree $\mathcal{T}$ of $\mathcal{M}$: We now give an outline of our algorithm for computing the topology tree. We refer to Appendix B for a detailed description. We will use a plane-sweep algorithm to construct the edges of $\mathcal{T}$ from the sorted line segments in $\mathcal{M}$. During the algorithm we will also compute the BFS number of each polygon P in $\mathcal{T}$. After the algorithm it is easy to augment every segment in $\mathcal{M}$ with the BFS number of the polygon P it belongs to in a simple sorting and scanning step.

For a given $\mu \in \mathbb{R}$, let y_μ be the horizontal line through μ. Starting at $\mu = y_\infty$, our algorithm sweeps the line y_μ though the (pre-sorted) edges of $\mathcal{M}$ in the negative y-direction. We maintain a search tree $\mathcal{S}$ on the set of segments of $\mathcal{M}$ that intersect y_μ. For each edge in $\mathcal{S}$ we have already computed its parent and this information is stored with the edge. Each edge also knows the identity of its own polygon. The set of edges in $\mathcal{S}$ changes as the sweep-line encounters endpoints of segments in $\mathcal{M}$, and each endpoint v from some polygon P has two adjacent edges s_1 and s_2. If the other two endpoints of s_1 and s_2 are both above y_μ we simply delete s_1 and s_2 from $\mathcal{S}$. If both endpoints are below y_μ and there is not already a segment from P in $\mathcal{S}$, then this is the first time P is encountered in the sweep. We can then use $\mathcal{S}$ to find the closest polygon segment s_3 from some other polygon P_1 to the left of v. Depending on the orientation of s_3 we know that the parent of P, $\hat{P}$ is either P_1, or the parent of P_1 which is stored with P_1 in $\mathcal{S}$, and we can output corresponding edge $(P, \hat{P})$ of $\mathcal{T}$. It is easy to augment this algorithm so that it also computes the BFS number of each P in $\mathcal{T}$ by also storing, with each edge of $\mathcal{S}$ BFS rank r of its parent, the rank of a child polygon is then $r + 1$.

Analysis: The algorithm for computing the contour map $\mathcal{M}$ uses $O(\text{Sort}(|\mathcal{M}|))$ I/Os: First it scans the input triangles to produce the segments of $\mathcal{M}$ and invokes the practically efficient connected component algorithm of Arge et al. [6]

that uses $O(\text{Sort}(|\mathcal{M}|))$ I/Os under the assumption that the segments in $\mathcal{M}$ intersecting any horizontal line fit in memory. Then it sorts the labeled segments using another $O(\text{Sort}(|\mathcal{M}|))$ I/Os. Finally, it scans the segments to sort each polygon in internal memory, utilizing that the segments in any polygon P in $\mathcal{M}$ fits in memory. Also the algorithm for computing the topology tree $\mathcal{T}$ uses $O(\text{Sort}(|\mathcal{M}|))$: After scanning the segments of $\mathcal{M}$ to produce L, it performs one sort and one scan of L, utilizing the assumption that $\mathcal{S}$ fits in memory. Augmenting each segment in $\mathcal{M}$ with the BFS number of the polygon P that it belongs to, can also be performed in $O(\text{Sort}(|\mathcal{M}|))$ in a simple way.

4 Simplifying Families of Polygons

This section describes our practical and I/O-efficient algorithm for simplifying a set of marked polygons in a family of polygons given an algorithm for simplifying a single polygon P within its polygonal domain $\mathcal{P}$. Thus in essence the problem consist of computing $\mathcal{P}$ for each marked polygon P. We assume the family of polygons is given by a contour map $\mathcal{M}$ represented by a sequence of line segments such that the clockwise ordered segments in each polygon P of $\mathcal{M}$ appear consecutively in the sequence, and a topology tree $\mathcal{T}$ given as a sequence of edges $(P,\hat{P})$ indicating that $\hat{P}$ is the parent of P; all segments in $\mathcal{M}$ of polygon P are augmented with (a label for) P and the BFS number of P in $\mathcal{T}$.

To compute $\mathcal{P}$ for every P we need to retrieve the neighbors of P in $\mathcal{M}$. These are exactly the parent, siblings and children of P in $\mathcal{T}$. Once $\mathcal{P}$ and the simplification P' of P has been computed we need to update $\mathcal{M}$ with P'. We describe an I/O-efficient simplification algorithm that allows for retrieving polygonal domains and updating polygons without spending a constant number of I/Os for each polygon. The algorithm simplifies the polygons across different BFS levels of $\mathcal{T}$ in order of increasing level, starting from the root. Within a given level the polygons are simplified in the same order as their parents were simplified. Polygons with the same parent can be simplified in arbitrary (label) order. The following paragraphs describe the steps of our algorithm. In the first step, we reorder the polygons in $\mathcal{M}$ such that they appear in the order they will be simplified. In the second step, we describe how to simplify $\mathcal{M}$.

Reordering: We compute the *simplification rank* of every polygon P i.e. the rank of P in the simplification order described above. The simplification rank for the root of $\mathcal{T}$ is 0. To compute ranks for the remaining polygons of $\mathcal{T}$, we sort the edges $(P, \hat{P})$ of $\mathcal{T}$ in order of increasing BFS level of P. By scanning through the sorted list of polygons we assign simplification ranks to vertices one layer at a time. When processing a given layer we have already determined the ranks of the previous layer and can therefore order the vertices according to the ranks of their parents .

Simplifying: Assume that the polygons in $\mathcal{M}$ appear in order of increasing simplification rank. Consider the sibling polygons $P_1, P_2 \ldots P_k$ in $\mathcal{M}$ all sharing the same parent P in $\mathcal{T}$. The polygonal domains of these sibling polygons all share the polygons $P, P_1, P_2 \ldots P_k$. We will refer to these shared polygons as the

open polygonal domain of P and denote them $\mathcal{P}_{open}(P)$. It is easily seen that $\mathcal{P}$ for P_i where $i = 1 \ldots k$ is equal to $\mathcal{P}_{open}(P)$ together with the children of P_i.

