I/O-efficient removal of noise from terrain data

van Walderveen, F.

Award date:
2009

I/O-efficient removal of noise from terrain data

Freek van Walderveen

Supervisor: Herman Haverkort[1]
External supervisor: Lars Arge[2]

August 2009

[1]Faculteit Wiskunde en Informatica, Technische Universiteit Eindhoven, the Netherlands.
[2]MADALGO, Department of Computer Science, Aarhus University, Denmark.

Contents

Chapter 1

Introduction

Due to the developments in terrain scanning technology during the last two to three decades, large volumes of terrain data can be produced at a high rate. Manual intervention in the process between data acquisition and data application has therefore become a major bottleneck. Especially light detection and ranging (LIDAR) technology, operating lasers from airplanes, has significantly increased the size of digital terrain models of land areas. For scanning the seabed, the shift from single- to multibeam echosounders (MBES) has caused a significant increase in the data collection rate [Mayer 2006]. Recently, ships have even been equipped with a pair of echosounders to obtain even higher accuracy.

Apart from the obvious application of digital terrain models to making height maps and nautical depth charts, these models are also used for terrain analysis. For example water flow analysis on terrains above sealevel can predict erosion and the effects of a rising sealevel. Seabed scans on the other hand are used for analysing the seafloor, for example in the search for oil. Once oil has been found and pipes have been laid for transporting the oil to the shore, it is important to keep these pipes in good condition. Again, seabed scanning provides the necessary information for scheduling maintenance, such as the position of the pipe and the geometry of its surroundings.

1.1 The problem

In order to make further processing feasible, raw data from a scanning mission first needs to be cleaned. This cleaning is necessary because the raw data — supplied as a set of three-dimensional points — includes a lot of noise. First of all, not only features on the terrain or seabed are reported, but also (flocks of) birds in the sky and (shoals of) fish in the water. For most applications these features are unwanted and have to be removed. Spurious measurements form another problem: for example multiple reflections of laser beams in windows can cause trouble for LIDAR scanners and multiple reflections, or refraction in for example gas bubbles can lead to noisy sonar results.

Apart from problems with the scanned environment, also miscalibration of measurement devices, including scanners detecting their own presence, can negatively affect scan results. For sonar systems, local differences in sound speed due to unclear water can also cause imprecise measurements. Finally, inaccuracy of the sensors themselves and the various systems correcting for external problems (such as movement of the ship or airplane) can add to the amount of error in the final result.

One needs to remove this noise to obtain more accurate digital terrain models, usable for further processing and analysis. Currently, terrain data is often cleaned by hand. Where tools exist, they are either too primitive or too slow to cope with huge amounts of data.

1.1.1 Goals

In this thesis, I am interested in identifying different types of noise in terrain data, in particular from multibeam sonar scans of the seabed, and on efficient, automated methods to clean terrain data by removing the noise.

We start by looking at examples of actual noise from real data sets. In two steps, this leads to concrete definitions suggesting which points in an input data set may be considered as noise. First, we look at mathematical models of realistic terrains and noise. Second, we investigate how to define noise in existing frameworks that give some hope for efficient implementations. Although I do not include full proofs of the correct classification of points from data sets that fit the models defined earlier, these models do provide a nice framework for discussing the value of the proposed tools. After giving these definitions, we have a look at algorithms that can efficiently identify points that are noise according to the definitions. I focus on the efficiency of these methods in case the data is so large that it does not fit in main memory. Finally, we have a look at the results of implementations of the proposed algorithms, run on real-world data sets.

The ultimate goal is to get algorithms sophisticated enough to automatically clean up point sets efficiently, but nowadays such tools are often just used to mark sets of points that are likely to be noise (possibly with an indication of how sure the algorithm is that points are noise), and still have a human operator make final decisions.

1.1.2 Assumptions

We assume that the surface we need to reconstruct is a terrain, that is, there are no overhangs or places where there are two patches of the surface on top of each other. Furthermore, we assume that all parts of the surface are connected via smooth paths such that we can distinguish real terrain from noise. We will make these assumptions more formal later (in Section 2.2).

As mentioned, the input data we consider consists of three-dimensional points, but in specific applications it may contain more information such as the angle of the sonar or laser beam with which the measurement was made, and the estimated inaccuracy of the measurement as reported by the sensor hardware. We will not consider this extra information, but adapting the proposed algorithms to take a measure of accuracy into account should not be hard.

1.1.3 Examples

To get some feeling for the types of noise we would like to remove we will have a look at some typical examples of noise in MBES data. The images are made by constructing a surface for the raw point set (how this is done is explained in Section 3.1) and displaying this surface using a perspective projection with shading.

The first and most typical example of noise is caused by fish. Shoals of fish may swim under the scanner and reflect some or all of the soundings. As Figure 1.1 shows, fish may reflect so many pulses from the echosounder that the measured points cluster together around the fish. Some undershoots, points lying far beneath the surface, are also visible in this picture. Note that they cause dark spots on the top side of the surface due to the shading used.

Figure 1.2 shows another type of noise: a long, sharp ridge of *gross errors* is present in the middle and some smaller clusters of outliers appear on the sides. Although the source of these particular errors is unknown, a possible explanation is the miscalibration of the scanning device or a scanner detecting its own presence. Similar noise is shown in Figure 1.3, but here we can also distinguish a feature that is *not* supposed to be removed: the pipeline in the middle of both pictures. A particularly noisy area is viewed from above in Figure 1.4. Even noisier examples exist, but they do not produce very interesting pictures. Finally, Figure 1.5 shows an example of noise of a smaller scale. Note the difference between the noise in the fore- and background; it seems like the area in the foreground has been scanned twice, resulting in slightly offset data.

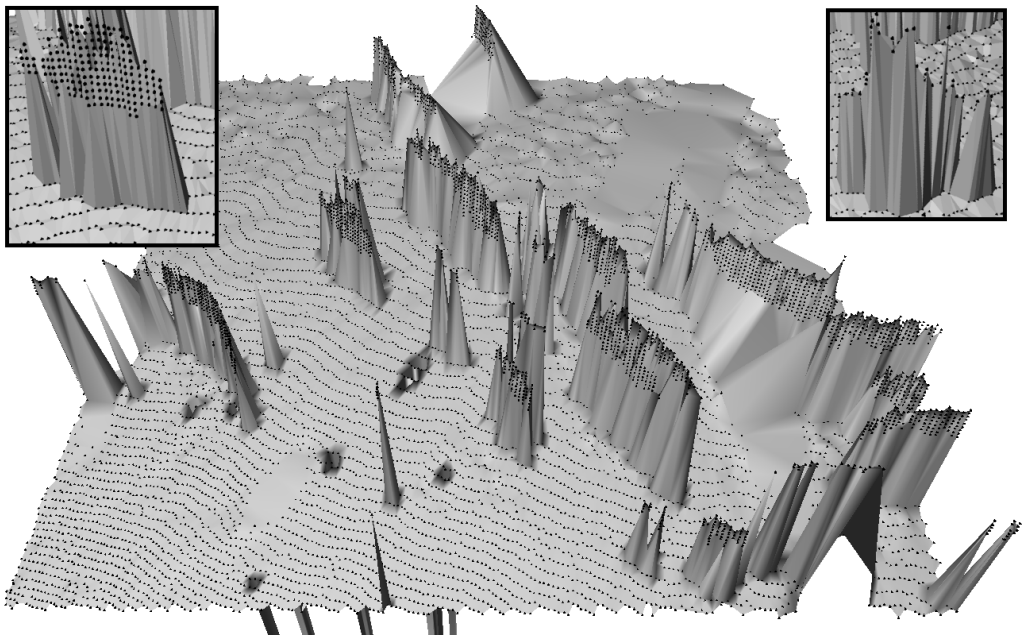


Figure 1.1: Noise caused by fish. Black dots represent the input points. The overlays show the front and back sides of one group of points.
Data from StatoilHydro, patch shown is 12×12 meters.

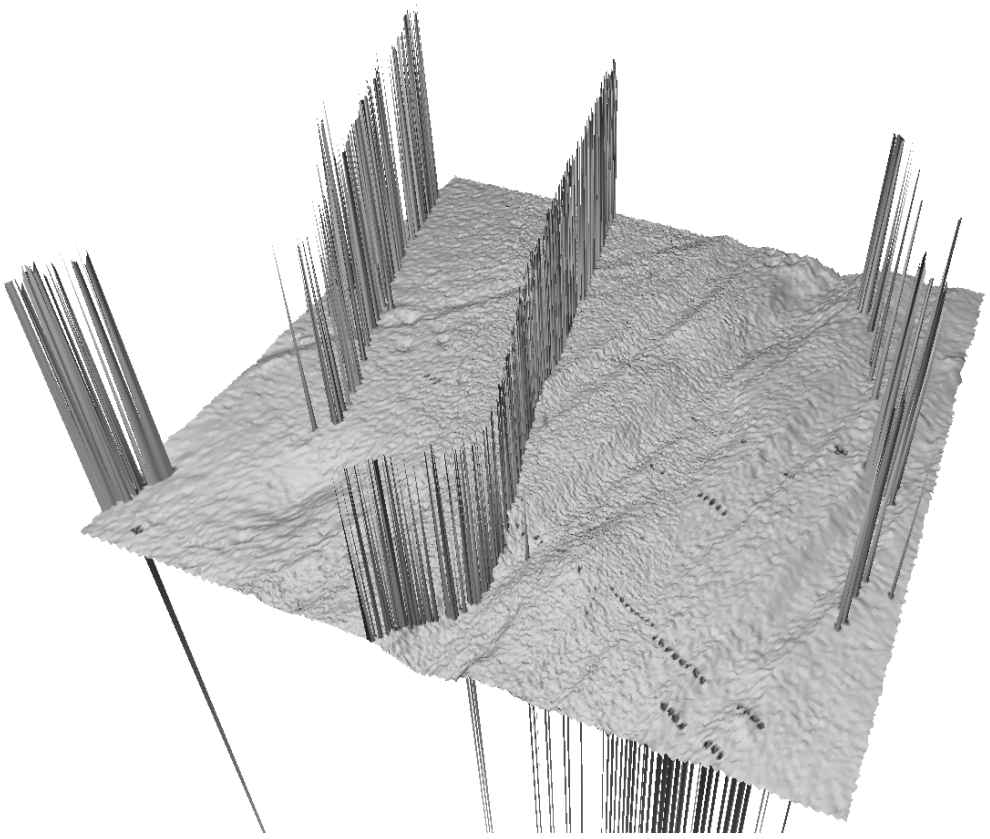


Figure 1.2: Structural and random noise of unknown origin.
Data from StatoilHydro, patch shown is 10×10 meters.

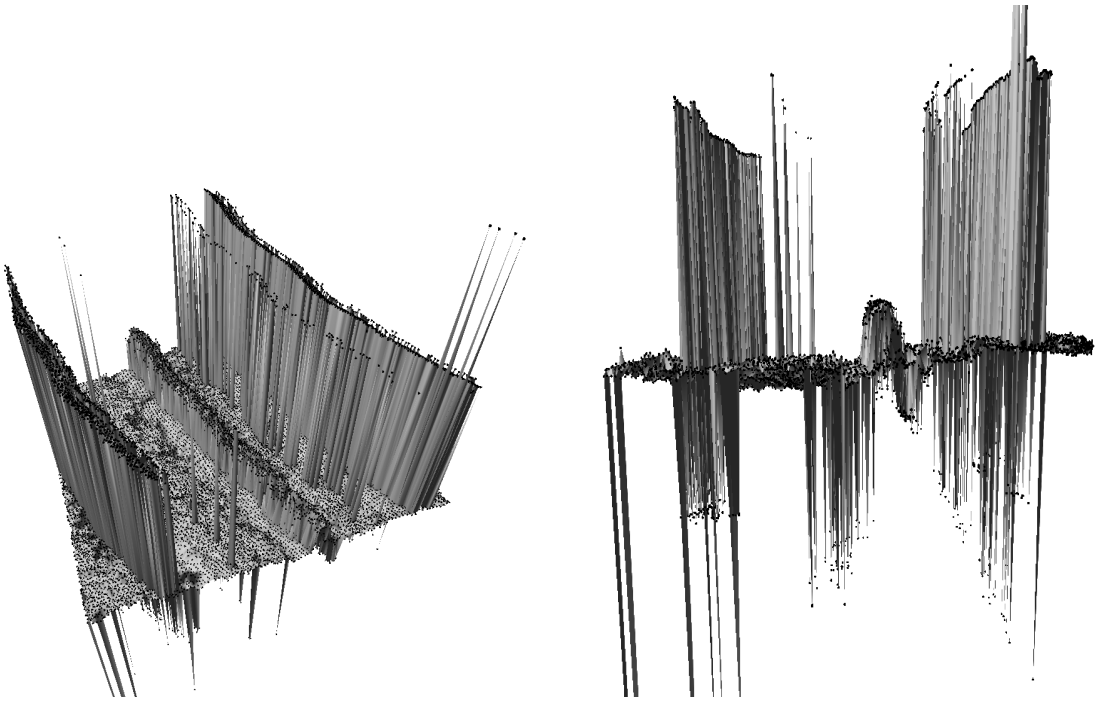


Figure 1.3: Two views of the same area. A pipe is present in the middle of both pictures. Again, the black dots indicate the original data points.
Data from StatoilHydro, patch shown is 3×3 meters.

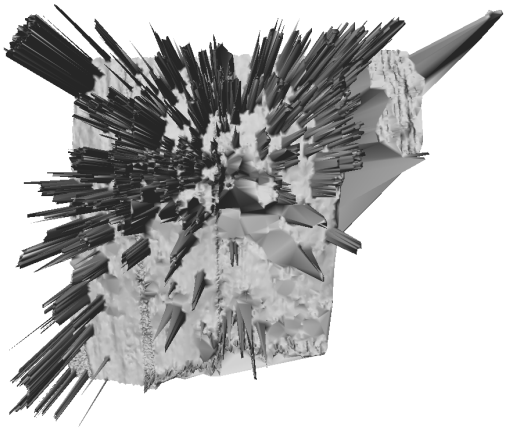


Figure 1.4: Very noisy patch of terrain viewed from above. Data from StatoilHydro, patch shown is 6×6 meters.

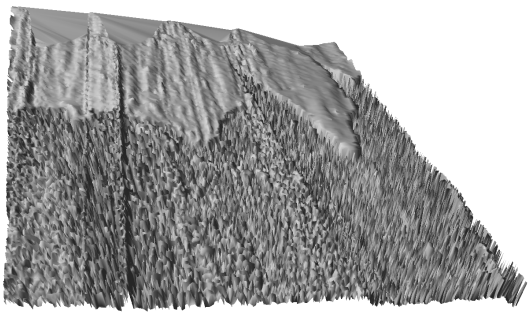


Figure 1.5: Small-scale noise.
Data from StatoilHydro, patch shown is 4×4 meters.

1.2 Previous work

A large number of tools have been developed to aid the — originally fully manual — cleaning of terrain data, and more generally the reconstruction of surfaces from (noisy) point sets. There is a large amount of literature dedicated to heuristic methods, often developed in the remote sensing community. I will not try to give a complete overview of these methods, but rather discuss in more detail some algorithms that are widely used in industry for cleaning MBES and LIDAR data respectively. The methods developed in this thesis should perhaps not only be seen as replacements of these, but also as possible additional pre- or postprocessing that handles specific types of noise not handled well by the other algorithms.

On the theoretical side of the spectrum, relevant work has been done in surface reconstruction algorithms that yield provable relations (topological or by closeness of surface) between the original object and the reconstructed surface. I am not aware of work that does so even for data with gross errors.

Before having a look at these automatic tools, we will first shortly review some standard methods for manual data cleaning.

1.2.1 Manual cleaning

One way of manual cleaning that is employed for MBES data is a procedure based on histograms. The operator demarcates a polygonal region on an approximate terrain. A histogram plots the number of points on that patch of the terrain, for different elevations. The operator can then select an elevation range in which data points should be removed.

This kind of manual cleaning works well for flat areas, but may lead to "features" being removed only partially because it has no sense of the topology of the terrain. Indeed, as shown in Figure 5.6 (page 39), this happens in practice. In this particular case it seems like the operator has tried to remove the more significant outliers on the right side by drawing a polygon around them and happened to select half of the feature on the left, too. This also exemplifies the arbitrariness and subjectivity that is inevitably involved in manual cleaning.

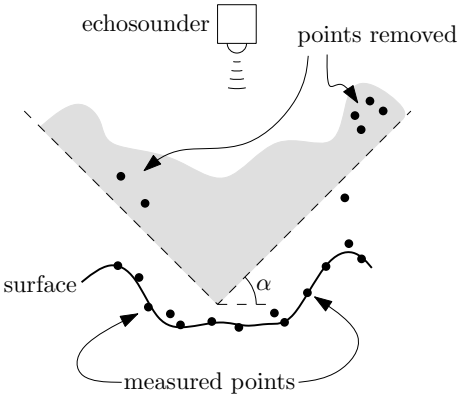


Figure 1.6: An angle gate of $\alpha = 45°$.