When traversing the polygons of $\mathcal{M}$ in the order specified by their simplification ranks, we will refer to P as an *unfinished polygon* if we have visited P but not yet visited all the children of P. During the traversal we will maintain a queue Q containing an open polygonal domain for every unfinished polygon. The algorithm handles each polygon P as follows; if P is the root of $\mathcal{T}$ then $\mathcal{P}$ simply corresponds to the children of P which are at the front of $\mathcal{M}$. Given $\mathcal{P}$ we invoke the polygon simplification algorithm to get P'. Finally, we put $\mathcal{P}_{open}(P')$ at the back of Q. If P is not the root of $\mathcal{T}$, it will be contained in the open polygonal domain $\mathcal{P}_{open}(\hat{P})$. Since $\hat{P}$ is the unfinished polygon with lowest simplification rank, $\mathcal{P}_{open}(\hat{P})$ will be the front element of Q. If P is the first among its siblings to be visited, we retrieve $\mathcal{P}_{open}(\hat{P})$ from Q, otherwise it has already been retrieved and is available in memory. To get $\mathcal{P}$, we then retrieve the children of P from $\mathcal{M}$ and combine them with $\mathcal{P}_{open}(\hat{P})$ (if P is a leaf then $\mathcal{P} = \mathcal{P}_{open}(\hat{P})$). Finally, we invoke the polygon simplification algorithm on P and $\mathcal{P}$ to get P' and put the open polygonal domain of P' at the back of Q.

Analysis: Both reordering and simplifying is done with a constant number of sorts and scans of $\mathcal{M}$ and therefore require $O(\text{Sort}(|\mathcal{M}|))$ I/Os.

5 Internal Simplification Algorithm

In this section we present our simplification algorithm which is given a single polygon P to simplify along with its polygonal domain, $\mathcal{P}$. The output P' is a valid simplification of P and is thus homotopic to P in $\mathcal{P}$.

Simplifying P: We first briefly describe how to compute a simplification Q^* of a path or polygon Q such that Q^* is homotopic to Q in $\mathcal{P}$, based on Douglass-Peucker's algorithm [12]. We refer to Appendix C for more details. The recursive algorithm is quite simple, given Q and a sub-path Q_{ij} for $i < j + 1$ to simplify, the algorithm finds the vertex p_k in Q_{ij} maximizing the error $\varepsilon(i, j)$. It then inserts vertex p_k in the output path Q^* and recurses on Q_{ik} and Q_{kj}. When the error is sufficiently small, i.e. $\varepsilon(i, j) < \varepsilon_{xy}$ we check if the segment of Q^* is homotopic to P_{ij}. If this is the case the recursion stops. The output path Q^* is homotopic to Q since by construction every segment $p_i p_j$ of Q^* is homotopic to Q_{ij}. This implies that Q^* is strongly homotopic to Q which again implies that Q^* and Q are homotopic. Note that when the input to the algorithm above is the polygon P the output is also a polygon P^*. Even though P^* is homotopic to P, it is not necessarily a valid simplification since P^* may not be simple. Thus after the conclusion of the algorithm above we turn P^* into a simple polygon P' which is the final valid simplification of P output from the simplification algorithm. This is done by finding all segments s of P^* that intersect and add more vertices from P to those segments using the same algorithm as above. Once this has been done we check for new intersections and keep doing this until no more intersection are found.

Checking segment-sub-path homotopy: To check if a segment p_ip_j is homotopic to P_{ij} it is important to be able to navigate the space around $\mathcal{P}$. Since $\mathcal{P}$ is given as a set of ordered simple polygons, we can efficiently and easily compute its trapezoidal decomposition $\mathcal{D}$ using a simple sweep line algorithm on the segments of $\mathcal{P}$. We refer to Appendix C for a precise definition of $\mathcal{D}$ and the algorithm that computes it. We use the ideas of Cabello *et al.* [7] but arrive at a simpler algorithm by taking advantage of $\mathcal{D}$. We define the *trapezoidal sequence* of a path $t(Q)$ to be the sequence of trapezoids traversed by Q, sorted in the order of traversal. If two paths have the same trapezoidal sequence they are homotopic, this can be proven by the argument similar to the ones in [7, 17]. It can be shown that if $t(Q)$ contains the subsequence $tt't$ for trapezoids $t, t' \in \mathcal{D}$ then this subsequence can be replaced by t without affecting Q's homotopic relationship to any other path, we call this a *contraction* of $t(Q)$. By repeatedly performing contractions on the sequence $t(Q)$ until no more contractions are possible we get a new sequence $t_c(Q)$. We call this sequence the *canonical trapezoidal sequence* of Q. Q and Q' are homotopic if and only if $t_c(Q) = t_c(Q')$ [7, 17]. This suggests an easy algorithm for checking if two paths are homotopic: Simply compute and compare their canonical trapezoidal sequences. Note however that the size of $t(Q')$ and $t_c(Q')$ is not necessarily linear in the size of the decomposition $\mathcal{D}$. In our case, we are interested in checking an instance of the strong homotopy condition, i.e. if a candidate segment $s = p_ip_j$ is homotopic to $Q_{i,j}$. Since s is a line segment $t_c(s) = t(s)$ and we check directly if s traverses the set of trapezoids in $t_c(Q_{ij})$ without first computing $t(Q_{ij})$. This is easily done by traversing $\mathcal{D}$ for s and P_{ij} simultaneously, which can be done in $O(|t(Q_{ij})|)$ time.

6 Experiments

In this section we describe the experiments performed to verify that our algorithm for computing and simplifying a contour map $\mathcal{M}$ performs well in practice.

Implementation: We implemented our algorithm using the TPIE environment for efficient implementation of I/O-efficient algorithms, while taking care to handle all degeneracies (contours with height equal to vertices of $\mathbf{M}$, contour points with the same $x-$ or y-coordinate, and the existence of a single boundary contour). The implementation takes an input TIN $\mathbf{M}$ along with parameters ε_{xy} and ε_z, and Δ, and produces a simplified contour map $\mathcal{M}$ with equi-spaced contours at distance Δ.

We implemented one major internal improvement compared to the described algorithm, which results in a speed-up of an order of magnitude: As described in section 5 we simplify a polygon P by constructing a trapezoidal decomposition of its entire polygonal domain $\mathcal{P}$. In practice, some polygons are very large and have many relatively small children. In this case, even though a child polygon is small, its polygonal domain (and therefore also its trapezoidal decompositions) will include the large parent contour together with its siblings. However, it is easy to realize that for a polygon P it is only the subset of $\mathcal{P}$ within the bounding box of

P that can constrain its simplification. Line segments outside the bounding box can be ignored when constructing the trapezoidal decomposition. We incorporate this observation into our implementation by building an internal memory R-tree [15] for each polygon P in the open polygonal domain $\mathcal{P}_{open}(\hat{P})$. These R-trees are constructed when loading large open polygonal domain into memory. To retrieve the bounding box of a given polygon P in $\mathcal{P}_{open}(\hat{P})$, we query the R-trees of its siblings and its parent, and retrieve the children of P as previously.

Data and Setup: All our experiments were performed on a machine with an 8-core Intel Xenon CPU running at 3.2GHz and 12GB of RAM out of which 10GB were available for our experiments. For our experiments we used a terrain model for the entire country of Denmark constructed from detailed LIDAR measurements (the data was generously provided to us by COWI A/S). The model is a $2m$ grid model giving the terrain height for every $2m \cdot 2m$ in the entire country, which amounts to roughly 12.4 billion grid cells. From this grid model we built a TIN by triangulating the grid cell center points. Before triangulating and performing our experiments, we used the concept of topological persistence [13] to compute the depth of depressions in the grid. This can be done I/O-efficiently using an algorithm by Agarwal et. al [3]. For depressions that are not deeper than Δ it is coincidental whether the depression results in a contour or not. In case a contour is created it appears noisy and spurious in the contour map. For our experiments, we therefore raised depressions with a depth less than Δ (we handle peaks similarly by simply inverting terrain heights). Our results show that this can reduce the number of points in the non-simplified contour map with up to 70%. Refer to Appendix D for details.

Experimental Results: In all our experiments we generate contour maps with $\Delta = 0,5m$ since this seems to be widely used in detailed topographic maps. Furthermore, since the LIDAR measurements on which the terrain model of Denmark is based have a height accuracy of roughly $0,2m$, we used $\varepsilon_z = 0,2m$ in the experiments. In order to determine a good value of ε_{xy} we first performed experiments on a subset of the Denmark dataset, consisting of 844.554.140 grid cells and covering the island of Funen. Below we first describe the results of these experiments and then we describe the result of the experiments on the entire Denmark dataset. When discussing our results, we will divide points in the simplified contour map (output points) into ε_z *points* and ε_{xy} *points*. These are the points that were not removed due to respectively the constraint contours and the constraints of our polygon simplification algorithm e.g. ε_{xy}.

Funen dataset: The non-simplified contour map generated from the triangulation of Funen consists of 636.973 contours with 365.641.479 points (not counting constraint contours). The results of our test runs are given in Table 1. The number of output points is given as a percentage of the number of points in the non-simplified contour map (not counting constraint contours). From the table it can be seen that the number of output points drops significantly as ε_{xy} is raised from $0.2m$ up to $5m$. However, for values larger than $5m$ the effect on output size of increasing ε_{xy} diminishes. This is most likely linked with high percentage

ε_{xy} in m.	0,2	0,5	1	2	3	5	10	15	20	25	50
Output points (%)	40,4	23,7	15,2	10,2	8,8	7,9	7,6	7,6	7,6	7,6	7,6
ε_z points	0,8	5,0	13,9	33,5	46,0	59,3	71,7	76,3	78,8	80,4	84,1
ε_{xy} points	99,2	95,0	86,1	66,5	54,0	40,7	28,3	23,7	21,2	19,6	15,9

Table 1. Results for Funen with different ε_{xy} thresholds ($\varepsilon_z = 0.2m$ and with $\Delta = 0,5m$).

Dataset	Funen	Denmark
Input points	365.641.479	4.793.518.863
Contours	636.973	7.260.043
Constraint factor	3,0	3,0
Running time (hours)	1,5	49
Output points (% of input points)	7,9	9,2
ε_z points (% of output points)	59,3	65,7
ε_{xy} points (% of output points)	40,7	34,3
Total number of intersections	38.992	699.027
Contours with intersections (% of input contours)	2,4	3,1

Table 2. Results for Funen and Denmark with $\Delta = 0,5m$, $\varepsilon_z = 0,2m$ and $\varepsilon_{xy} = 5m$.

of ε_z points in the output e.g. for $\varepsilon_{xy} = 10m$ we have that 71.7% of the output points are ε_z points (and increasing ε_{xy} will not have an effect on these).
Denmark dataset: When simplifying the contour map of the entire Denmark dataset we chose $\varepsilon_{xy} = 5m$, since our test runs on Funen had shown that increasing ε_{xy} further would not lead to a significant reduction in output points. Table 2 gives the results of simplifying the contour map of Denmark. The non-simplified contour map consists of 4.793.518.863 points on 7.260.043 contours. Adding constraint contours increases the contour map size with a factor $3,0$ (the *constraint factor*) both in terms of points and contours. In total it took 49 hours to generate and simplify the contour map and the resulting simplified contour map contained 9.2% of the points in the non-simplified contour map (not counting constraint contours). Since $65,7\%$ of the output points were ε_z points, it is unlikely that increasing ε_{xy} would reduce the size of the simplified contour map significantly. This corresponds to our observations on the Funen dataset. Table 2 also contains statistics on the number of self-intersections that is removed after the simplification (as discussed in Section 5); both the actual number of intersections and the percentage of the contours with self-intersections are given. As it can be seen these numbers are relatively small and their removal does not contribute significantly to the running time.

References

1. P. Agarwal, L. Arge, T. Mølhave, and B. Sadri. I/O-efficient algorithms for computing contours on a terrain. In *SCG '08: Proceedings of the 24th Annual Symposium on Computational Geometry*, pages 129–138, 2008.
2. P. Agarwal, L. Arge, T. Murali, K. Varadarajan, and J. Vitter. I/o-efficient algorithms for contour-line extraction and planar graph blocking. In *Proceedings of*

the ninth annual ACM-SIAM symposium on Discrete algorithms, SODA '98, pages 117–126, 1998.