Other types of manual cleaning include *angle gates* and *line/swath editing*. Both methods work on a single scan line: the intersection of the point set with a vertical plane orthogonal to the direction the echosounder moves. An angle gate is a very rough way of removing outliers that works by removing all points above a ray starting at a certain point below the echosounder with an adjustable angle to the horizon (see Figure 1.6). Line/swath editing is a more refined way to remove points by mouse interaction, for example by drawing polygons around points to be removed or drawing an approximation of the actual surface to indicate which points should not be removed.

1.2.2 Algorithms with theoretical guarantees

Recently, two approaches to surface reconstruction [Cheng & Poon 2004, Dey & Goswami 2006] have been proposed that come with theoretical guarantees on the relation of the reconstructed surface to the original, even when the sample point set is noisy. Actually, Dey & Goswami go so far as to require, in a sense, a certain amount of noise to be able to guarantee a locally uniform sample distribution (which is necessary to disallow adversary bias in the input data). Cheng & Poon on the other hand propose a set of conditions on their sample sets that they prove (for certain parameters) to be provided by a uniform sampling of a smooth surface. This sampling can be perturbed in the direction of the surface normals by independent uniform random offsets. In

both cases however, the amplitude of the noise that is allowed is bounded: input points should be close to the actual surface.

Both papers prove that the reconstructed surface is homeomorphic to the original surface, and "close" to it. For Cheng & Poon, close means that the vertices of the constructed polygonal surface get arbitrarily close to the original surface as the number of points increases, and secondly the normals get arbitrarily close to the original normals. The reconstruction of Dey & Goswami attains similar closeness for increasingly dense samplings.

1.2.3 CUBE

One of the most used methods in industry for analysing multibeam echosounder data is the CUBE (Combined Uncertainty and Bathymetry Estimator) algorithm [Calder & Mayer 2003, Hall 2006]. The main goal of the algorithm is not so much to remove noise from the data as to give depth estimates at the vertices of a grid laid over the terrain. The vertices of the grid are called *estimation nodes*. Each estimation node includes information on the accuracy of the estimate at that point based on statistical analysis of data points in the neighbourhood of the node. The type of grid is not essential for the algorithm, but an estimation node needs a certain minimum number of data points in its vicinity in order to make the estimation robust against outliers. Data points can however contribute to multiple estimation nodes, so making the grid too fine does not have to cause problems.

The feature that makes this algorithm interesting from a noise-removal point of view is that it is able to make multiple "hypotheses" for the depth of the terrain at each estimation node. The algorithm generates new hypotheses in cases where different measurements that contribute to the same estimation node disagree significantly in terms of vertical position. Earlier methods estimated the depth by just taking the shoalest (highest) estimate. From a nautical perspective, such an estimate is considered safe because it gives vessel operators using maps based on such estimates a better guarantee of the clearance between the keel and the seabed. For other applications (and to some extend also for sea navigation) however, it is more useful to have an estimate that is as close to the true depth as possible.

To obtain multiple hypotheses, CUBE maintains a current set of models (estimates with corresponding variances) at each estimation node, starting with no models and adding new ones as data points arrive that are not compatible with any of the existing models. Existing models are updated with new (compatible) points using a Kalman filter. This filter updates the estimate (minimizing the mean squared error) without requiring access to past data, except for the last state [Welch & Bishop 2006]. Therefore, CUBE only keeps a constant amount of information per estimation node while processing the data.

Interestingly, the algorithm does not implicitly solve the problem of determining which hypothesis is best, though the authors do suggest the following heuristic.

1. If there is only one hypothesis, choose it.
2. If there are multiple hypotheses, use a combination of the following measures for the likelyhood that a hypothesis is correct:
   (a) The more data points supporting a certain hypothesis, the more likely it is chosen (using the logarithm of the number of data points to compare to measure (b)).
   (b) Look at a local neighbourhood and find the closest estimation node for which there is only one hypothesis. The smaller the depth difference between that estimate and the estimate for the current hypothesis, the more likely the current hypothesis is true. Actually, the neighbourhood is chosen not to be a disc (vertical cylinder in 3D) but as an annulus around the estimation node of a certain pre-defined size, because this is said to give better results for burst noise.

This last remark suggests that there is still room for improvement and indeed Calder & Mayer put the problem of determining an appropriate neighbourhood as an open problem and remark that

finding the closest point in the local neighbourhood is time consuming. In practice condition 2b appears not to be used for this reason.

Furthermore, because of condition 1, CUBE always selects "unchallenged" hypotheses, even in cases where a cluster of noisy points lies relatively far away from the rest of the terrain, without any other points at the same horizontal position to supply counterevidence.

Another disadvantage of CUBE, especially when trying to make the algorithm run on-line, is that the models at the estimation nodes need to be "protected" against outliers appearing early in the data stream by moving them forward. This is implemented as a preprocessing filter, delaying real-time processing.

1.2.4 LIDAR data point classification

For LIDAR terrain scans, the main obstacle to obtaining smooth terrain data that can be used for generating for example contour maps, are man-made features and forests. For many purposes points on buildings and trees are undesirable and only points on the ground are to be considered in later processing steps. This makes for an interesting opposite working principle for many existing algorithms for LIDAR data cleaning, as compared to practice in bathymetry: low points are considered more "important" then high points because they are more likely to represent the bare earth, where for bathymetry the high points were taken to be "safe" estimates of the terrain.

Assuming that no other "noise" exists, this principle is used in a number of cleaning methods. For example, terrain models are refined starting with the lowest points of the terrain and iteratively adding lowest points in subregions. Other algorithms classify points by checking whether there are any other points inside a cone or funnel placed under a point: if the cone or funnel is empty, the point is assumed to belong to the bare earth, otherwise it is removed (see Figure 1.7 for an example). For more information on the different algorithms used in the LIDAR community, see the experimental survey of Sithole & Vosselman [2004].

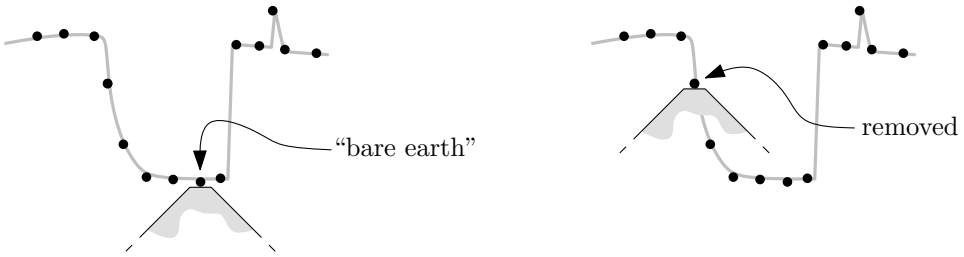


Figure 1.7: Example of a funnel-based classification algorithm.

A different type of cleaning that has been applied to LIDAR data is based on *topological persistence*, a technique explained later in Section 3.2. Agarwal et al. [2006] apply this approach to removing minor depressions that prohibit water flow analysis.

Chapter 2

Defining noise

As we have seen in Chapter 1, many current automated approaches to removing noise do not have a clear a-priori objective of which points they want to remove. Any accurate description of what they regard as noise will invariably end up specifying the exact implementation of the algorithm to actually do the removal, yielding a cyclic answer to the question "What does the algorithm do?".

In order to develop a verifiable approach to recognizing noise — that is, one for which we can check if the algorithm does what we promise it to do — we need a precise definition of noise that we can give without referring to the actual implementation. Then we can prove that a certain algorithm implements a particular kind of noise recognition. Such exact definitions of what we want an algorithm to recognize will be given in the next chapter. These definitions however rely on certain frameworks that were still developed more with implementations in mind than noise recognition.

To bridge the gap between a definition inside a framework and intuitive ideas about what noise is, I propose in this chapter a model of real terrains and noise that can be used as a test case for noise recognition methods. In the next chapter, we can then try to see how far the methods proposed there achieve the goal of correctly classifying point sets that correspond to the models in this chapter, as a function of the parameters of these models.

The models are set up in two steps that mimic real-world data acquisition: we first define a physical model of terrains and noise-inducing features and second define assumptions on sets of sample points from these physical models. In both cases we assume an "orthogonal" view of the world: the physical models are assumed to be mathematical terrains (that is, surfaces without overhangs) and the sample points are assumed to be obtained only by vertical measurements. Before diving into the models, we first try to see if there are any useful divisions to be made in the types of noise found in practice.

2.1 Types of noise

As noted in Chapter 1, we are faced with many different types of noise. We may be able to handle some of those with the same algorithm, but not all. The distinctions made in Chapter 1 were mostly based on the source of the noise. Now, we want to get a more abstract view of these types of noise such that we can group (and handle similarly) noise with different sources but with the same "appearance". The first rough division in noise types that is necessary is that between clear outliers (points that lie far out of the way), and points that are approximately right.

The first type of noise is what we will call "big noise" — points that are often caused by uninteresting physical artefacts such as birds, fish, and air bubbles, or by misconfiguration of the scanner. Most of the time they are obviously wrong to the human eye and should not be considered in any further processing of the data.

The second type of noise is what we will call "small noise" — points that are just a little off

compared to the actual terrain or points in their neighbourhood because of, for example, sensor inaccuracy. These points may either be kept because they are not required to be removed for a particular application, or we might want to move them a little up or down so as to "smooth out" the terrain. This is roughly the type of noise that the algorithms mentioned in Section 1.2.2 can handle.

In the remainder, we will mainly focus on recognizing big noise, but also mention small noise here and there. Specifically, the definitions of noise and recognition methods proposed are geared for big noise and, in contrast to the algorithms with provable guarantees mentioned in the introduction, aim at recognizing noisy points as opposed to reconstructing a surface that is as close as possible to the original.

A further subdivision of big noise that is made by some tools (see for example the book by Li et al. [2004]) is that between single outliers and clusters of gross errors. For single points it often works to look at a local neighbourhood around a point to decide whether it is far off. For larger clusters of erroneous points one either needs to compute an approximation of the entire terrain to find outliers or construct "regional" surfaces as neighbourhoods to compare to. The main problem with these methods is that one has no idea how large such a neighbourhood needs to be to detect all clusters, while generating an approximating surface for a large enough part of the terrain is time-consuming (not to mention the problem of the points in error that are also counting towards the average).

2.2 Noise models

2.2.1 Climbing-free terrains

The first part of the model is concerned with the physical terrain we would like to reconstruct. We cannot hope to be able to reconstruct arbitrary terrains because they may look exactly like noise in whatever definition. Therefore, it seems natural to start by bounding the slope of the terrain.

Definition 2.1 *A point p on a terrain $h : \mathbb{R}^2 \to \mathbb{R}$, $p \in \mathbb{R}^2$ is* slope-bounded *by σ if h is differentiable at p and the length of the gradient at p, $|\nabla h(p)|$, is at most σ.*

Definition 2.2 *A terrain $h : \mathbb{R}^2 \to \mathbb{R}$, is* slope-bounded *by σ if and only if all points $p \in \mathbb{R}^2$ on this terrain are slope-bounded by σ.*

This gives quite some power to prove things about such terrains, but excludes very natural features such as cliffs (see for example Figure 2.1), unless σ is taken arbitrarily high in which case the parameter is pointless. Therefore I propose a relaxed version of the bounded-slope assumption that allows cliffs to a limited extend. For this we first need an auxiliary definition of a "thick" path in the plane (the path needs to be thick because we need to be sure that it gets hit by enough sample points, see also Section 2.2.3).

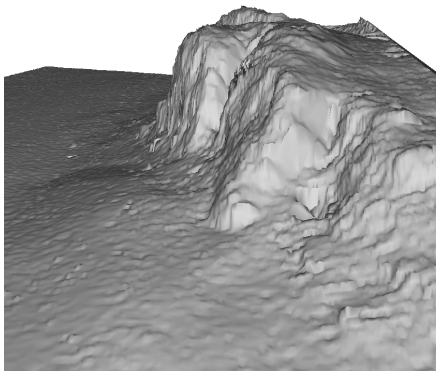


Figure 2.1: A real terrain with a combination of steep slopes and paths with relatively low slope.
Data from StatoilHydro, patch shown is 1×1 kilometer.

Definition 2.3 *A* corridor *$C \subset \mathbb{R}^2$ of width δ is a connected region of the plane that is defined by a simple curve $P \subset \mathbb{R}^2$ and consists of all points $p \in \mathbb{R}^2$ for which there exists a point $q \in P$ such that $|p - q| < \delta$.*

Definition 2.4 *A terrain $h : \mathbb{R}^2 \to \mathbb{R}$ is* climbing free *for width δ and slope σ if and only if for any two points $p, q \in \mathbb{R}^2$ there exists a corridor $C \subset \mathbb{R}^2$ of width δ such that $p, q \in C$ and the lifting $h(C)$ $(= \{(x, y, h((x, y))) \mid x, y \in C\})$ of the corridor to the terrain is slope-bounded by σ at every point on the corridor.*

Intuitively this implies that for any two points on the terrain, there exists a path that a person who is δ meters wide can take without needing climbing equipment if he can handle slopes up to $\sigma \cdot 100\%$ (see also Figure 2.2(a)). Now cliffs are allowed, but it is required that someone standing on the edge of the cliff can walk down to the ground below without needing climbing equipment. On the other hand, we do not allow objects to be part of the terrain that are delimited by steep slopes on all sides, the reason of course being that samplings of such an object look quite like noise. At the same time, we do allow pipe lines as discussed in the introduction as long as they are wider than δ meters and have at some point along their length a path connecting their top to the rest of the terrain (this happens very naturally when the pipe is partially buried under a sandbar).

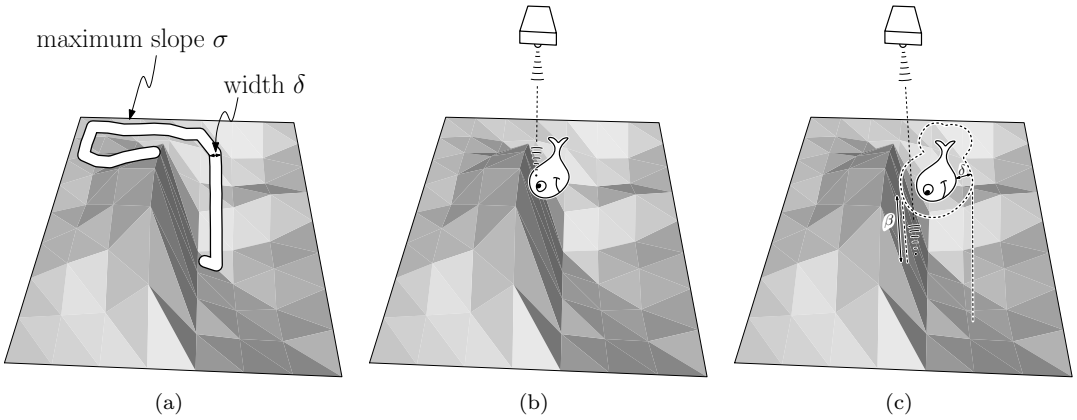


Figure 2.2: (a) An example of a climbing-free corridor connecting two points that would otherwise have a too high height difference. (b) Big noise occurring too close to a cliff may be interpreted as an extension of that cliff. (c) Big noise should be surrounded by a "buffer zone" that is also separated from the terrain.

2.2.2 Big noise

We define big noise using noise-inducing objects that may prevent scanning beams from hitting the real surface. They are modelled as a patch of terrain that is everywhere far away from the original terrain. To avoid confusing noise close to a cliff with an extension of the cliff itself (see Figure 2.2(b)), we also require points on the terrain that are close to noise (in horizontal position) to be far away from the patch of noise (Figure 2.2(c)).

Definition 2.5 *A patch of terrain $h' : R \to \mathbb{R}$, $R \subset \mathbb{R}^2$, is* β-separated noise *for a climbing-free terrain h and width δ if R is a finite connected region, h' is differentiable at every $p \in R$, and for any $p \in R$ and $q \in \mathbb{R}^2$ with $|p - q| < \delta$, it holds that $|h(q) - h'(p)| > \beta$.*

2.2.3 Samples and small noise

In the above we have defined models of physical terrains and noise-inducing objects, but the data delivered by scanning equipment consists only of a set of sample points of which we then have to recognize which parts belonged to the terrain and which to the noise-inducing objects. If we want to guarantee that a recognition method can distinguish noise from terrain data, we need such a sample set to be dense enough in the horizontal direction (consider again the situations in Figure 2.2(b) and (c)).