3. P. Agarwal, L. Arge, and K. Yi. I/O-efficient batched union-find and its applications to terrain analysis. Proc. ACM Symposium on Computational Geometry, 2006.
4. A. Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.
5. L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*, pages 313–358. 2002.
6. L. Arge, K. Larsen, T. Mølhave, and F. van Walderveen. Cleaning massive sonar point clouds. In *GIS '10: Proceedings of the 18th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems*, pages 152–161, 2010.
7. S. Cabello, Y. Liu, A. Mantler, and J. Snoeyink. Testing homotopy for paths in the plane. In *SCG '02: Proceedings of the eighteenth annual symposium on Computational geometry*, pages 160–169, 2002.
8. H. Carr, J. Snoeyink, and M. van de Panne. Flexible isosurfaces: Simplifying and displaying scalar topology using the contour tree. *Computational Geometry*, pages 42–58, 2010. Special Issue on the 14th Annual Fall Workshop.
9. A. Danner, T. Mølhave, K. Yi, P. Agarwal, L. Arge, and H. Mitasova. TerraStream: From elevation data to watershed hierarchies. In *GIS '07: Proceedings of the 15th Annual ACM International Symposium on Advances in Geographic Information Systems*, pages 1–8, 2007.
10. M. K. David Kirkpatrick and R. Tarjan. Polygon triangulation in o(n log log n) time with simple data-structures. In *Symposium on Computational Geometry'90*, pages 34–43, 1990.
11. M. de Berg, M. van Kreveld, and S. Schirra. A new approach to subdivision simplification. In *Proc. 12th Internat. Sympos. Comput.-Assist. Cartog.*, pages 79–88, 1995.
12. D. Douglas and T. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature, 1973.
13. H. Edelsbrunner, D. Letscher, and A. Zomorodian. Topological persistence and simplification. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 454–463, 2000.
14. M. Garland and P. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '97, pages 209–216, 1997.
15. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. SIGMOD International Conference on Management of Data*, pages 47–57, 1984.
16. P. S. Heckbert and M. Garland. Survey of polygonal surface simplification algorithms. Technical report, CS Dept., Carnegie Mellon U. to appear.
17. J. Hershberger and J. Snoeyink. Computing minimum length paths of a given homotopy class. *Comput. Geom. Theory Appl.*, pages 63–97, 1994.
18. F. P. M. Garey, David Johnson and R. Tarjan. Triangulating a simple polygon. *Inf. Process. Lett.*, pages 175–179, 1978.
19. A. Saalfeld. Topologically consistent line simplification with the douglas peucker algorithm. In *Geographic Information Science*, 1999.
20. R. Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Comput. Geom. Theory Appl*, pages 51–64, 1991.
21. J. Vitter. External memory algorithms and data structures: Dealing with MASSIVE data. *ACM Computing Surveys*, pages 209–271, 2001.

A Details from Section 2

Two paths Q and Q' are *homotopic* with regards to a polygonal domain $\mathcal{P}$ if there exists a continuous mapping $f : [0,1] \times [0,1] \to \mathbb{R}^2 \setminus \mathcal{P}$ morphing Q to Q', see Figure 1. In other words if $Q(\cdot)$ and $Q'(\cdot)$ represents the continuous curves (parametrized in $[0,1]$) defined by the paths Q and Q' then the following should hold for f:

$$f(0,t) = Q(t) \text{ and } f(1,t) = Q'(t), \text{ for } t \in [0,1],$$
$$f(\alpha,0) = Q(0) = Q'(0) \text{ and } f(\alpha,1) = Q(1) = Q'(1), \text{ for } \alpha \in [0,1].$$

B Details from Section 3

Computing the topology tree $\mathcal{T}$ of $\mathcal{M}$: We use a plane-sweep algorithm to construct the edges of $\mathcal{T}$ from the sorted line segments in $\mathcal{M}$. During the algorithm we will also compute the BFS number of each polygon P in $\mathcal{T}$. After the algorithm it is easy to augment every segment in $\mathcal{M}$ with the BFS number of the polygon P it belongs to in a simple sorting and scanning step.

For a given $\mu \in \mathbb{R}$, let y_μ be the horizontal line through μ. Starting at $\mu = y_\infty$, our algorithm sweeps the line y_μ though $\mathcal{M}$ in the negative y-direction by scanning a list L of the vertices in $\mathcal{M}$ (except for those in $\mathcal{U}$) sorted by increasing y-coordinate. Assume that a vertex v in L is augmented with the (label) of the polygon P to which it belongs, and its immediate predecessor $\text{pred}(v)$ and successor $\text{succ}(v)$ in the cyclic clockwise order of P. We can easily construct L from the segments in $\mathcal{M}$ in a simple scan and sort step. During the sweep we maintain the set of line segments that intersect y_μ in a search tree $\mathcal{S}$, ordered by the x-coordinate of the intersection between the segment and y_μ. Each line segment uv in $\mathcal{S}$ is augmented with $\text{pred}(u)$, $\text{succ}(u)$, $\text{pred}(v)$ and $\text{succ}(v)$, as well as with the parent polygon of the polygon P in $\mathcal{M}$ containing uv. Additionally, we maintain a dictionary $\mathcal{D}$ over the set of (labels of) polygons intersection y_μ. The dictionary $\mathcal{D}$ contains the set of tuples $(P, s, \hat{P}, o)$, where $P \in \mathcal{M}$ is any polygon for which there is currently a segment in $\mathcal{S}$ and $\hat{P}$ is the parent of P in $\mathcal{M}$. The integer s denotes how many segments of $\mathcal{S}$ belong to P, and o the BFS number of P in $\mathcal{T}$. Note that since P is a simple polygon, s is always an even positive integer. Initially $\mathcal{D}$ contains just one entry corresponding to the boundary polygon $\mathcal{U}$, without a parent and with BFS number 0.