Definition 2.6 *A sample set $P \subset \mathbb{R}^2$ is* ε-dense *if and only if for any point $p \in \mathbb{R}^2$ there exists a point $q \in P$ such that $|p - q| < \varepsilon$.*

The next step is specifying how the height of a sample point is derived. Here we assume some randomness. For a given climbing-free terrain $h : \mathbb{R}^2 \to \mathbb{R}$ and separated noise $h' : R \to \mathbb{R}$, we first define an indicator random variable I that indicates whether a point in R is taken from h ($I = 0$) or h' ($I = 1$). A given opacity $\alpha \in [0, 1]$ of the noise-inducing object is used as the probability that points are taken from h', so $\Pr[I = 1] = \alpha$. This opacity is introduced to model the differences in noise we see in practice. Some fish, such as those in Figure 1.1, block almost all soundings, while other phenomena cause less dense clusters of noise (see for example Figure 1.4). The sample function $s : \mathbb{R}^2 \to \mathbb{R}$ is then defined as

$$s(p) := \begin{cases} h'(p) & \text{if } p \in R \text{ and } I = 1 \\ h(p) & \text{otherwise} \end{cases} .$$

If we also want to model small noise, we may consider adding it via a random variable J. Because we do not know the real distribution of small noise (that may have many sources), we cannot use it here. For simplicity, J is taken to be distributed according to a uniform distribution $U(-\gamma, \gamma)$, for some parameter γ.

$$s'(p) := \begin{cases} h'(p) + J & \text{if } p \in R \text{ and } I = 1 \\ h(p) + J & \text{otherwise} \end{cases}$$

Chapter 3

Recognizing noise

As alluded to in the last chapter, we will now have a look at some frameworks that can be used to define noise and give definitions of noise inside these frameworks. We will also refer back to the models in the last chapter to validate the proposed definitions and provide some intuition on what may be provable.

3.1 Triangulation

Since we assume that the terrain we would like to reconstruct does not have overhangs, it seems natural to use this assumption while recognizing noise. There are two common ways of representing a terrain as a surface: by fitting the data into a grid or by triangulating the points to create a TIN, a triangulated irregular network. Grids are easier to compute on but we would loose accuracy and be unable to handle the differences in sample rate in different places on the terrain. A triangulation represents the terrain much more accurately because it keeps the original data points and just interpolates the terrain between them. Because we would like to stay as close as possible to the original point set, a triangulation seems to be the best choice. More specifically, we project the point set down to the horizontal plane (ignoring the vertical coordinates of the points) and construct a two-dimensional Delaunay triangulation of the projected point set (a triangulation where no points of the point set lie inside the circumscribed circle of any triangle [Berg et al. 2008]). We lift this triangulation back to the original heights at the vertices to obtain a terrain surface (see Figure 3.1 for an example). This is the surface we then use for further processing.

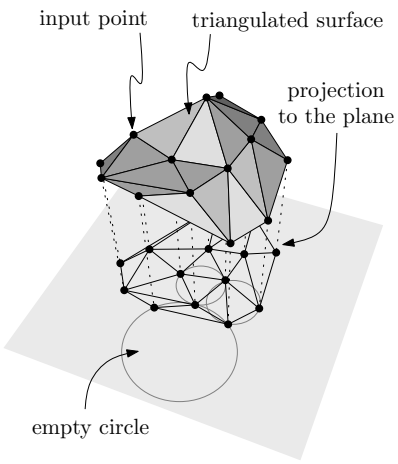


Figure 3.1: The construction of a TIN.

A triangulation also provides a convenient starting point for recognizing noise because we can refer to the neighbours of a point (the ones it shares a triangle edge with), and the local neighbourhood around a point (basically consisting of all triangles it is incident to). In the context of surface reconstruction of for example 2-manifolds, three-dimensional Delaunay triangulations are also popular in the literature (they are for example used in both surface reconstruction algorithms with theoretical guarantees mentioned in the introduction [Cheng & Poon 2004, Dey & Goswami 2006]). We would however like to make processing feasible for large data sets, which is both theoretically and practically problematic if we would first need to construct a three-dimensional Delaunay triangulation. While in the plane the complexity of the triangulation is linear, 3D point sets exist that have a Delaunay triangulation of quadratic complexity. Also, the construction of 3D Delaunay triangulations has not yet been well-studied for large data sets where algorithms cannot

run in main memory (see also Chapter 4), with only heuristic approaches currently available. Two-dimensional Delaunay triangulations have however been widely studied, and algorithms are available that are both theoretically and practically efficient, even when the algorithm cannot run fully in main memory [Agarwal et al. 2005].

3.2 Topological persistence

A framework that has been used earlier for the elimination of local depressions from a terrain (or peaks when viewing the terrain upside down) is topological persistence. Topological persistence is a notion introduced by Edelsbrunner et al. [2003] that tries to capture the inherent hierarchical structure of depressions (pits) in terrains. For example a large valley contains mid-sized depressions which can themselves contain multiple local minima. This recursive structure of pits within pits is seen as the terrain's structure, and Edelsbrunner et al. propose a method to get rid of the most insignificant pits. The original definition is much broader and can for example also be applied to general 2-manifolds, but for simplicity we will only look at it from the perspective of triangulated terrains.

The idea with topological persistence is to "grow" the terrain by sweeping a horizontal plane through it from bottom to top; the part of the terrain that is already passed by the sweep plane grows as the plane progresses. Pits are considered to be features that get born at some point in time and die at some later point. They are born when the sweep plane passes a local minimum of the terrain (a *sink*, see for example Figure 3.2(a)), and die when they get assimilated into a more significant feature. The sink is called the pit's representative because it is the point that started the pit. The standard measure of the significance of a pit is its height, that is, the difference in height between the moment it is born and the moment it dies (other significance measures exist, but we will have a look at those later). Each time the sweep plane hits a *saddle point* of the terrain where two previously unconnected pits merge (such as at point s in Figure 3.2(d)), the most significant one survives and its representative becomes the representative of the bigger pit. The least significant pit dies and its representative is said to be *paired* with the current saddle point.

We assume that the part of the world outside the given terrain (in our case, outside the triangulation, that is, outside the two-dimensional convex hull of the input points) is lower than any other point on the terrain, and we model this as one *global sink*. Hence, any growing pit will at some point be assimilated into the pit represented by the global sink.

We have up to now seen two types of vertices of terrains: minima and saddle points. The other two types one can distinguish are *regular points* and *maxima*. More formally, assuming no two neighbouring vertices have the same height, the type of a vertex v is determined by the structure of its lower link: the neighbours of v in the triangulation that have a lower elevation than v. We look at the connected components these vertices form within the cycle of vertices neighbouring v.

- If the lower link is empty, v is a local minimum.
- If the lower link consists of one component that spans the whole cycle, v is a local maximum.
- If the lower link consists of one component that does not span the cycle, v is a regular vertex.
- If the lower link consists of multiple components, v is a saddle point.

In case the vertex is a saddle point, we can further distinguish by the multiplicity of the saddle point. If there are $k+1$ components in the lower link, the saddle has multiplicity k, meaning that $k+1$ different features merge when the sweep plane passes the saddle point. In the following, we can assume that saddles always have multiplicity one because saddles with multiplicity $k > 1$ can always be unfolded into two saddles of multiplicity i and j, $1 \leq i, j < k$ such that $i + j = k$ [Edelsbrunner et al. 2003].

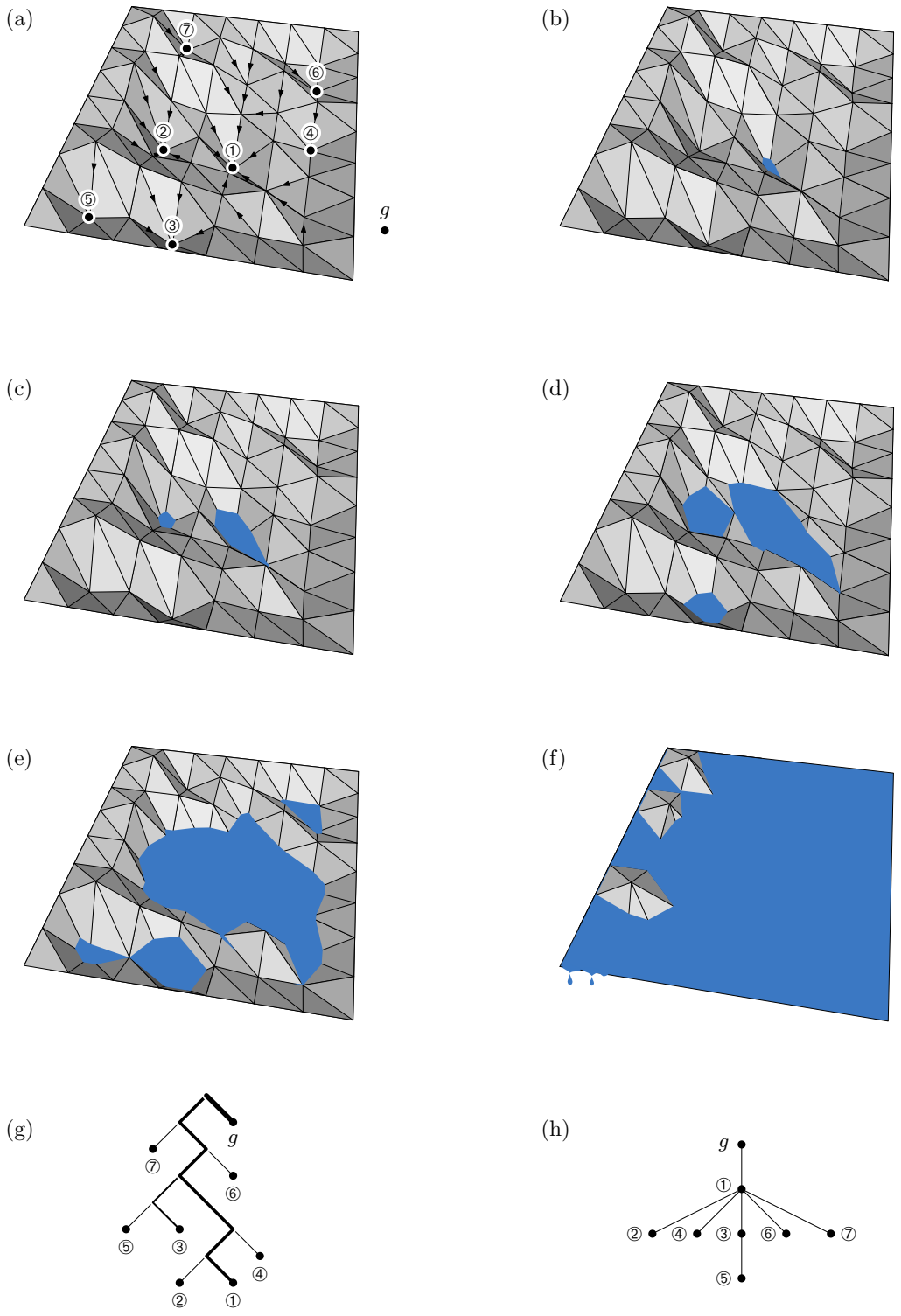


Figure 3.2: (a) A triangulated terrain with dots at local minima (with g representing the global sink), an indication of relative height (1 is lowest), and flow directions. (b)–(f) Rising water level. (g) Topological merge tree. (h) Compact merge tree.

3.2.1 Merge trees

After the sweep is completed we end up with a single pit representing the whole terrain. One can also see this sweep as an imaginary flooding of the terrain by raising the groundwater level. At some point the water does not stay in the ground any more and starts filling the lowest of the pits of the terrain: at this moment a new feature is born (Figure 3.2(b)). The water in this pit continues to rise and in the mean time other pits are starting to fill with water (Figure 3.2(c)). When two neighbouring pits are filled with so much water that it is about to flow over the edge dividing the two lakes, the two pits merge into a larger lake (Figure 3.2(d)). The smallest (for example shallowest) one dies and the largest one survives as its lowest point now becomes the representative of the whole lake.

Topological merge tree. The information we are interested in when defining noise is the order in which the different features merge and which of them die. That is, we are interested in the tree formed by taking all sinks of the terrain as leaves and making internal nodes representing saddle points, with edges from a child feature to a saddle-node if that feature merged together with another feature at that saddle point (see Figure 3.2(g)). This structure is similar to what is known in topology as the *contour tree* of a terrain, but includes only saddle points and local minima, but not local maxima. I will refer to this tree as the (height based) *topological merge tree* and for convenience assume that every node also stores the height of the corresponding terrain vertex (saddle or sink), and the persistence of the sink in case of leaf nodes.

Because we assume all saddles to have multiplicity one, this tree is always binary. It also follows from the definition that the heights of the nodes in the merge tree always decrease on every root-to-leaf path.

Compact merge tree. We will now review an alternative version of the merge tree that is slightly more compact, but more importantly makes some definitions later easier to state. The *compact merge tree* has only sinks as vertices; saddles are represented as edges that connect the vertices representing two pits that are merged together at that saddle (see Figure 3.2(h)). This means we can make such a tree T_C by transforming a topological merge tree T_T in the following way (see Figure 3.3).

Take all leaves of T_T (the sinks) as nodes of T_C. For each leaf s in T_T trace back the path $\langle s, a_1, a_2, \ldots, a_n \rangle$ in the direction of the root until you hit the saddle point a_n where sink s died (that is, $h(s)$ is higher than the lowest sink in the surviving subtree). Now let t_i, $1 \leq i < n$, be the node in T_C that corresponds to the sink dying at saddle a_i (so $h(a_i) > h(s)$), and let s' be the node in T_C corresponding to leaf s. Add a directed edge (t_i, s') in T_C for $1 \leq i < n$. Store with every edge (t_i, s') the height of the corresponding saddle a_i and store with every node s' the height of corresponding leaf s in T_T.

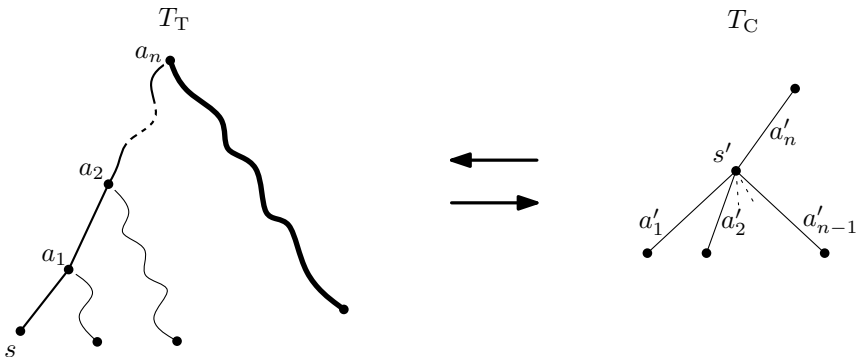


Figure 3.3: Correspondence between the topological and compact merge trees.

Note that, as opposed to T_T, T_C is not a binary tree. In T_C though, the root of the tree is the

global sink while in T_T the root of the tree is the saddle that connected the most significant lake with the global sink.

Lemma 3.1 *The compact merge tree so constructed contains the same information as the original topological merge tree.*

Proof. By construction, the tree does not contain more information, so we only need to proof that we can construct a topological merge tree T'_T from the compact tree T_C such that $T_T = T'_T$.

Analogous to our earlier construction, we take the nodes of T_C to be the leaves of T'_T and add a node to T'_T for every edge in T_C. Then, we order the children of every node in T_C by increasing height of the saddle point that is represented by the edge to that child. For each non-leaf node s' in T_C, consider the ordered sequence of edges to children $(a'_1, a'_2, \ldots, a'_{n-1})$, and edge a'_n from s' to its parent in T_C. Then, create a path of edges (s, a_1), (a_1, a_2), $\ldots$, (a_{n-1}, a_n) in T'_T. Copying the height information from the nodes of T_C to the corresponding leaves of T'_T, and from the edges of T_C to the corresponding nodes of T'_T then completes the construction of T'_T.

We proof by induction on the height of T_T that $T_T = T'_T$. As the leaves are copied verbatim twice, the base case is trivial. Also the internal nodes of T_T are mapped bijectively onto a set of edges in T_C and then back to nodes again in T'_T, while keeping the same height information associated. Hence, we only need to proof that the edges are put back in the right places. For this, first note that for any edge $(a, p(a))$ in T_T we have $h(p(a)) > h(a)$. Therefore, as $(a, p(a))$ is part of exactly one path $\langle s, a_1, a_2, \ldots, a, p(a), \ldots, a_n \rangle$ from sink s to the saddle a_n where s died, the edges in T_C corresponding to a and $p(a)$ will be ordered consecutively below s'. Hence, they will again show up as child and parent in an edge in T'_T. □

3.2.2 Persistence measures

We may look at a pit as the polyhedron formed by the water inside the pit when it is filled up to the height of its saddle point. Perhaps a bit more formally one might describe this polyhedron as the volume $P(v)$ for a sink v that touches v and is bounded from below by the terrain and from above by the horizontal plane $t(v)$ at the height of the saddle point of the pit. The standard measure of the significance of a pit with representative v is then the height of $P(v)$. Now one can define other measures similarly. In his master thesis, Revsbæk [2007] defines a general framework for computing such measures. In particular, he introduces the following new ones:

- *projected area*: the area of the part of the surface of $P(v)$ touching $t(v)$;
- *surface area*: the area of the surface of $P(v)$ *not* touching $t(v)$;
- *volume*: the volume of $P(v)$;
- any combination of the above, for example a tuple of height and volume.

With these measures we can now make different merge trees based on different "survival criteria". Instead of choosing the pit with the lowest sink to survive we may for example take the one with the largest volume. This means we do not only change the information, but also the structure of the compact merge tree. We take a look at a possible application of these extra possibilities in Section 3.3.2.

3.3 Defining noise using persistence

One of the main practical reasons for the development of the topological persistence framework was the need to simplify complexes such as triangulations of spatial point sets [Edelsbrunner et al. 2000]. Later, the approach was also applied to terrains [Edelsbrunner et al. 2003, Agarwal et al. 2006]. Topological simplification was much needed in this domain because for example contour maps generated from terrains with many minor, local depressions (small noise) can contain

superfluous contour lines around these depressions. Also for water flow analysis it is necessary that the terrain does not contain (many) sinks because these are assumed to evaporate all water that touches them, disrupting the water flow (as opposed to the flooding model where the water stays in the sink). In these cases, topological persistence indeed helps to decrease significantly the problems caused by small noise. We will now first see how topological persistence achieves this and then see if we can also use it for removing big noise.

3.3.1 Topological persistence for small noise

The simplification procedure described by Edelsbrunner et al. *cancels* sinks against saddle points, starting with the feature with the lowest persistence. The procedure carries on iteratively until the lowest persistence is higher than a given threshold, effectively removing all sinks with a persistence lower than the threshold. By definition, independent of the significance measure used as survival criterion, in a compact merge tree for a given sink s, any descendant s' is less significant than s. Therefore, if we select all pits with a significance smaller than a certain threshold, we know that if we select s, we also select s'. This is an important consequence for noise removal based on persistence because we can eliminate all sinks with a persistence lower than a given threshold by only pruning subtrees of the compact merge tree. Note that the subtrees of the compact merge tree correspond exactly to what I have been calling "pits" up to now and that the top $t(s)$ of a pit is at the height of the saddle point paired with s which is stored with the edge from s to its parent in the compact merge tree.

Now consider removing a particular subtree. Probably, we do not want to only remove the sinks and saddle points that appear in the tree; we also want to remove the regular points and maxima around those points (consider for example a cluster of points like in Figure 1.1, where only a few of the highest points are sinks when the terrain is turned upside down, while we want to remove all points from a cluster). This is easy to do in the persistence framework. When finding the sinks we can also find the points around there that are in the so-called *watershed* of that sink. The watershed of a sink v is the set of points on the terrain for which rain that falls on those points eventually ends up in v, assuming that rain always goes down the edge to the lowest neighbour vertex (as shown in Figure 3.2(a)). We define the watershed of a pit as the union of the watersheds of the sinks inside that pit. If we then decide to eliminate a certain pit, we can also remove points from its watershed. Note that we do not always want to remove all points from the watershed, because a pit may lie next to a hill such that a lot of points on the hill have their water flowing to this sink while we do not want to remove the side of the hill when eliminating the sink next to it. Therefore, the approach Agarwal et al. [2006] take is to determine the height of the saddle point of the pit p to be removed and then *raise* all points in the watershed to $t(p)$ unless their elevation is already higher than that of the saddle point.

If we want to remove *peaks* in the same way, we can simply turn the terrain upside down and do the exact same thing again.

3.3.2 Topological persistence for big noise

This approach has been shown to work quite well for small noise, so we may consider extending the approach to make it work for removing big noise, too. We can however not simply eliminate pits with high height persistence because we would be removing all or a large part of the terrain (depending on whether we count the pit represented by the global sink). Removing only the sinks themselves and not the points in their watersheds would not work either. If a chunk of noise consists of multiple points that fall inside the same pit, we would only remove the most extreme one (the representative sink) because the others are only points inside the watershed of that one extreme point.

Apparently, we indeed need to remove points from the watersheds, but we have to be more careful in choosing which sinks to select. Looking at a typical example of noise such as shown in Figure 1.2, there is one thing that seems to be true for most of the chunks of noise we see: the peaks and pits they induce are very skinny. This suggests that we may want to eliminate sinks

that have small projected area. Fortunately, this can already be done within the extension of the persistence framework by Revsbæk [2007]: we can use "large projected area" as survival criterion and simply use the approach outlined above for small noise with projected area instead of height.

A potential problem with doing this is that we are still only removing the single outliers and not the clusters of errors, because they may make up a large pit with a big projected area (see also Section 5.2). Single outliers are arguably not the most challenging type of noise to remove. Secondly, we will also remove some of the lowest points of valleys and tops of hills, even while they are not noise, just because they have small projected area (see the bottom of pit (b) in Figure 3.4).

3.3.3 Saddle separation

It is clear that we will never be able to solve these problems when sticking to the small projected area idea — it simply does not always correspond to what we want to remove. Tracking back to the high height idea then, we will somehow have to limit the amount of terrain removed for high pits. Considering the climbing-free terrain model from Section 2.2, it might be a good idea to only remove those parts of pits that are not "nicely" connected to the terrain around the pit, such as the points in the bottom of pit (a) in Figure 3.4. We unfortunately do not have enough information to detect the kind of connectivity on the terrain we are looking for because the merge tree does not contain any information about the triangulation around the sinks and saddle points. Now we can do two things, either bend the framework enough to somehow be able to get this information, or loosen the connectivity idea. We will now first go with this second approach and then later see if we can fix it up.

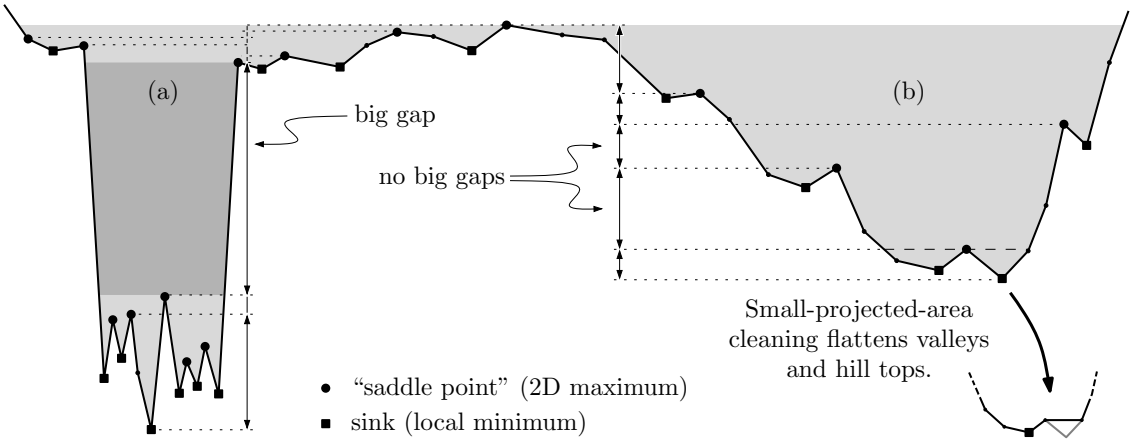


Figure 3.4: Example of saddle separation in 2D, and why cleaning based on small projected area removes too much.

The information we do have available in the merge tree about a pit is the set of sinks and saddle points it consists of and their heights. That is, we know which pits merged at which height, in which order. We can use this information to define, for a given pit, a height under which points have no climbing-free path to the outside world over sinks and saddles. The idea is to search for a large gap in the heights between two consecutive saddles or a saddle and a sink in the merge tree. If such a gap exists, we know there are no sinks or saddles in this height interval in the part of the pit containing the sink (such as the darker part of pit (a) in Figure 3.4), so we can select all sinks below this gap to be removed.

Definition 3.2 *A sink s in a topological merge tree T is* τ-saddle-separated *if s has an ancestor a (possibly equal to s itself) with parent $p(a)$ such that $h(a) + \tau \leq h(p(a))$. If this is the case, $h(p(a))$ is called the* raise elevation *of s. If there are multiple such ancestors, we choose the one closest to the root.*

This raise elevation (terminology re-used from the literature on removing small noise by flooding) can then be used to define which of the regular points and maxima have to go.

Definition 3.3 *A vertex v of a terrain h is* τ-saddle-separated *if and only if the sink s whose watershed contains v is τ-saddle-separated, and $h(v)$ is less than the raise elevation of s.*

Note that when we define noise in this way, perfectly smooth and nice terrains may be considered to be very noisy: because we only have a few sinks and saddles (the real minima of valleys and natural saddle points), the height distance between them may well be larger than the given threshold. If one would for example remove the sink points on the sides of pit (b) in Figure 3.4, one would be left with a nice, smooth valley with only sink and a big gap between that sink and the topmost saddle point. Therefore valleys in such terrains are wrongly classified as noise. In practice however, TINs always contain many local sinks and saddle points everywhere on the terrain, partially because this is also true in nature, and partially because of small noise, as we will see in Chapter 5. This implies that if we would want to place an algorithm implementing this type of big noise removal inside a longer processing chain, we would want to run it *before* any algorithm removing small noise, but this seems to be a natural order anyway.

Guarantees. If we want to prove that this definition of noise works well for climbing-free terrains, we would thus need to assume the existence of small noise. The small noise model given in Section 2.2.3 should be suitable for this purpose. The Delaunay triangulation has the property that for every edge there exists an empty circle through the end points of the edge [Berg et al. 2008], so all edges in a Delaunay triangulation of an ε-dense sample must be shorter than 2ε (when projected to the plane). If there would be an edge e longer than 2ε as in Figure 3.5, there should be a point p within ε distance of the centre c of the (supposedly empty) circle through the end points of e, contradicting the empty-circle property. Hence, we also know that for any edge (p, q) the original height difference (before measurement) $|h(p) - h(q)|$ on a terrain h that is slope bounded by σ, is at most $2\varepsilon\sigma$. By then assuming a minimum noise amplitude $\gamma > \varepsilon\sigma$, we can prove a lower bound $E_s > 0$ on the expected number of saddle points per unit of area for a sampling of a patch of slope-bounded terrain.

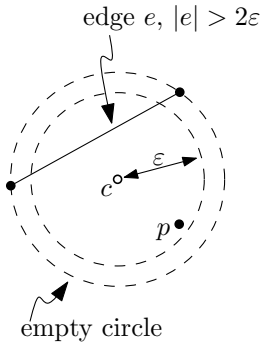


Figure 3.5: Maximum edge length.

Applying this to climbing-free corridors of given width and length, we can calculate the probability that there exist enough saddle points in a triangulation of a sampling of that corridor, such that there is at least one every τ units of height. This should then lead to a result stating that, for climbing-free terrains with wide-enough climbing-free corridors (relative to their length), no saddles and sinks of the triangulation are τ-saddle-separated, hence all points are correctly classified.

The second step would be to prove that points belonging to τ-separated noise will indeed be τ-saddle-separated, while adding this noise does not change the classification of other points in the terrain. The first part should be doable for reasonable values of τ with respect to the other parameters (γ, big noise should be larger than small noise, and therefore also larger than $\varepsilon\sigma$), because the terrain around the noisy spot R will have a significantly different elevation due to the "buffer zone" condition, with no points at intermediate heights to act as saddle points to prevent removal. The second part, guaranteeing that points on the original terrain are not influenced by removing big noise, is a bit more problematic. Because the "raise elevation" is taken to be the height of a *saddle point* that is close to the terrain, there may be some regular points higher than the saddle point that get recognized as noise, even though they are close to the actual terrain (see Figure 3.6).

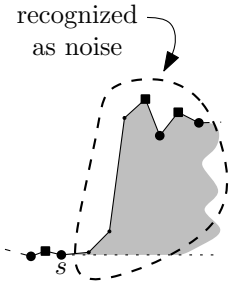


Figure 3.6: Area above selected saddle point s is shaded.

This problem exists mainly because we cannot make a good guess for the "correct" raise elevation when only looking at the points inside merge tree. We will therefore next consider an approach that completely circumvents the use of the merge trees and the persistence framework in general, and goes "back to basics".

Filtration. Before moving on to that topic, a last thing to talk about regarding the theory around this type of noise cleaning is its behaviour under varying thresholds: we would like the threshold τ to act as a paramater in a filtration of the point set. A filtration is a sequence of sets that is totally ordered by inclusion, where in our case $-\tau$ acts as an "index" into this sequence. This is very desirable in practice: it means that if we decrease τ, we always add more points and never remove any. Indeed, the set of τ-saddle-separated points forms a filtration parametrized by $-\tau$. The raise elevation of any sink only increases when τ decreases, hence more points will be τ-saddle-separated. Because we choose the ancestor a closest to the root (and hence highest) for which the height difference with its parent is at least τ, choosing a lower threshold always makes us pick either a or one of its ancestors. Hence, the raise elevation will be equal or higher.

3.4 Connected component cleaning

After having seen a definition of noise based (only) on topological persistence and its main artefact, the merge tree, let's see if we can make a shortcut and work directly on the triangulation to get a definition of noise that matches better with the climbing-free terrain idea. The idea is that if we have a cluster of points that we would classify as big noise, it is probably easy to separate their part of the triangulation from the rest of the triangulation because we know that they lie relatively far away from the other points, so the edges between these two parts of the triangulation are relatively long, especially in the vertical direction. The proposed definition of noise based on this idea is very easy: treat the triangulation of the terrain as a graph and remove all edges (p, q) from this graph of which $|h(p) - h(q)| > \tau$, for a given threshold τ, to obtain a graph G_τ. Then, the points belonging to the largest component (the one with the most points) in G_τ are considered part of the terrain, and the points in all other connected components are noise.

The idea of taking the largest component is given both by practice and by the definition of climbing-free terrains. Both suggest that G_τ will consist of one very large component spanning almost all points, a number of singleton components for the single outliers, and bigger components for the larger chunks of dense noise. Although some edges will be cut between points on the top of a cliff and below it, the climbing-free corridor condition should ensure that we do not consider points on sane terrains as noise.

To formally complete this definition of noise, we would have to specify which component is to be kept if there are multiple components with the same number of points. We could make some arbitrary decision here (for example taking all of them), but this seems unsatisfactory and really asks why we do not simply introduce an extra parameter specifying the minimum size of components to be kept. Apart from this introducing a mostly useless extra parameter to the definition that would need to be fine-tuned, it may also not be an intuitive parameter: it depends on the sample density and it is not clear why noisy components should always be smaller than non-noise components if the ("real") terrain indeed consists of multiple components. Note that taking component area instead of node number may be a little more intuitive, but is not well-defined on a "half-triangulated" graph like G_τ.

Filtration. Still, adding a parameter specifying the maximum component size gives the algorithm the nice property that varying τ yields a filtration of the point set. This is not the case when we always take the largest component, as shown in Figure 3.7. If we decrease τ gradually from ∞ to 0, and remove only components smaller than a given threshold, any particular point p will be considered as noise after one particular moment τ_p and never appear again, because connected components only get smaller when we decrease τ as we remove more and more edges. Hence, this variation of the definition does yield a filtration when decreasing τ. Note that increasing the other

parameter, the maximum component size, also yields a filtration simply because we exclude more and more components and hence points from the graph.

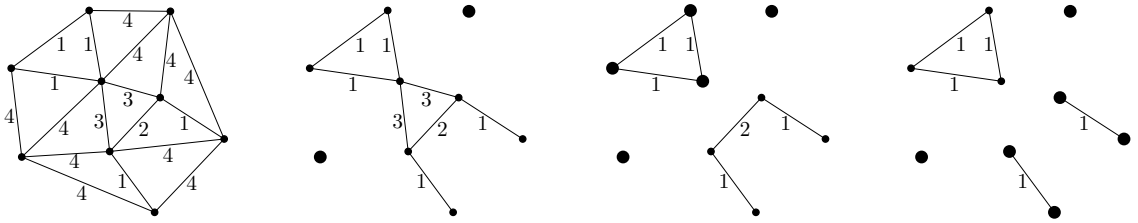


Figure 3.7: Always taking the largest component does not yield a filtration. The numbers on the edges indicate the difference in height between the endpoints, the bold points are the ones considered as noise for the given values of τ. Between $\tau = 2$ and $\tau = 1$, the three points in the top left appear again after having been considered as noise before.