Now for simplicity, assume that no two vertices of L have the same y-coordinate. Whenever the sweep line y_μ encounters a new vertex v we update $\mathcal{S}$ and $\mathcal{D}$. With v in L we obtain the (label of the) polygon P containing v along with $u = \text{pred}(v)$ and $w = \text{succ}(v)$. Let $\text{low}(v) = \arg\min_{p \in \{u,w\}}(y(p))$ and $\text{high}(v) = \arg\max_{p \in \{u,w\}}(y(p))$ be the lower and highest point of u and v, here $y(p)$ denotes the y-coordinate of point p. Based on the y-coordinate of u and w there are now three different case. (i) u and w lie on different sides of y_μ, i.e. $y(\text{high}(v)) > \mu > y(\text{low}(v))$, corresponding to case (R) and (L) in Figure 3. In this case we can simply delete the old segment $\text{high}(v)v$ from $\mathcal{S}$ and insert the

new segment $v\text{low}(v)$. The parent of the new segment is the same as that of the old segment. (ii) both u and w are above y_μ, i.e. $y(\text{low}(v)) > \mu$, corresponding to case (E1) and (E2) in Figure 3. In this case we simply delete the two segments uv and wv from $\mathcal{S}$. Unless v is on the boundary $\mathcal{U}$, we locate and delete the entry for P from $\mathcal{D}$. Let $(P, s, \hat{P}, o)$ be this just-deleted entry. If s is larger than 2, there are still segments from P intersecting y_μ and we insert $(P, s-2, \hat{P}, o)$ into $\mathcal{D}$. However, if $s = 2$ (this can only happen in case (E1)), v must be the bottom-most vertex of P and we do not insert anything into $\mathcal{D}$. At this point we output $(P, \hat{P})$ as an edge of the topology tree $\mathcal{T}$, along with the BFS number of P. (iii) both u and w are below y_μ, i.e. $y(\text{high}(v)) < \mu$, corresponding to case (B1) and (B2) in Figure 3. In this case we need to insert two new segments vu and vw into $\mathcal{S}$. If v is on $\mathcal{U}$ we are done, since $\mathcal{U}$ has no parent and $\mathcal{D}$ was initialized with an entry for $\mathcal{U}$. Otherwise we update $\mathcal{D}$, but to do that we need to find the parent polygon of P. This is found by a look-up in $\mathcal{D}$. If P is not in $\mathcal{D}$, v must be the top-most vertex of P. Since P is a simple polygon this implies that we are in situation (B1), in other words P^i is beneath y_μ. In this case we find the predecessor segment ab of vu in $\mathcal{S}$, this predecessor exists because v is not in $\mathcal{U}$. We observe that the polygon P' containing a is either the parent of P, or a sibling of P. By looking at the relative order of a and b and their predecessor and successors stored in $\mathcal{S}$ we can easily determine which is which by a similar case analysis as the one depicted in Figure 3. If it is a sibling we look up P' in $\mathcal{D}$ and get a new polygon P'' and breadth-first rank o'' corresponding to the parent P' and we insert $(P, 2, P'', o''+1)$ into $\mathcal{D}$, otherwise we simply insert $(P, 2, P', o'+1)$ where o' is the breadth-first rank of the polygon associated with P' returned by $\mathcal{D}$ initially.

C Details from Section 5

Simplifying P: We can describe how to compute a valid simplification P' of P using a modified version of Douglass-Peucker's simplification algorithm [12]. The modifications ensure that the output simplification P' is homotopic to P. Although the algorithm is initially invoked on polygon P, we describe it in terms of a path Q.

The algorithm is given indices i and j, $1 \leqslant i < j \leqslant n$ of vertices in some path Q. We maintain a list of vertices which at the end will be the output simplification Q^*, initially Q^* consists of p_1 and p_n. We use a recursive procedure that works on a sub-path Q_{ij}. The procedure has two steps and is invoked initially using $i = 0$ and $j = n$ as the parameters. In the first step (1) we check if $\varepsilon(i, j) < \varepsilon$, if that is the case we check whether segment $p_i p_j$ is homotopic to Q_{ij}. If both checks succeed we do not need to further simplify P_{ij} and we are done. Otherwise we proceed to step (2) and find the vertex internal to $Q_{i,j}$ that maximizes the error $\varepsilon(i, j)$, i.e. $k = \arg\max_{i<k<j}(d(i, j, p_k))$. We add vertex the p_k to Q^* and recurse on (i, k) and (k, j). Note that at the first step when $Q_{1n} = Q$, $\varepsilon(1, n)$ is not necessarily well-defined since the segment $p_1 p_n$ is degenerate if Q is a closed path (i.e. a polygon) — on this case we skip the test in step (1) and go directly

to step (2). The algorithm terminates since any segment is homotopic to itself with error 0 and $0 < j - i$ strictly decreases in the recursion. The output path Q^* is homotopic to Q since by construction every segment $p_i p_j$ of Q^* is homotopic to Q_{ij}. This implies that Q^* is strongly homotopic to Q' which again implies that Q^* and Q are homotopic.

Note when the input to the algorithm above is the polygon P the output is also a polygon P^*. Even though P^* is homotopic to P, it is not necessarily a valid simplification since P^* may not be a simple polygon, see Figure 4(b). Thus after the conclusion of the algorithm above we turn P^* into a simple polygon P' which is the final valid simplification of P output from the simplification algorithm. In the remainder of this section we will describe how this is done, and also describe how we perform the segment-sub-path homotopy check in step (1) of the algorithm above.

The trapezoidal decomposition of $\mathcal{P}$: For any point $p \in \mathcal{P}$, we imagine a ray σ^+ in the positive y-direction starting at p. This ray may intersect multiple segments from $\mathcal{P}$, and we define $p^\uparrow$ to be the segment whose intersection point with σ^+ is closest to p along σ^+. Similarly we imagine a ray σ^- in the negative y-direction and define $p^\downarrow$ to be the closest intersection point, if such a point exists. The *trapezoidal decomposition* $\mathcal{D}$ of $\mathcal{P}$ consists of the segments $pp^\downarrow$ and $pp^\uparrow$ for all points $p \in \mathcal{P}$ where these are defined, as well as the segments from $\mathcal{P}$. These segments can, however, be split up: If a segment $s = uv$ from $\mathcal{P}$ is hit by a ray σ^+ from point p it is split into two sub-segments $up^\uparrow$ and $p^\uparrow v$, and similar for $p^\downarrow$ points. If s is hit by multiple such rays it is split into multiple segments. We note that $\mathcal{D}$ is a planar graph with $O(|\mathcal{P}|)$ edges and faces and that every face of $\mathcal{D}$ is a trapezoid defined by two upper segments from $\mathcal{D}$ and two vertical segments. Note that in the case of degeneracies a face may become triangular.