Guarantees. As with the previous approach, let's see how far we can get in proving that this definition of noise classifies points from climbing-free terrains with separated noise correctly. In this definition, it is much easier to use the climbing-free corridors: as long as the width δ of the corridor is more than 4ε, the Delaunay triangulation always contains a path over the corridor (with bounded slope), connecting two points that are close to the ends of the corridor. If we add τ-separated noise (again for reasonable values of τ), we will not have the same problems at the boundary of the noisy spot R, but still have problems with points on the terrain that are entirely surrounded by outliers. Because all neighbours of such a point p have a significantly different elevation, all edges to p are removed from G_τ. Point p gets separated in its own connected component and thus recognized as noise. This will however only happen (on a large scale) for noise-inducing objects with a high opacity α, because otherwise there will be a high probability that a path exists in G_τ connecting p to the rest of the terrain.

3.5 Hybrids

Inspired by the good perspectives of the previous methods, it might be an interesting idea to combine them in order to overcome the disadvantages of both. In particular for the last problem we considered for connected component cleaning, points at terrain height surrounded by noisy outliers, it might help to have information about the geometry of the "pit" defined by the noisy points and conversely saddle separation can get help from the connectivity information available in the connected component cleaning framework.

The idea is to first find relevant pits that are candidate for removal using saddle separation, and then refine the removal process by looking at the triangulation of the points inside a pit. We could for example redefine the raise elevation as the height of the highest point that is still connected via a path of short edges to the terrain outside the pit. Using this different raise elevation, we can keep Definition 3.3 for selecting points to be removed.

3.6 Properties of CUBE

Finalizing our noise-technical deliberations, we will have a look at how the industry standard CUBE algorithm we saw in Chapter 1 behaves in the setting we have sketched. Because I did not have access to an implementation of the algorithm (it is implemented in a number of commercially available tools), this discussion is based on the description of the algorithm given by Calder &

Mayer [2003], assuming that the estimation nodes are placed on the vertices of a regular square grid.

For this discussion we will first have to identify which parameters the algorithm has that can be tuned by the operator. Apart from some statistical parameters such as the worst amount of error expected in the input, the following parameters seem to be relevant.

- Resolution of the estimation node grid.
- The maximum distance a measure point can be away from an estimation node for it to contribute to that node's estimate. This parameter is calculated per measure point, based on its error and some global, maximum allowable error. It is however always at least as large as the spacing between the nodes, so measure points always contribute to at least one estimation node.
- The length of the filter that delays spurious points in the data stream.
- The minimum distance of a sample's height to the current estimates at an estimation node that is required to start a new hypothesis. This parameter seems to be "hard coded" in the model and depends on the standard deviation of the estimate.
- The choice of hypothesis-selection method in case there are multiple hypotheses. In case of the use of estimation nodes in the neighbourhood, the size of the annulus needs to be chosen (Calder & Mayer suggest an inner radius of 5 meter and an outer radius of 10 meter).

Because the output of CUBE is a set of estimates with a surface, turning it into a noise removal algorithm requires another step, for example removing all points that are far away from the computed surface. This would involve another parameter specifying what is "far away". This also seems to be the parameter that comes closest in meaning to the thresholds in the algorithms proposed above.

The main problem with the algorithm appears to be condition 1: hypotheses without counter-evidence are always chosen. This implies that for any large enough noise-inducing object with high opacity, the estimation nodes placed well inside the region covered by the noise will generate only one hypothesis (or perhaps two, but the second only supported by so few samples that it will not be chosen), and hence the reconstructed surface follows the noise instead of the real terrain.

Chapter 4

I/O-efficient algorithms for noise removal

Now that we have seen a number of possible definitions of noise, we will have a look at some algorithms that efficiently identify points that are noise according to these definitions.

The classical approach to measuring how "efficient" an algorithm is, is to define the size n of a problem instance (such as the number of points in a given point set), and then prove an asymptotic bound on the number of CPU operations that need to be performed when running the algorithm for an instance of size n, when n goes to infinity. For big data sets however, in practice the running time of many programs stops adhering to the proved bound at some point well before the input size reaches infinity. What is happening is that the program is trying to use so much main memory that the operating system starts swapping out part of it to the hard disk. The hard disk is a lot slower than the main memory (with typical access times of 10 milliseconds) and the program is still trying to read and write data to various different places in memory, so the operating system has to fulfil all the program's memory-page access requirements, and the hard disk becomes the main bottleneck.

The underlying problem here is that the algorithm (and its implementation) is usually optimized to work in internal memory where it does not matter much if you read and write data to "random" places, whereas for a program to work efficiently with data on disk it needs to work as much as possible with data around the same location around the same time. The reason for this is that moving the head of the hard disk to a certain random place on disk takes very long, but when it is finally there reading a large number of consecutive bytes is very fast. There has been a lot of research on designing algorithms that make use of this fact, and since in the context of this thesis we deal with very large data sets it is a good idea to do so too.

4.1 I/O-efficient algorithms

In trying to come up with algorithms that use hard disks efficiently, a model has been developed to assess the efficiency of such algorithms. The model is based on the assumption that for any particular computer setup a number B can be given that specifies how many *consecutive* bytes one can approximately transfer in the time it takes to move the disk head. A chunk of data of this size is called a *block*, and B is called the *block size* or page size. The reason for specifying the block size in this way is that asymptotically it does not matter any more if multiple blocks are read consecutively or at random places on disk (the running time can only vary by a small constant factor). Furthermore, the model has a parameter M that specifies the size of the main memory in bytes. Algorithms in this model are allowed to make use of these two parameters and optimize their behaviour based on them, but can only use M bytes of main memory and transfer data to and from disk in blocks of size at most B. There is no limit on the amount of hard disk space an algorithm can use. The analysis of an algorithm then proceeds by counting the worst-case number

of disk accesses it needs to make when the size N of the input data and the block size B tend to infinity (for historical reasons N is often written with a capital letter as opposed to the classical model).

4.1.1 Basic results

Because we are counting the number of blocks transferred, and each block contains $\Theta(B)$ elements, scanning an input data set of N elements can be done in $O(N/B)$ I/Os (disk accesses). Sorting a set of N elements can also be done much faster than the trivial bound you would get from an optimal internal-memory algorithm that would use $O(N \log N)$ I/Os. By using a slightly changed version of merge sort, we can sort N elements using $O(N/B \log_{M/B} N/B)$ I/Os [Aggarwal & Vitter 1988]. The idea here is to use an M/B-way merging routine that keeps the current block from M/B data streams in internal memory such that the recursion tree gets as shallow as possible. Because this bound turns up very often in the analysis of I/O-efficient algorithms, it is abbreviated to $\textsc{sort}(N)$.

Another important technique used in the design of I/O-efficient algorithms is *time-forward processing* [Chiang et al. 1995]. It can be used to move information around in a directed acyclic graph to evaluate for all nodes some function that only uses information from a node's predecessors in the graph. When using an I/O-efficient priority queue, the method works by going through the graph in topological order (which is assumed to be given) and inserting an element in the priority queue for every edge, with a priority equal to the topological order number of the target vertex. When a particular vertex is to be processed, the information for that vertex is found and removed from the top of the priority queue. Using a priority queue with an amortized I/O complexity of $O(1/B \log_{M/B} N/B)$ per operation, we can processes a graph in $O(\textsc{sort}(N))$ I/Os [Arge 2003].

4.2 I/O-efficient topological persistence

The main problem with implementing persistence I/O-efficiently is finding which saddles merge which lakes. A natural way of solving this problem is to formulate it as a sequence of Union operations and Find queries on a disjoint-set data structure. In short, the lakes from the flooding process are represented by the sets and each time a new vertex is processed (in order of increasing height), a new set is created and merged with any lakes in the vertex's lower link. By inserting Find queries before merging the lakes below a saddle point, we can construct the compact merge tree T_C mentioned in Section 3.2.1 as follows. Because for every saddle point we know the answers to the $\textsc{Find}(u)$ and $\textsc{Find}(v)$ queries issued before merging the saddle point with the vertices u and v in its lower link, we can create an edge $(\textsc{Find}(u), \textsc{Find}(v))$ in T_C for every saddle.

The problem of I/O-efficiently answering the Find queries in an a-priori given sequence of Union operations and Find queries has been studied by Agarwal et al. [2006]. They give an optimal $O(\textsc{sort}(N))$-I/O algorithm for solving this problem on a sequence of N operations[1]. The algorithm is unfortunately too complicated to be practically useful, so Agarwal et al. also give an alternative, much simpler algorithm to get the same result in $O(\textsc{sort}(N) \log N/M)$ I/Os.

Revsbæk [2007] describes algorithms solving a more general version of this problem in which sets have properties associated to them that have to be merged appropriately in every Union operation using a given function. The bounds he gets are the same as above. Revsbæk also defines such properties and merge functions corresponding to the projected area, surface area and volume of lakes.

4.2.1 Partial flooding

After the compact merge tree with associated sink persistence values and saddle heights has been constructed, we can simulate partial flooding with threshold τ by finding new heights for all vertices in the terrain in two steps [Danner 2006, Section 3.3.2].

[1] In case all Union operations merge two different sets, which is true in our application.

The first step is to find the raise elevation for all sinks: for sinks with persistence at least τ this is the height of the sink, and for sinks with persistence lower than τ this is the height of the saddle point corresponding to the most significant sink in the subtree that is removed. This means that we need the height of such a saddle point in all descendant sinks. This information can be sent to the right locations using time-forward processing: if we direct the edges of the compact merge tree away from the root we get a directed acyclic graph in which we want to compute a function (that computes the raise elevation) for every node that only depends on information from its predecessors (namely *their* raise elevation). A topological order for this graph is given by the heights of the saddle points paired with the sinks. Because with every edge we store the persistence of the highest of the two sinks and the height of the saddle point, we can thus compute the raise elevation for every sink as follows. Set the raise elevation of the root (the global sink) to $-\infty$, and the raise elevation of any other sink s equal to that of their parent $p(s)$, except when the persistence of s is lower than τ and the raise elevation of $p(s)$ is $-\infty$, in which case we set the raise elevation of s to the height of the saddle point paired with s (see Figure 4.1).

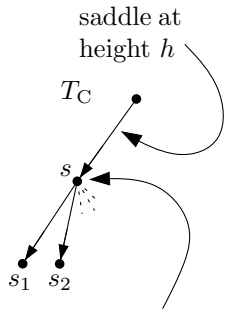


Figure 4.1: Time-forward processing for raise elevations.

The second step consists of forwarding the raise elevation of the sinks to the terrain vertices in their respective watersheds. By adding FIND queries after all UNION operations in the union–find sequence for regular vertices and maxima, we can find their watersheds by scanning through the output of the batched union–find algorithm. A simple sorting and scanning step then suffices to get the raise elevations to the vertices of the terrain. We can now determine locally for each vertex if it is lower than the raise elevation and has to be (re)moved, or it is at least as high as the raise elevation and can stay.

4.3 Saddle separation

We would like to determine for each vertex of the terrain if it is τ-saddle-separated according to Definition 3.3. If we can first find a raise elevation for all sinks, we can reuse the second step above and (re)move the vertices in their watersheds that are too low. Finding whether sinks are τ-saddle-separated and then finding the correct raise elevation requires a different approach.

For removing sinks with low persistence, the definition of noise lined up nicely with the compact merge tree constructed from the union–find output: we basically selected subtrees from the compact merge tree. Saddle separation however we defined by selecting subtrees from the *topological* merge tree. In order to use a similar I/O-efficient approach we either have to rewrite the definition or transform the compact merge tree. We will take the first approach because it turns out that in this way we can actually keep much of the structure of the solution for low persistence.

Definition 3.2 states that for a sink s to be τ-saddle-separated, there should be an edge on the path from s to the root for which its two endpoints have height difference at least τ. On this path we will at some point encounter a saddle point a at which s dies (except when s is the global sink, in which case a is the root of the tree). If there is such a long edge, we consider two cases: either the chain is broken between the root and a, or it is broken between a and s (in case it is broken in both places, we consider the one closest to the root). If it is broken between the root and a, all sinks in the subtree rooted at a in the compact merge tree have to be selected and the raise elevation is dictated by a sink higher up in the tree, as was the case for selecting sinks with small persistence. When the chain is broken at the ith link in the path from s to a however (counting the edge to s as $i = 0$), we want to select s together with all sinks in the subtrees rooted below the ith link. In the compact merge tree this corresponds to selecting the i child sinks with the lowest saddle points, and only s itself if $i = 0$ (see Figure 4.2).

This observation helps us to implement the sink selection step I/O-efficiently: we can again use time-forward processing over the edges of the compact merge tree, but we now sort the edges

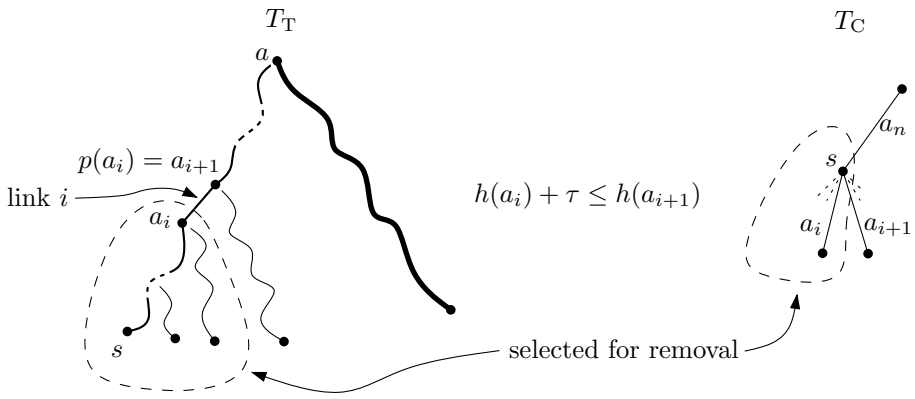


Figure 4.2: Selection of sinks to remove in the topological and compact merge trees.

lexicographically by increasing height of the source vertex (parent in the tree) and decreasing height of the saddle represented by the edge. We can then scan over each sink s and find in the priority queue if any of its ancestors summoned it to be selected and then send this on to its children, or otherwise run over all its children and see if there is a τ-sized gap between, consecutively, the height of s's saddle point, the heights of the saddle points of s's children and the height of s itself. If at any point during this scan such a gap is found, any child sinks that are still to be processed can immediately be summoned to be selected by inserting a corresponding item in the priority queue. If a gap was found in this way, a itself is selected, otherwise it is not.

In this way we are still only scanning over the edges and pushing an item on the priority exactly once for each edge, so the algorithm runs in the same number of I/Os, $O(\textsc{sort}(N))$ for a terrain of N vertices.

4.4 Connected component cleaning

The definition of connected component cleaning was pretty simple and describing an implementation is fortunately equally simple. It consists of the following steps.

1. Extract all edges from the given TIN and write to an output stream only those edges that span less than τ in the vertical direction.
2. Label the connected components in the graph represented by this stream.
3. Find the largest component.
4. Write all vertices from the largest component to an output stream.

Instead of taking the largest component, we can of course take all components larger than a given threshold k. The last two steps should then be replaced by the following steps.

3′. Count the size of all components.

4′. Write all vertices to an output stream that are part of a component with more than k vertices.

All steps except step 2 can trivially be implemented using a few scan and sort steps. For step 2 we can run an algorithm to find the connected components of a planar graph using $O(\textsc{sort}(N))$ I/Os [Chiang et al. 1995].

4.4.1 Optimizations

Some of these steps can be optimized to run in $O(N/B)$ time, but there are places where it is hard to get rid of the sorting step. We will now investigate the procedure step-by-step to see what can be optimized.

In step 1, we need to extract the edges from the TIN. Depending on the representation this may be done in one scan, but in order to be able to run step 2 in $O(N/B)$ I/Os, we need the edges to be in a particular order: sorted lexicographically by the horizontal components of the coordinates of the end points of the edges (first by the x and y coordinate of the lexicographically smallest end point, and then the x and y coordinates of the other end point). Unless the input is in this order, we need to sort the edges at this point.

For step 2, there exists an algorithm that runs in $O(N/B)$ time, but apart from the assumption on the order of the edges we also need to assume that the intersections of a horizontal or vertical sweep line and the (embedded) graph always fit in main memory. An algorithm that achieves this running time is based on a method to find connected components in flat parts of grid terrains [Danner 2006, Section 3.4.1] and is not yet published [Arge et al. 2009]. The algorithm performs two line sweeps over the set of edges and therefore needs the edges to be sorted in the order the sweep line would visit them. The output of this algorithm is the same set of edges, but now labelled with a number representing the connected component they are part of.

The next step, number 3, is concerned with finding the largest component, or any component with more than k vertices. For this, we will first need to extract the *unique* vertices from the list of edges. We can do this by writing for each edge both end points to an output stream, sorting this stream and removing the duplicate vertices.

If we can then have a counter in memory for each connected component, a single scan suffices for steps 3 and $3'$, otherwise we can use an $O(\text{SORT}(N))$ algorithm (for example by sorting the list of component labels and counting the duplicates).

The final step consists of dropping all vertices that belong to a connected component that is found to be too small. If we can store an identifier for each large (or small) connected component in memory, this can again be done by only scanning over the data, otherwise we again need to resort to sorting.

Preprocessing. The only unavoidable sorting step left is the one to make the set of vertices unique. In case we want to run a number of connected component cleanings with different values for τ and/or k, we can choose to do some preprocessing taking $O(\text{SORT}(N))$ I/Os such that the remainder of the algorithm runs in $O(N/B)$ time. This preprocessing consists of finding for every vertex v the incident edge $e_s(v)$ that is shortest in height (with minimum height difference between the end points, and some tie-breaking rule), and marking for every edge $e = (u, v)$ whether $e_s(u) = e$ and/or $e_s(v) = e$. We then order the edges including these two extra bits of information in the order necessary for the connect component labelling algorithm. This preprocessing can clearly be done in a few sort and scan steps and allows for quick cleaning with specific thresholds as follows.

Because the edges are already in the correct order, we only have to drop the ones that are too high and run the connect component labelling algorithm. We then output for each edge $e = (u, v)$ end point u if and only if $e_s(u) = e$ (which is stored locally with the edge), and end point v if and only if $e_s(v) = e$. This guarantees that for every non-singleton component all vertices are output once, so we do not need to remove duplicate vertices.

4.5 Hybrids

The hybrid approach may be implemented I/O-efficiently using a combination of the tools described above. We follow the saddle separation approach up to finding the raise elevations. Instead of forwarding the elevation, we only flag the pits(' representatives) as candidates for consideration and write them to a separate stream. Then, we aggregate the vertices of the terrain according to

the flagged pit they belong to, and ignore (thus keep) them if they do not belong to any such pit. This can be implemented using a few sort and scan steps because we know which points belong to which watershed. The next step is to run the connected component labelling algorithm on the subtriangulations corresponding to each pit's vertices, with extra edges marking the border of the pit that is connected to the rest of the terrain. This results in a labelling where we know the identity of the border component. Taking the maximum height of any vertex in this component yields the new raise elevation and can be applied to all vertices in the pit.

Chapter 5

Results

In this chapter, I will first shortly discuss an implementation of the algorithms described in the last chapter and a tool to visualize the results, and then give examples of typical results of the algorithms applied to a number of real world data sets (mostly from multibeam echosounders, but also from LIDAR scans). We have a look at the differences between the various automatic cleaning methods and the manually cleaned data sets. I conclude with a discussion of running times in practice.

5.1 Implementation

The algorithms were implemented within the TerraSTREAM framework [Danner et al. 2007], based on the TPIE (Templated Portable I/O Environment) library [Arge et al. 2002] for implementing I/O-efficient algorithms. TPIE provides the low-level primitives for handling disk blocks and streams, as well as some basic data structures such as a priority queue. TerraSTREAM includes functionality to handle terrain data such as readers and writers for point data and I/O-efficient two-dimensional Delaunay triangulation, as well as libraries and front-ends to for example generate contour maps and do flood and water flow simulations. The contour map generation code includes an implementation of the connected component labelling algorithm mentioned in Section 4.4. TerraSTREAM also includes code to compute the topological persistence (in any of the height, projected area, surface area or volume measures) of all sinks of a terrain, construct a corresponding compact merge tree, and subsequently eliminate sinks with small persistence by partial flooding as described in Section 4.2.1. The saddle separation method was implemented as a step in this process, replacing the partial flooding step.

The implementation of the connected component cleaning algorithm implements the "non-primed" version of the algorithm from Section 4.4 using the connected component labelling algorithm already present in TerraSTREAM. Because the input data is given as two files, one containing all vertices with their coordinates and another containing all triangles with the identifiers of their vertices in the first file, we first need to write out edges containing only the identifiers of their end points and then sort this stream twice to store the coordinates with the edges, and then sort them again for the connected component labelling algorithm.

The hybrid approach put forward in Section 3.5 and 4.5 has not yet been implemented, but it seems interesting enough to be worth trying in the future, also considering the results shown later in this chapter of the other two algorithms.

5.1.1 Visualization

There are many software tools available to visualize point sets or terrain meshes, but I am not aware of any that are both capable of visualizing very large data sets, and give an exact rendering of a terrain as given by a Delaunay triangulation.

More specifically, one visualization tool we used is called *NaviModel* from Eiva which is developed to visualize large point sets as terrains quickly. It does a good job at that, but perhaps a bit too good in the sense that it only shows an approximation of the terrain most of the time, while viewing the raw data — from which we want to identify the noisy points — is hard. The approximated terrain is drawn using a triangulation that does not correspond to the triangulation used in our noise removal software. This makes it hard to see which points we want to remove and on which grounds we would want to do that.

Other tools are mostly geared towards viewing smaller data sets that fit in main memory and allow drawing arbitrary shapes (including the triangles from a given triangulation). Although we are not looking for a tool to do the currently infeasible thing of showing *all* triangles of the Delaunay triangulation of a large terrain, we would still like to be able to view parts of it and also see the difference between the raw input data, manually cleaned data and algorithmically cleaned data so as to judge the classificational power of the proposed algorithms.

Second, we would like to be able visualize the terrain as seen from the persistence framework: where are the sinks and how are they organized within the (compact) merge tree?

My visualization tool solves both problems by first presenting the user with an overview of the terrain by plotting the sinks of the terrain in 2D in any of a number of different user-selectable ways, and then allows the user to get an interactive 3D visualization of a part of the terrain that is small enough for the video card of the computer to render.

Overview plot. The 2D overview plot shows all sink points from a compact merge tree as produced by software earlier in the pipeline. The data includes both the horizontal and vertical position of the sink points (x, y and z coordinates), and their persistence according to one or more measures. The user can choose which of these values should be positioned on the x and y axes of the plot and whether they should be plotted using a logarithmic scale. Zooming functions are available to get more detail in a selected area. Example plots are shown in Figure 5.1 and 5.2.

The overview plot can also be used to display all information available for a specific sink, and draw the subtree of the compact merge tree that is rooted at that sink. In combination with the detailed terrain view described next, this provides valuable information for determining how good the merge tree is at describing the intricacies of a particular terrain.

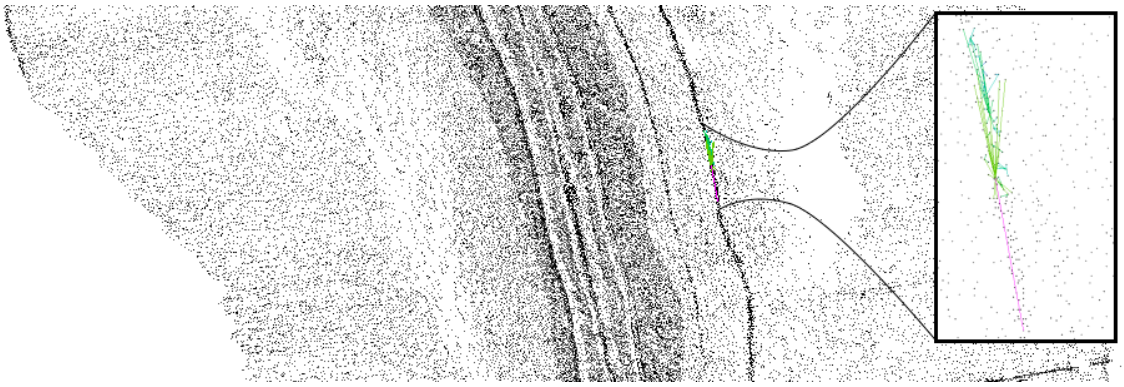


Figure 5.1: Typical view of sinks by x and y coordinates: they appear everywhere on the terrain with a density depending on the actual data point density. The larger white spots are indeed empty regions in the input data. The overlay shows a subtree of the compact merge tree; the purple line connects the selected sink to its parent, and the green to blue lines connect the selected sink recursively to its descendants. Data from StatoilHydro, patch shown is 38×11 meter.

Detailed terrain view. The detailed terrain view gives in an interactive visualization of a data set by drawing a small sphere for every input point, a triangle for every triangle of a given triangulation, and a line for each edge of this triangulation (although any of these can be disabled

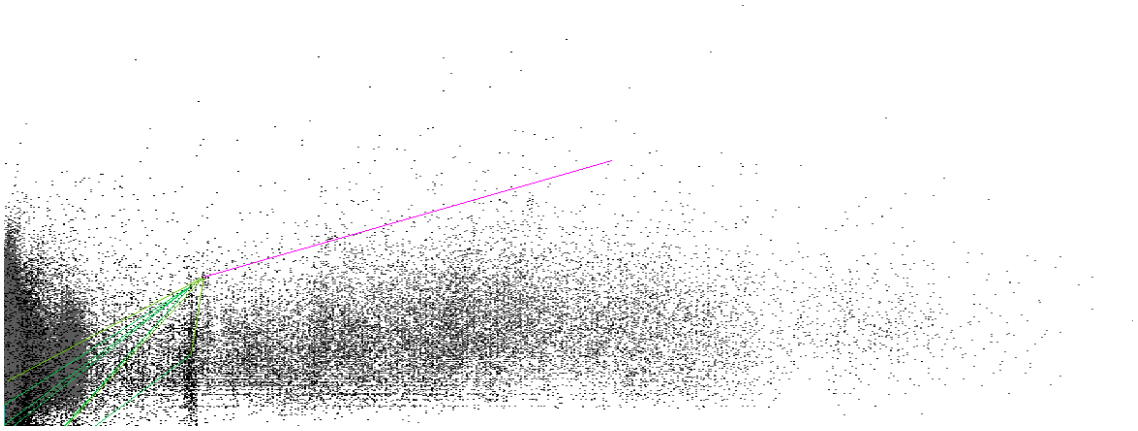


Figure 5.2: Sinks by height (on the x-axis from 0 to 6.8 meter) and projected area (on the y-axis in logarithmic scale from 0 to 146 square meter). Most sinks appear in the bottom-left corner of the graph with height less than 0.5 meter and projected area less than 0.1 meter. Other sinks (especially those with high height) are often located at the top of a noise cluster.
Data from StatoilHydro, same data set as used for Figure 5.1.

to get a less cluttered view), in scaled world coordinates. The subtree of the compact merge tree rooted at a sink point can also be visualized in this view using lines between sinks to represent the edges of the tree, but this turned out not to be much more useful than getting the same information in the overview plot. In order to get reasonable response times for opening this view on the input data within a given horizontal window (all data points and triangles for which their projection on the 2D plane falls completely within a given rectangle), we use a two-dimensional Hilbert R-tree [Kamel & Faloutsos 1993] constructed over the set of points and triangles (with a triangle being represented by a point in its interior, because we are only interested in triangles that are contained in the query window). The R-tree allows us to answer this query efficiently without loading the full input data set into memory.

The most important extra feature of this view is the ability to colour points according to their occurrence in different data sets. The user can provide the tool with up to three data sets: the raw input data including all points, a subset of these points that are noise according to a manual cleaning, and an algorithmically cleaned version of the terrain. The tool then draws the points of the original data set in four different colours (see the figures on pages 39 and 40).

- Green points are classified as noise by both the manual and the automatic noise removal processes.
  Intuition: Equal classification; green is a general colour for good things.
- Blue points are considered part of the terrain by both processes.
  Intuition: Equal classification; blue is also the colour of the background triangulation, so "Don't worry".
- Red points are removed by the algorithmic cleaning method, but not in the manual cleaning.
  Intuition: Different classification; red colour warns about trouble.
- Orange points are removed by the manual cleaning, but not in the algorithmic cleaning.
  Intuition: Different classification; nearly as troublesome as red, but perhaps less problematic.

Furthermore, the user can switch between the triangulations of the input data set and the algorithmically cleaned one. This view makes it instantly clear how well a particular noise removal algorithm is doing, and was helpful both in identifying problems with the implementation as well as the underlying theory and ideas.

All pictures of real terrains in this thesis were generated using this tool.

5.2 Cleaning with small projected area persistence

The algorithm described in Section 3.3.2 moves points inside pits with a low projected area up (or down, when looking at the other side of the terrain) to the height of the saddle point that is paired with the sink representing that pit. The algorithm gives quite nice results on first sight for data sets with small groups of outliers. Figure 5.3 presents a data set that is cleaned in this way and shows all relevant properties of this type of cleaning in practice. For example, spikes with one or two data points are cleaned well (compare the right-most part of (a) and (b)).

Still, there appears to be quite some noise left in Figure 5.3(b). For the most part, this is due to the flooding; all points in a pit are moved to the same height such that long, narrow pits are only cut short but not completely removed. In order to get a better comparison with the other algorithms, Figure 5.3(c) shows the results of the same algorithm, but now *removing* points inside selected pits instead of moving them. The result is much better, but there is still some noise left that apparently has a too high projected area. Increasing the threshold is however not an option, as it already becomes clear that tops of hills (even the very low one in the picture) get cut off. When increasing the threshold, this effect will get worse and worse, making the result unacceptable. For comparison, the result of cleaning this data set with the connected component cleaning algorithm is shown in Figure 5.3(d).

5.3 Saddle separation

Figure 5.4 shows a patch of terrain like those in Figure 1.1 and 1.2. The saddle-separation algorithm works well for these types of noise and the original terrain is clearly left intact.

A more intricate example is shown in Figures 5.8 and 5.9. These pictures display the same data set, including a pipeline, from different angles. The colouring of the data points is as explained in Section 5.1.1, and the surface in Figure 5.8 is drawn only through the points that are kept by the saddle separation algorithm.

These figures show a few interesting patterns. A lot of noise appears not only above and around the pipe, but also underneath it. The algorithm classifies most of these points correctly as long as they are relatively far away from the other points. At various places we can however also spot red points indicating that the algorithm removed points that should probably have been kept. This happens especially at two places.

First, there are red points that appear to be part of the pipe (see the overlay in Figure 5.8). These are removed because they are surrounded (in the 2D triangulation) by other points that are on the ground. Most likely, neither of these points are in gross error, but the effect is caused by the fact that the pipe is round and some soundings hit the ground under the pipe. The algorithm then removes the seeming outliers because they form a high peak of their own.

Second, there are scattered red points on the flat surface around the pipe that are caused by the existence of outliers close to them in horizontal position. The artefact noted in Figure 3.6 appears to show up in practice.

5.4 Connected component cleaning

Figure 5.5 shows the results of two runs of the connected component cleaning algorithm on a LIDAR data set from an urban area. The data was already mostly cleaned and the only points left are the ones that are classified as points on the bare earth by another algorithm. Because LIDAR cleaning algorithms assume that the lowest points in an area are the true ground, pits found at for example construction sites are kept. Here, our algorithms can help to remove also those last remaining features, as long as they are "well separated" from the terrain. This is not the case for ramps, where you might recognize some climbing-free corridors in the example data. Anyway, it is not clear whether consumers of terrains are interested in such features or not.

Sometimes smaller components in a pruned triangulation exist that should not be removed. An example of such a situation can be seen in the centre of Figure 5.7.