Computing $\mathcal{D}$: There already exist several algorithms for constructing trapezoidal decompositions of a set of line segments in the plane (e.g. [20, 10, 18]), however since we only need to add vertical segments to an already existing polygonal domain, constructing $\mathcal{D}$ is not hard and can be done by a sweep-line algorithm scanning the segments of $\mathcal{P}$ in the positive x-direction. Let μ be the current x position and x_μ the vertical line $x = \mu$. We maintain a binary search tree $\mathcal{S}$ containing the segments currently intersecting x_μ, this can be done using the same idea as in Section 3. When x_μ encounters some segment endpoint v we delete/insert vu and vw, where $u = \text{succ}(u)$ and $w = \text{pred}(u)$ into $\mathcal{S}$ as appropriate (similar to Figure 3 rotated 90 degrees). Furthermore, we can find $p^\uparrow$ by finding the predecessor of p in $\mathcal{S}$, and $p^\downarrow$ by finding the successor of p. The complexity of this algorithm is $O(|\mathcal{D}| \log |\mathcal{D}|)$ and it uses no I/O's since it assumes that $\mathcal{D}$ is given in memory. We store $\mathcal{D}$ in such a way that we can easily traverse the edges and faces of this planar graph in time linear in the length of the traversed path. This can easily be achieved using a number of standard data-structures.

Removing Self intersections: The simplification algorithm outputs a polygon P^* which is not necessarily simple. Here we describe how we turn P^* into a valid simplification P' which is the final outputted version of the original polygon P.

There are various ways of dealing with this problem [19] but we use a simple algorithm that is easy to implement and works well in practice. First we use a standard line-segment intersection finding algorithm to report the a list of segments $s_1, \ldots, s_m$ of P^* involved in intersections, see Figure 4(b). If $k = 0$, P^* is already a valid simplification, and we are done. Otherwise we invoke our simplification algorithm described in the beginning of this section on each segment $s_t = p_i p_j$ and force it to go through at least one step of the recursion, i.e. we find the vertex $p' \neq p_i, p_j$ among the vertices of P_{ij} maximizing $\varepsilon(i, j)$ and insert this point into P^* and recurse as needed on the two new segments $p_i p'$ and $p' p_j$ that may no longer be homotopic to their corresponding sub-paths. Once this has been done for all segments $s_1, \ldots, s_k$, we re-run the line-segment intersection algorithm and repeat the process if new intersections are found. This process stops when no two segments of P^* intersect.

This algorithm has a poor worst case performance of $O(|P^*|^2 \log(|P^*|))$ because every segment might cause an intersection and thus an iteration of the sweep line algorithm. However we expect it to work well in practice since contours are relatively well behaved.

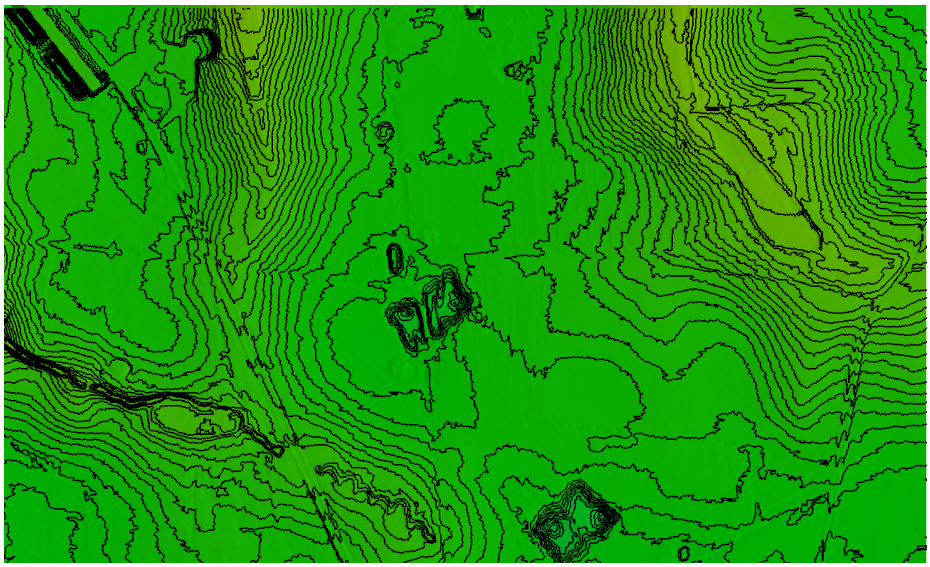
D Details from Section 6

Topological persistence [13] of a triangulation matches each minimum vertex v to a higher saddle vertex w and assigns a *persistence value* $\pi(v)$ to v; where $\pi(v)$ is defined as the height difference between w and v. For a given pair of minimum and saddle vertices, we can define the corresponding *depression* of the terrain as the maximal connected component containing v and vertices with heights between $h(v)$ and $h(w)$. Intuitively, the persistence value of a depression can seen as the depth of the depression and a depression will correspond to a contour in all ℓ-level sets with $h(v) \leqslant \ell \leqslant h(w)$. The definitions above also applies to *peaks* in the terrain by simply inverting the terrain heights.