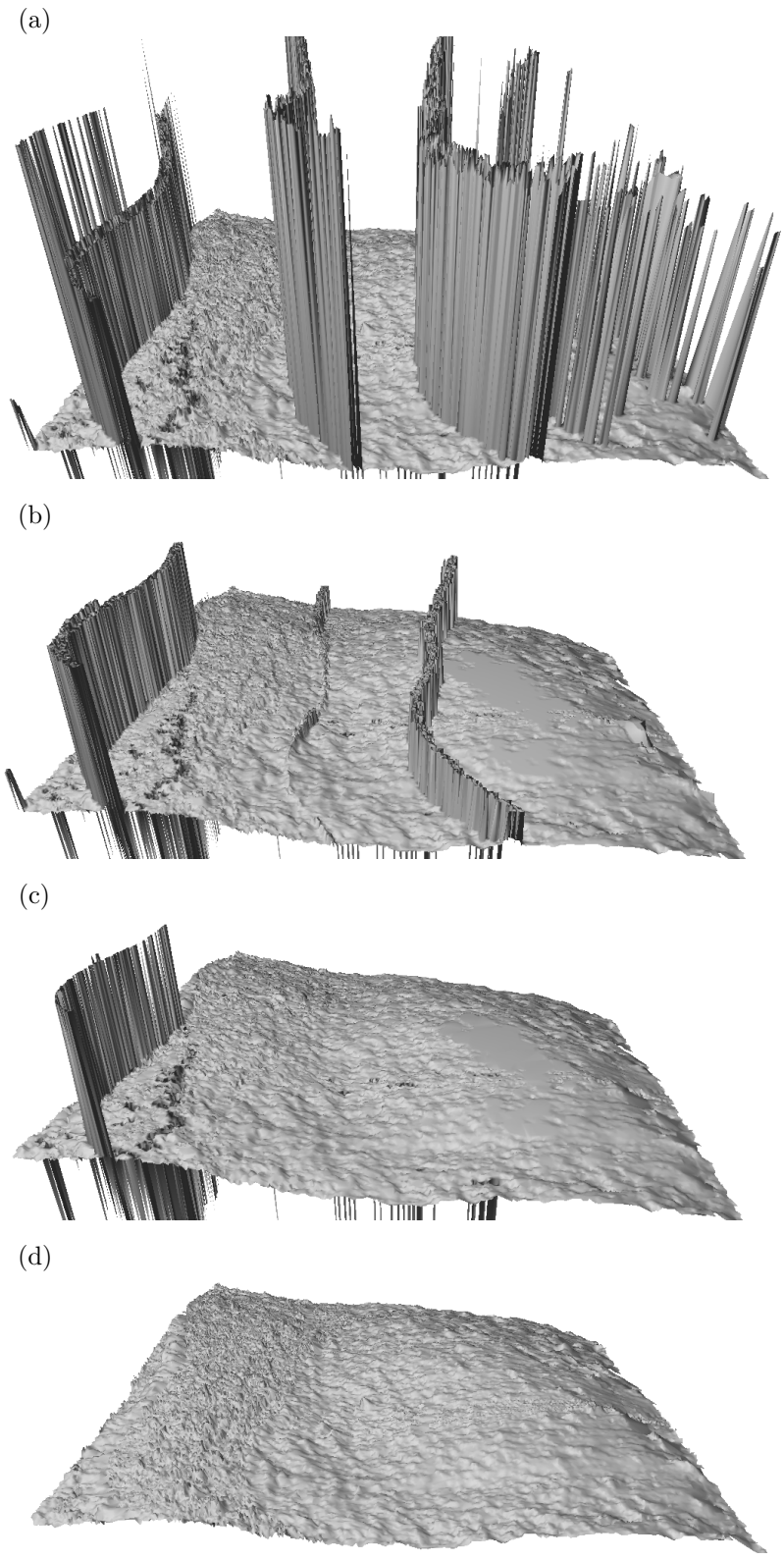
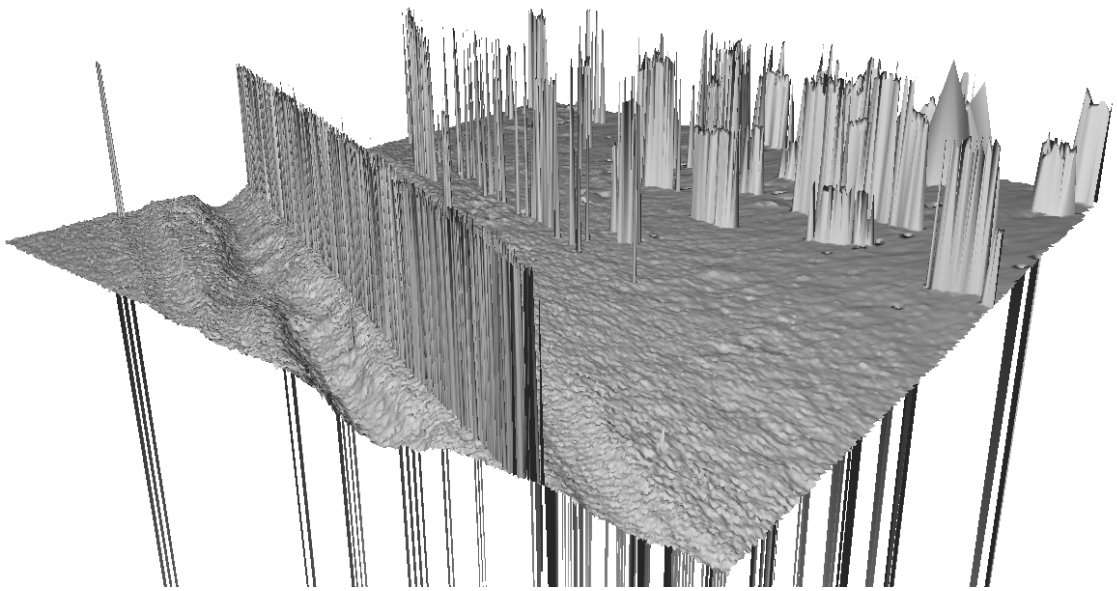


Figure 5.3: Results of cleaning sinks with small projected area persistence.
(a) Input data. (b) Flooding sinks with projected area smaller than 10 m^2. (c) Same as (b), but removing points instead of flooding. (d) Connected component cleaning with $\tau = 10$ cm.
Data from StatoilHydro.

(a)



(b)

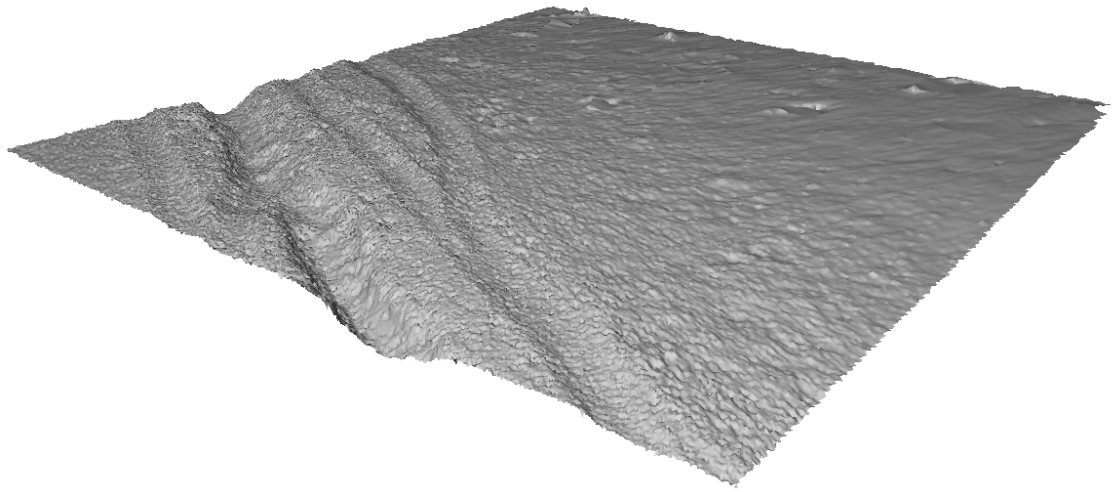
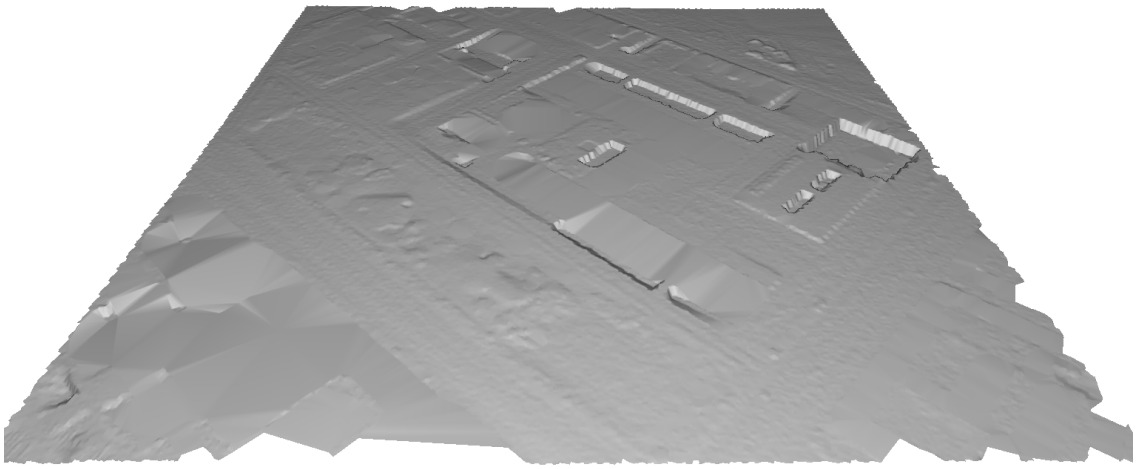
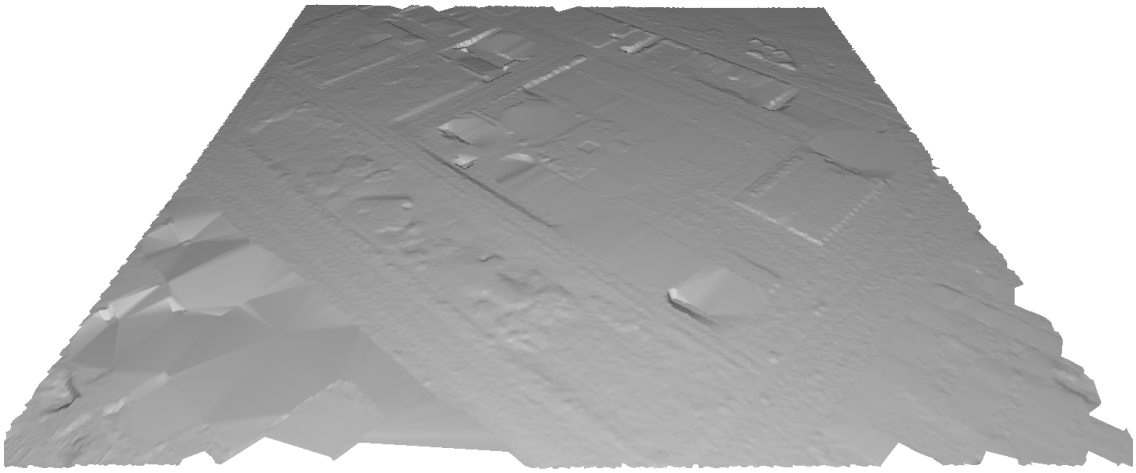


Figure 5.4: (a) Uncleaned data. (b) Data cleaned with the connected component cleaning algorithm with a threshold $\tau = 20$ cm.
Data from StatoilHydro, patch shown is 14×14 meter.

(a)



(b)



(c)

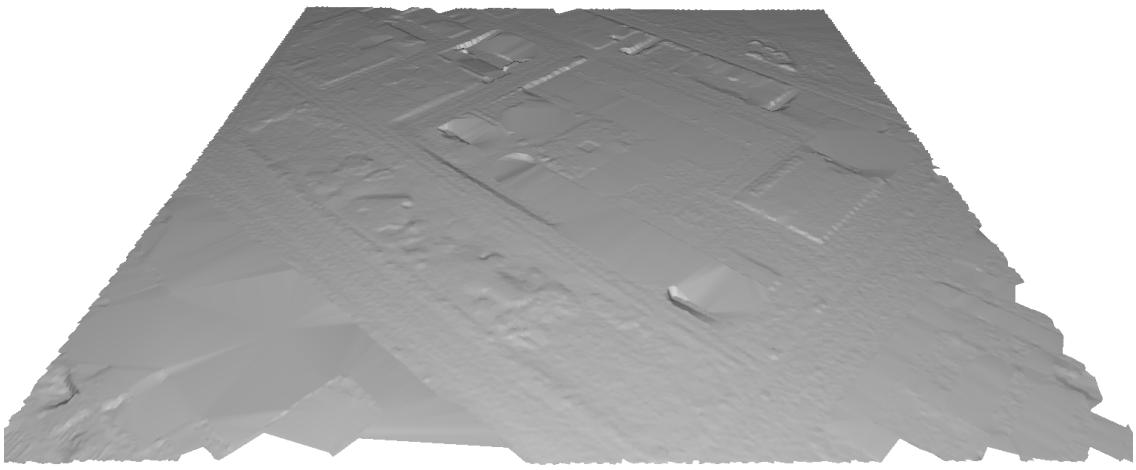


Figure 5.5: Some pits in the ground are not removed by LIDAR point cleaning. Connected component cleaning can remove them. Shown is the original data (a), and cleaned versions for thresholds $\tau = 3$ m (b) and $\tau = 2$ m (c). Data from COWI, patch shown is 500×500 meter.

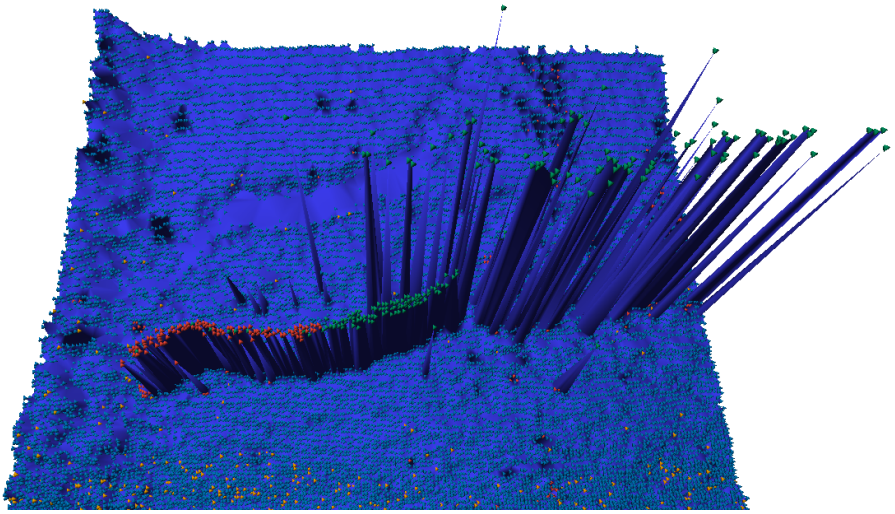


Figure 5.6: The red points on the left were (likely) misclassified during manual cleaning. Note that the green points in the centre seem to be part of the same feature but are classified correctly. Data from Eiva, patch of 5×5 meter.

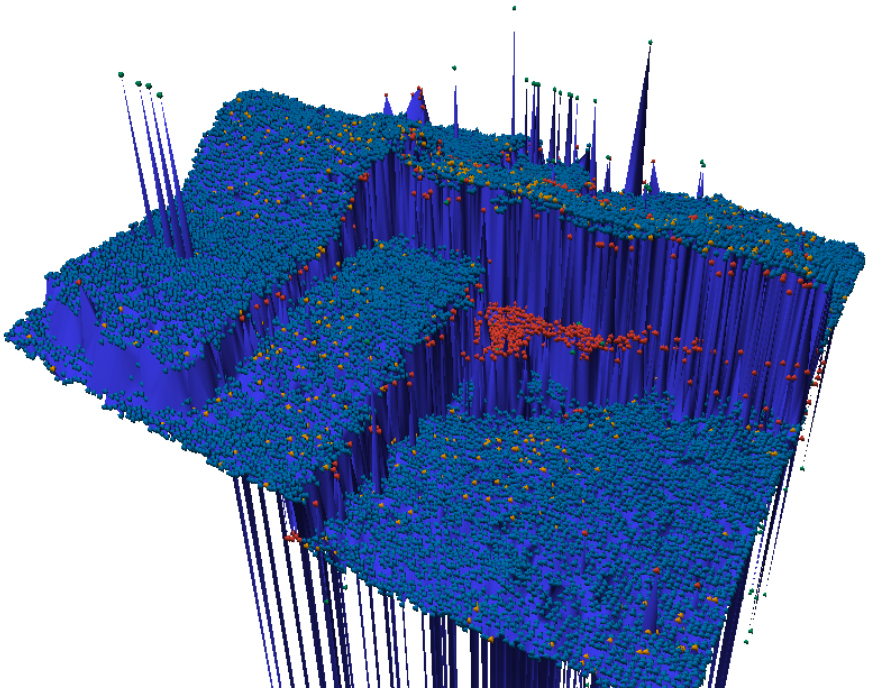


Figure 5.7: The red points in the centre are misclassified by the connected component cleaning algorithm because they form a connected component that is separated from the rest of the terrain. Data from Eiva, patch of 4×4 meter, $\tau = 10$ cm.