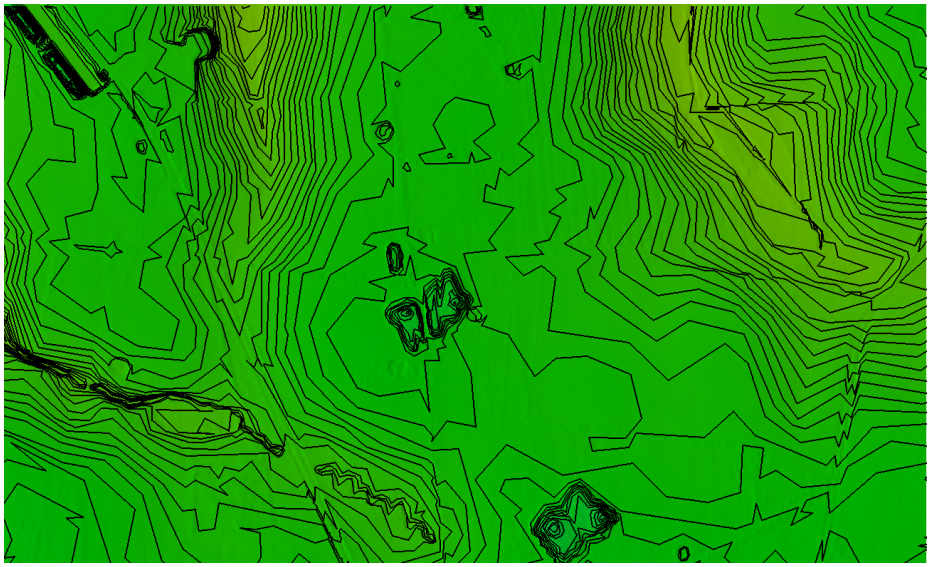
When generating a contour map of level-sets for every Δ meter it is purely coincidental whether a depression/peak with persistence value less than Δ will result in a contour or not. Contours corresponding to depressions/peaks that have persistence value less than Δ, often appear as small spurious contours spread around the contour map. Refer to Figure 5. For all our experiments we have used an I/O-efficient algorithm given by [3] to compute the persistence value of all depression/peaks and subsequently raised/lowered depressions/peaks with persistence value less than Δ.

E Samples From Our Dataset

In Figure 6 and 7 we illustrate a subset of a contour map simplified with different ε_{xy}. The contours were made using $\Delta = 0,5m$ and $\varepsilon_z = 0,2m$.



Original contour map $\mathcal{M}$.



Simplified contour map.

Fig. 2. A snapshot of contours generated from the Danish island of Als. The contour granularity is $\Delta = 0.5$m, the xy-constraints was $\varepsilon_{xy} = 4$m and the vertical constraint was $\varepsilon_z = 0.2$m.

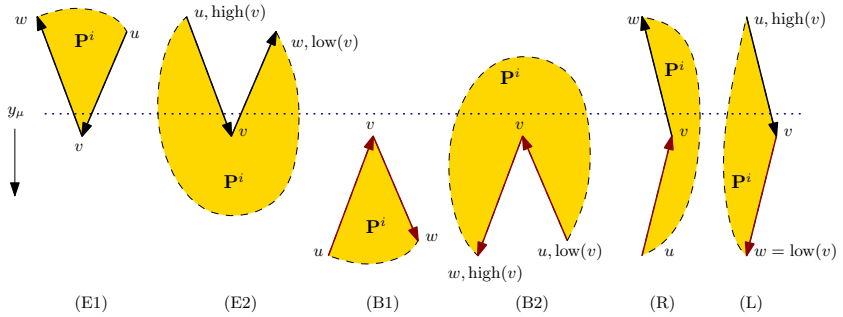


Fig. 3. The possible events that occurs when the sweep-line y_μ encounters vertex v on polygon P. The vertex $u = \text{pred}(v)$ precedes v in the clockwise ordering of the vertices of P, and $w = \text{succ}(v)$ succeeds it. The solid-colored region denotes P^i i.e. the interior of P. The edges intersecting y_μ are stored in $\mathcal{S}$ and are tagged with their parent polygon.

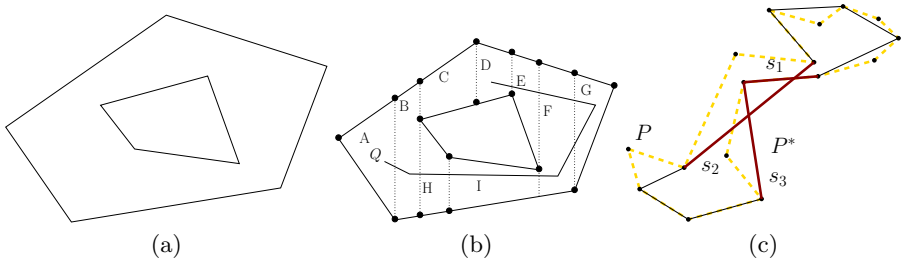


Fig. 4. (a) The empty polygonal domain $\mathcal{P}$. (b) The trapezoidal sequence of the path Q is $t(Q) = ABHIFGFED$. It can be contracted to the canonical trapezoidal sequence $t_c(Q) = ABHIFED$. (c) The output from the recursive simplification algorithm P^* contains three intersecting segments s_1, s_2 and s_3 which will be simplified recursively.

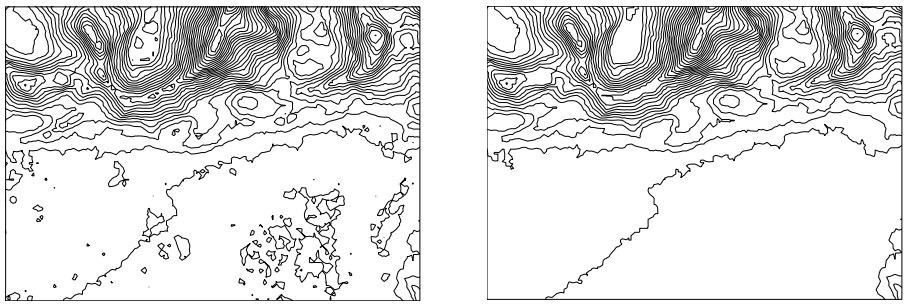


Fig. 5. A small sample of the non-simplified contour map for an area around the Danish island Læsø. To the *left*, no depressions or peaks have been removed. To the *right* depressions and peaks with persistence value less than $0.5m$ have been removed.

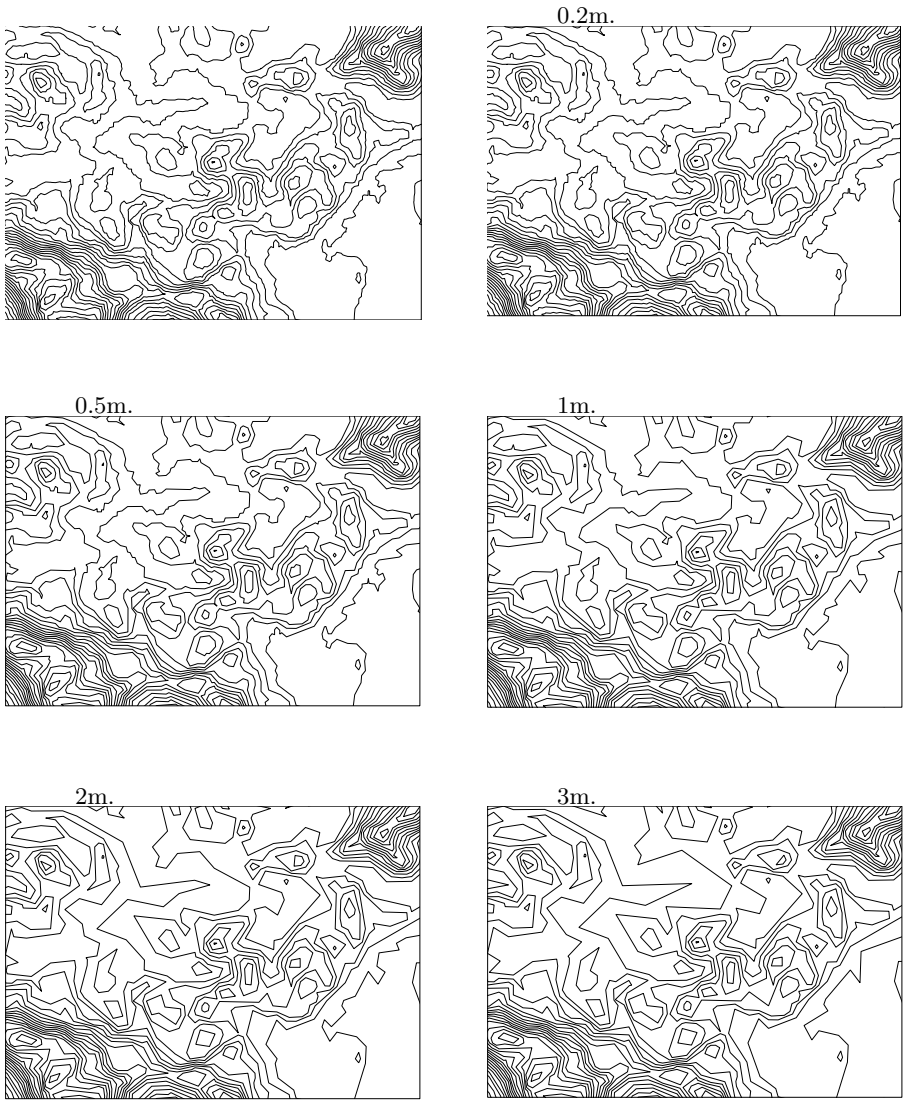


Fig. 6. The top *left* figure shows the original curves while the remaining figures show the contour map subset for different values of ε_{xy}.

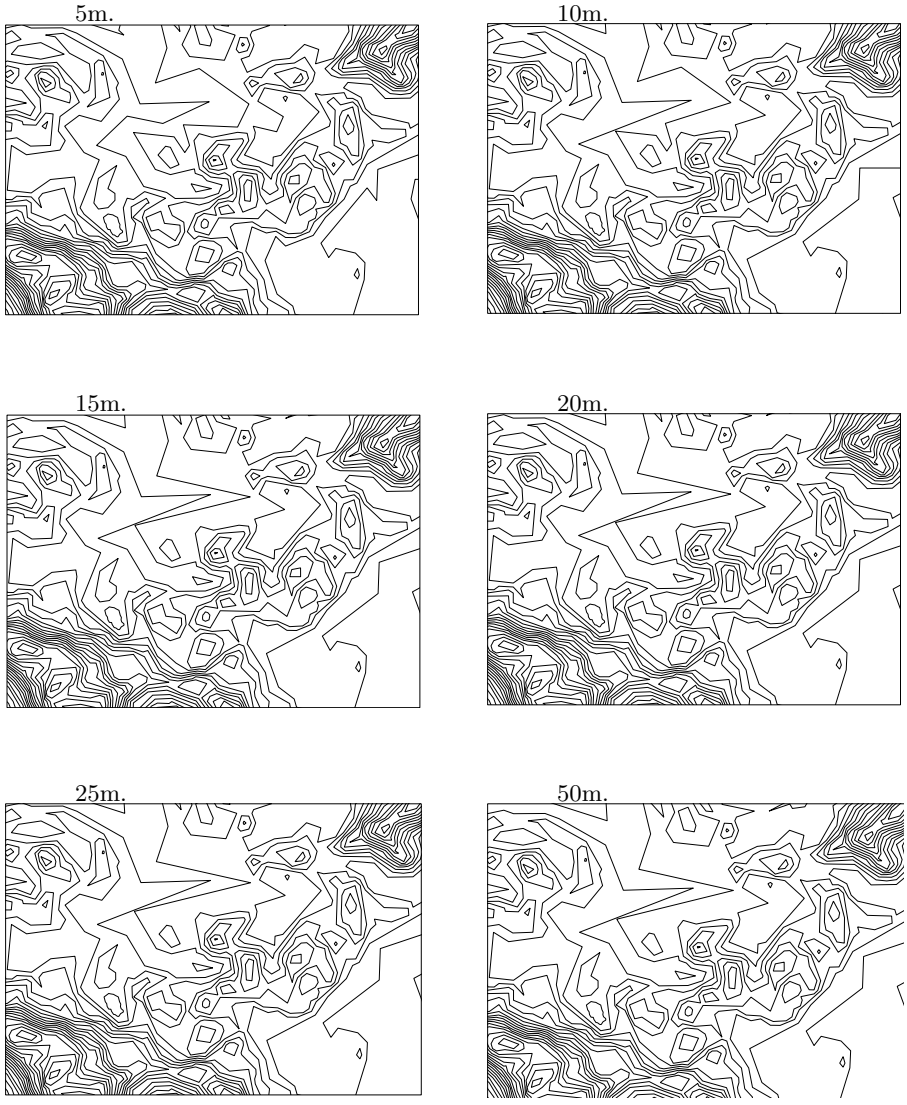


Fig. 7. Shows the contour map subset for different values of ε_{xy}.