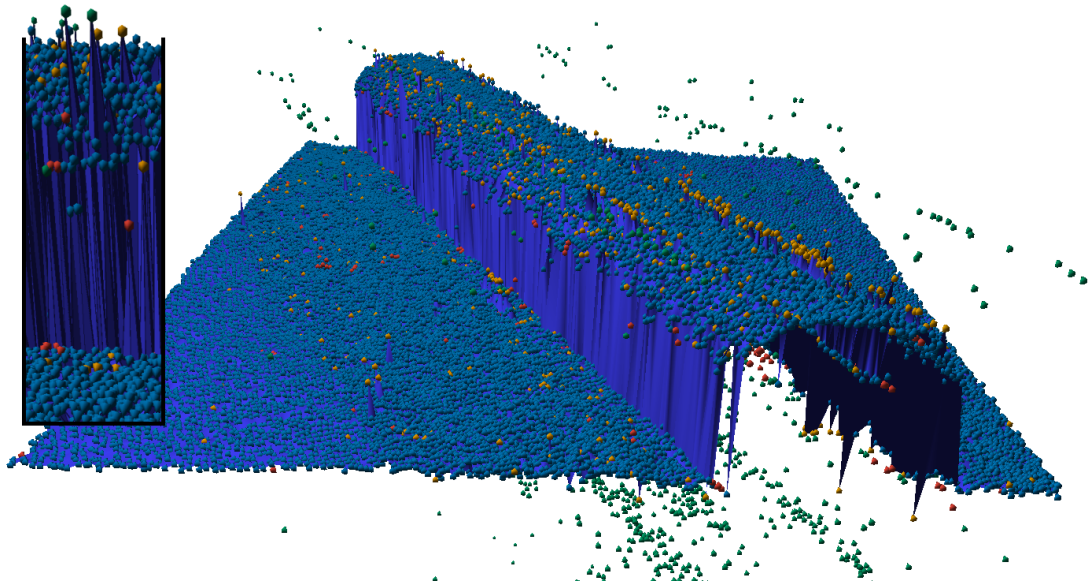


Figure 5.8: Example of point classification around a 90 cm pipe using saddle separation cleaning. Surface shown is a retriangulation of the cleaned point set. Overlay shows a close-up of the triangulation of the original point set.
Data from Eiva, patch of 5×5 meter, $\tau = 30$ cm.

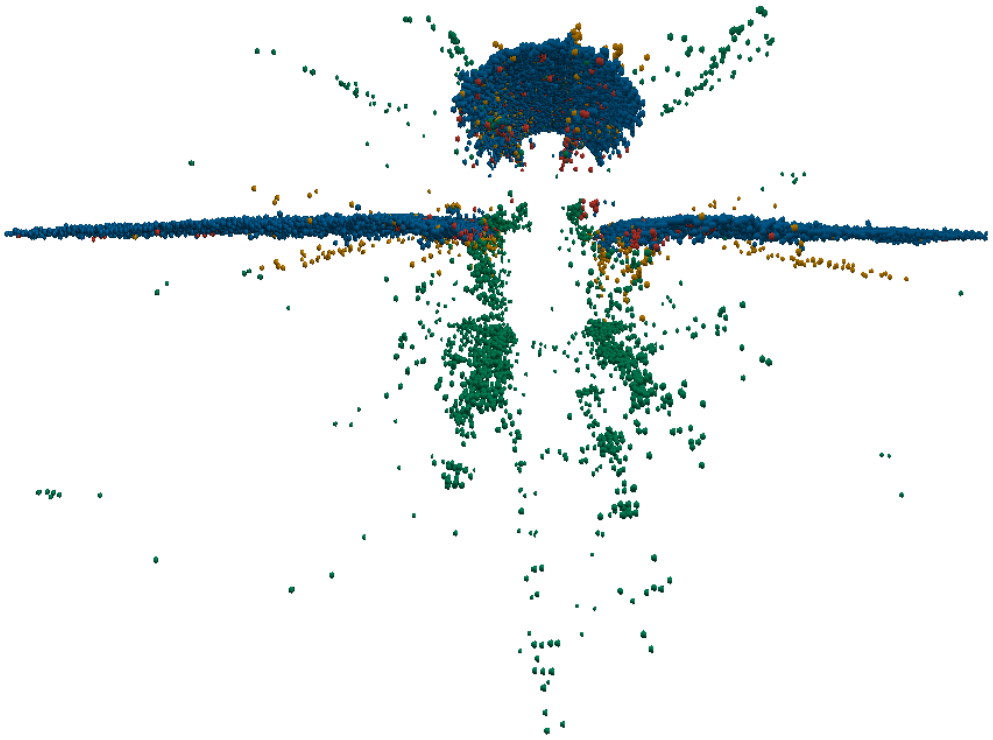


Figure 5.9: Same as Figure 5.8, but only showing the points, looking in the direction of the length of the pipe.

5.5 Performance

Although the implementations are not much optimized for performance (in particular connected component cleaning), Table 5.1 lists running times of (parts of) the algorithms named above for a typical data set of about 4.7 million points (and about 9.4 million triangles in the generated triangulation). The programs ran on a 1.7 GHz Pentium M with 700 megabyte of main memory available for the application, using a 7200-rpm hard disk.

Step	**Time**
Triangulation	2 min.
Merge tree generation	35 min.
Cleaning: small projected area	2 min.
Cleaning: saddle separation	2 min.
Cleaning: connected components	42 min.
Total (from point set to cleaned point set)	**Time**
Small projected area / Saddle separation; with point removal	78 min.
Small projected area / Saddle separation; with flooding	76 min.
Connected component cleaning	44 min.

Table 5.1: Running times for cleaning steps, rounded to nearest integer minutes.

Because all algorithms need a triangulation to work with, this step is listed separately and only added to the totals in the bottom part of the table. This step is optimized well, and a significant part of the time is spent on reading and writing the input and output, which is stored in an inefficient human-readable format.

Both the cleaning algorithm based on small projected area and saddle separation cleaning make use of the compact merge tree of the terrain (the required attributes are different, but their calculation does not contribute much to the overall running time). Therefore, the merge tree generation step is also listed separately. The two cleaning steps corresponding to those algorithms do most of their processing on the merge tree which is much smaller than the full data set (less than half a million sinks in this data set), so the total time spent is again largely dependent on the time for reading and writing the input and output files.

For connected component cleaning, a large amount of time is spent on bringing the data in a suitable format: the first 15 minutes are used to extract 28 million edges from the triangulation (which is given as a list of triangles with indexes to the points that are stored in a second file), to sort those edges and add the coordinates of the first end points, and sort the edges again to add the coordinates of the second end points.

The total running times are based on cleaning both sides of the terrain, and are composed as follows. For small projected area and saddle separation we first need to triangulate the given point set, generate the merge tree for the pits of the terrain, and clean the pits. Then, for the second run we can choose to either remove the selected vertices, or use the "partial flooding" approach in which we only move the vertices of the pit (we did not consider this for saddle separation but is perfectly possible). In the first case we need to retriangulate the point set because the earlier triangulation is invalidated by the removal of points. After this, we can again generate a merge tree — for the peaks of the terrain this time — and do another cleaning step. For connected component cleaning, we only need to add the triangulation time because the method automatically cleans noise on both sides of the terrain.

Although these experiments already tested the I/O-efficiency of the algorithms, it would be good to know if the implementations also scale to much larger data sets. For connected component cleaning such a test was done: a data set consisting of 200 million points (with a triangulation of about 25 gigabyte in the inefficient format mentioned before) was cleaned in less than 18 hours (the exact timing is unknown, though the machine was faster than the one used for the experiments above).

Chapter 6

Conclusion

We have seen that noise in terrain data can have many different causes and appearances — small noise from sensor inaccuracy, big noise both from birds and fish, and from gross mismeasurements. Previous work has either focused only on small noise, or made an attempt at removing big noise too, but fails for large enough clusters of outliers. Furthermore, little work has been done in this area on methods that also work efficiently when the data sets are too large to be worked on in main memory, again only small noise has been considered.

The reason that big noise has not had much interest from the theoretical side is not difficult to guess — if we are analysing worst-case scenarios, how can we give any guarantees on good surface approximations? From the more pragmatic side of the spectrum, assumptions (implicit or explicit) are made about the maximum size of the region affected by the noise.

This thesis presents a first model to capture the essence of big noise mathematically, and thereby poses the question in how far algorithms can be made to recognize such noise. Because we do not expect algorithms to do the impossible — such as reconstructing an unknown surface — there is some hope for the existence of such algorithms. The model may still be a rough first attempt, but it is already very helpful in assessing the qualities of new algorithms and leads the way for theoretical approaches to the problem.

I also proposed and implemented two new algorithms to recognize and remove big noise. On the theoretical side, they are shown to come some way in recognizing noise according to the given models, and proved to be efficient in the external-memory model (that is, run in the same asymptotic time bound as sorting). On the practical side, the algorithms have shown to be able to recognize much of the noise found in real-world data sets from MBES scanners, and also help cleaning features from LIDAR data that are left behind by other cleaning algorithms. The implementations of these algorithms are based mostly on known methods from the I/O-efficient algorithms literature and are therefore expected to scale well. The implementation was not optimized for performance, but has already shown to be able to work its way through data sets much larger than the available main memory.

6.1 Future work

For the MBES and LIDAR data sets considered in this thesis, the proposed algorithms appear to do a good job at removing gross errors, both when present as single outliers and occurring in larger clusters. Also noise removal around pipelines lying on the seafloor works reasonably well (the algorithms do not seem to make many more mistakes for gross outliers than manual cleaning). The most important feature of the MBES data sets that has not received much attention in this thesis is pipelines with *free spans*. Parts of a pipeline buried under piles of sand are uninfluenced by water streams and not likely to move much, so they will stay in good condition. On the other hand, parts of a pipeline that lie freely on a ramp or span a valley are much more likely to bend and leak due to the additional strain. Therefore it is very important that such parts of

pipelines are not recognized as noise during data cleaning. At the same time, the terrain below such free spans should not be removed either. This presents a problem in the current model, as we assume that the original surface does not have overhangs (or free-floating objects). Turning to a full three-dimensional model (with "terrains" as 2-manifolds) seems overkill compared to the small discrepancy between the model and the real world. It would therefore be interesting to see if the current model can be adapted to support free spans without going all the way to a three-dimensional model. Similarly, the algorithms may support free spans and similar structures without fundamental changes. For example, the connected component cleaning algorithm may be changed to use a more densely connected graph than the Delaunay triangulation such that there is a larger chance that points on a free span or on the terrain below it are connected to the rest of the terrain, and hence not removed.

As already hinted upon in the sections on "hybrids", the algorithms proposed in this thesis can be extended to provide better classification of point sets that correspond to the given model. At the same time, the model may also need to be refined, to on the one hand be as broad as possible to cover real world data sets, and on the other hand be useful for supporting algorithmic guarantees. Although a hybrid solution — using both a merge tree and connected component computations — seems to be an interesting direction of research, it may also be a good idea to see if we can get the same power — a global "context" of pits within pits — without going through the time-consuming merge tree generation process.

In Section 3.4, we considered what guarantees could be given for the connected component cleaning algorithm. A problem there turned out to be that noise-inducing objects with a high opacity (α) may cause the few points that are actually sampled on the terrain to also be considered as noise because they do not have any other points on the terrain as neighbours in the Delaunay triangulation. Therefore, it is interesting how high this opacity needs to be exactly before such an event is likely to happen, and if we can come up with other graphs for which the opacity can be higher before this happens. Note the correspondence of this question rising from theory to the very practical problem of supporting free spans.

As noted in the introduction, it may be interesting to use the proposed algorithms in combination with other methods for data cleaning, because different methods have different strengths and weaknesses. We already saw some interesting results for LIDAR data in Figure 5.5, where connected component cleaning is used on data that was already cleaned by another tool. Investigating combinations with other algorithms, and perhaps even integration, may yield similar useful results.

Acknowledgements

I would like to use this opportunity to thank some companies and people for their support in this project.

First of all, the data sets I received from Eiva, StatoilHydro and COWI were very helpful in all phases of this project, most visibly for being able to add the examples of real-world data sets that appear in this thesis.

Thanks also to the people from Eiva and DOF Subsea for providing a lot of valuable information about "the real world", and discussing the results of our cleaning algorithms.

I also thank Morten Revsbæk, Thomas Mølhave and Peter Hachenberger for some interesting discussions on the topic and help with TerraSTREAM.

Finally, thanks to Lars Arge and Herman Haverkort for supervising this project.

Bibliography

Agarwal, Pankaj K., Lars Arge, & Ke Yi. I/O-efficient construction of constrained Delaunay triangulations. In *Proceedings of the 13th annual European Symposium on Algorithms*, pp. 355–366 [2005]. doi:10.1007/11561071_33.

Agarwal, Pankaj K., Lars Arge, & Ke Yi. I/O-efficient batched union-find and its applications to terrain analysis. In *Proceedings of the 22nd annual Symposium on Computational Geometry*, pp. 167–176 [2006]. doi:10.1145/1137856.1137884.

Aggarwal, Alok & Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127 [1988].

Arge, Lars. The buffer tree: A technique for designing batched external data structures. *Algoritmica*, 37:1–24 [2003]. doi:10.1007/s00453-003-1021-x.

Arge, Lars, Kasper Dalgaard Larsen, Lasse Deleuran, & Morten Revsbæk. I/O-efficient contour-line generation and simplification [2009]. In progress.

Arge, Lars, Octavian Procopiuc, & Jeffrey Scott Vitter. Implementing I/O-efficient data structures using TPIE. In *Proceedings of the 10th European Symposium on Algorithms*, pp. 161–172 [2002]. doi:10.1007/3-540-45749-6_12.
URL: `http://www.madalgo.au.dk/Trac-tpie`

Berg, M. de, O. Cheong, M. van Kreveld, & M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, third edition [2008]. doi:10.1007/978-3-540-77974-2_9. Chapter 9.

Calder, Brian R. & Larry A. Mayer. Automatic processing of high-rate, high-density multibeam echosounder data. *Geochemistry Geophysics Geosystems*, 4(6) [2003]. doi:10.1029/2002GC000486.

Cheng, Siu-Wing & Sheung-Hung Poon. Surface reconstruction from noisy samples. Technical Report HKUST-TCSC-2004-05, HKUST Theoretical Computer Science Center [2004].
URL: `http://www.cse.ust.hk/tcsc/RR/2004-05.ps.gz`

Chiang, Yi-Jen, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, & Jeffrey Scott Vitter. External-memory graph algorithms. In *Proceedings of the 6th annual Symposium on Discrete Algorithms*, pp. 139–149 [1995].

Danner, Andrew. *I/O efficient algorithms and applications in geographic information systems*. Ph.D. thesis, Duke University [2006].

Danner, Andrew, Thomas Mølhave, Ke Yi, Pankaj K. Agarwal, Lars Arge, & Helena Mitasova. TerraStream: from elevation data to watershed hierarchies. In *Proceedings of the 15th ACM international symposium on advances in Geographic Information Systems* [2007]. doi:10.1145/1341012.1341049.
URL: `http://www.madalgo.au.dk/Trac-TerraSTREAM`

Dey, Tamal K. & Samrat Goswami. Provable surface reconstruction from noisy samples. *Computational Geometry: Theory and Applications*, 35:124–141 [2006]. doi:10.1016/j.comgeo.2005.10.006.

Edelsbrunner, Herbert, John Harer, & Afra Zomorodian. Hierarchical Morse–Smale complexes for piecewise linear 2-manifolds. *Discrete & Computational Geometry*, 30:87–107 [2003]. doi:10.1007/s00454-003-2926-5.

Edelsbrunner, Herbert, David Letscher, & Afra Zomorodian. Topological persistence and simplification. In *Proceedings of the 41st annual symposium on Foundations of Computer Science*, pp. 454–463 [2000]. doi:10.1109/SFCS.2000.892133.

Hall, John K. GEBCO Centennial Special Issue — Charting the secret world of the ocean floor: The GEBCO project 1903–2003. *Marine Geophysical Researches*, 27(1):1–5 [2006]. doi:10.1007/s11001-006-8181-4.

Kamel, Ibrahim & Christos Faloutsos. On packing R-trees. In *Proceedings of the 2nd international Conference on Information and Knowledge Management*, pp. 490–499 [1993]. doi:10.1145/170088.170403.

Li, Zhilin, Qing Zhu, & Chris Gold. *Digital Terrain Modeling: Principles and Methodology*. CRC Press [2004]. Chapter 7, particularly paragraph 7.5.

Mayer, Larry A. Frontiers in seafloor mapping and visualization. *Marine Geophysical Researches*, 27(1):7–17 [2006]. doi:10.1007/s11001-005-0267-x.

Revsbæk, Morten. *I/O efficient algorithms for batched union-find with dynamic set properties and its application to hydrological conditioning*. Master's thesis, Aarhus Universitet [2007].

Sithole, George & George Vosselman. Experimental comparison of filter algorithms for bare-Earth extraction from airborne laser scanning point clouds. *ISPRS Journal of Photogrammetry & Remote Sensing*, 59(1–2):85–101 [2004]. Full version: `http://www.itc.nl/isprswgIII-3/filtertest/Report.htm` (retrieved: 24-06-2009).

Welch, Greg & Gary Bishop. An introduction to the Kalman filter. Technical Report TR 95-041, University of North Carolina at Chapel Hill [2006].
URL: `http://www.cs.unc.edu/~welch/kalman/kalmanIntro.html`