Computational modelling of terrains

Ledoux, H.; Arroyo Ohori, G.A.K.; Peters, R.Y.

DOI
10.5281/zenodo.3992107

Publication date
2020

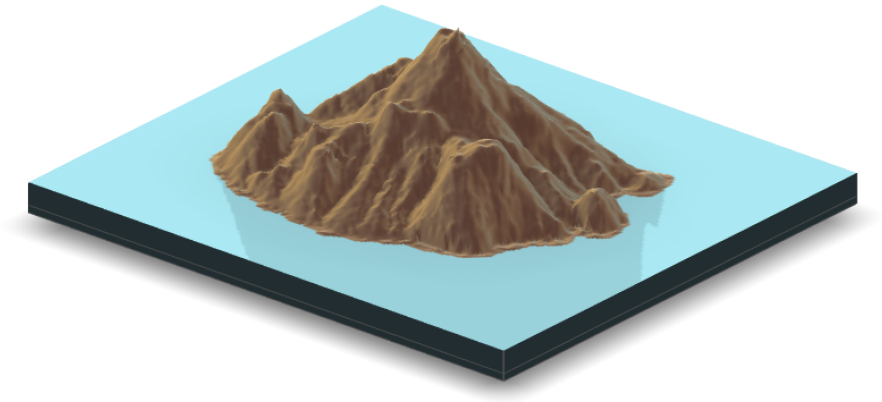
Document Version
Final published version

Citation (APA)
Ledoux, H., Arroyo Ohori, G. A. K., & Peters, R. Y. (2020). *Computational modelling of terrains*. (v0.7 ed.) GitHub/Zenodo. https://doi.org/10.5281/zenodo.3992107

Computational modelling of terrains

Hugo Ledoux Ken Arroyo Ohori Ravi Peters

v0.7.0



v0.7—2020–11–09
v0.6—2019–11–11

Download latest version
The latest version of this book can be downloaded in PDF at
https://github.com/tudelft3d/terrainbook/releases

Source code
The source code, in LaTeX, is openly available at
https://github.com/tudelft3d/terrainbook

Errors? Feedback?
Please report errors at
https://github.com/tudelft3d/terrainbook/issues

Colophon
This book was typeset with LaTeX using the `kaobook` class (https://github.com/fmarotta/kaobook). The figures were created using Ipe, OmniGraffle, Affinity Designer, or Blender. The cover image is the SRTM dataset around the Caribbean island Saba, which at 887m is the highest point of the entire Kingdom of the Netherlands; it was rendered with the R package *rayshader*.

Preface

This book presents an overview of algorithms and methodologies to reconstruct, manipulate, and extract information from terrains.

It covers different representations of terrains (eg TINs, rasters, point clouds, contour lines) and presents techniques to handle large datasets.

Open material. This book is the bundle of the lecture notes that we wrote for the course *Digitial terrain modelling* (GEO1015) in the MSc Geomatics at the Delft University of Technologies in the Netherlands. The course is tailored for MSc students who have already followed an introductory course in GIS and in programming.

Each chapter is a lesson in the course, and each lesson is accompanied by a video introducing the key ideas and/or explaining some parts of the lessons. All the videos are freely available online on the website of the course: https://3d.bk.tudelft.nl/courses/geo1015/

Who is this book for? The book is written for students in Geomatics at the MSc level, but we believe it can be also used at the BSc level.

Prerequisites are: GIS, background in linear algebra, programming course at the introductory level.

Acknowledgements. We thank Balázs Dukai for help in proof-reading, and all the students of the first year of the course (2018–2019) who helped by pointing out errors and typos. Also, the following students of the course all made pull requests to fix errors/typos: Chen Zhaiyu, Ardavan Vameghi, Li Xiaoai.

Contents

What is a terrain? 1

Defining what a terrain, also called a *digital terrain model* (DTM), is not simple because there is no universal agreement, neither among practitioners nor in the scientific literature. Different terms are used, often interchangeably.

In most cases, we can state that:

> A terrain is a representation of the Earth's surface. It gives us the *elevation*, which is the height above/below a certain reference point (a vertical datum)

However, the "Earth's surface" is also not a clear concept, since several objects on it can be present, eg man-made structures like buildings, roads, power lines, and bridges, and other objects like trees.

We use the following definitions in this course (see Figure 1.1):

DEM (**D**igital **E**levation **M**odel). In the literal meaning of the term, it is simply a model of the elevation. A DEM is either a DSM or a DTM.

DTM (**D**igital **T**errain **M**odel). The surface of the Earth is the *bare-earth*, that is no man-made objects or vegetation is present.

DSM (**D**igital **S**urface **M**odel). The surface includes all objects and structures on the terrain.

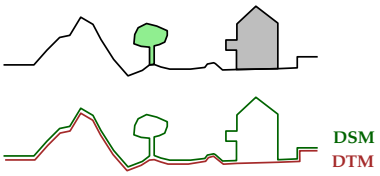


Figure 1.1: Top: a terrain with a mountain, a tree, and a building. **Bottom**: its DSM and DTM.

It should be noticed that in some countries a DEM is often synonymous with a grid of elevation (see below).

1.1 Dimensionality of DTMs

The term "3D" is misleading in a DTM context, as it is in a GIS context, because it might refer to three different concepts: 2.5D, 2.75D, and 3D (see Figure 1.2).

3D is misleading

1.1.1 2.5D

What is usually used for modelling terrains: a surface (which is a topologically a 2D object; also called a 2-manifold) is embedded in 3D space, and each location (x, y) is assigned to one and only one height z. In other words, the surface can be projected to the xy-plane and maintain its topology. When we refer to terrains in this course, this is what is usually mean, unlike explicitly stated otherwise. This is often what is used in GIS software, and both the well-known raster/grid is such a case. Observe that this restricts the real-world cases that can be modelled because, as shown in Figure 1.2b, vertical surfaces (eg walls of a building if we model all man-made objects with the terrain to construct a digital surface model), overfolds (eg the balcony of a house) and caves are impossible to represent. As shown in the figure, these are modelled as nearly vertical

2-manifold

surfaces; in practice the wall of building could deviate by for instance 1 degree from the vertical.

1.1.2 2.75D

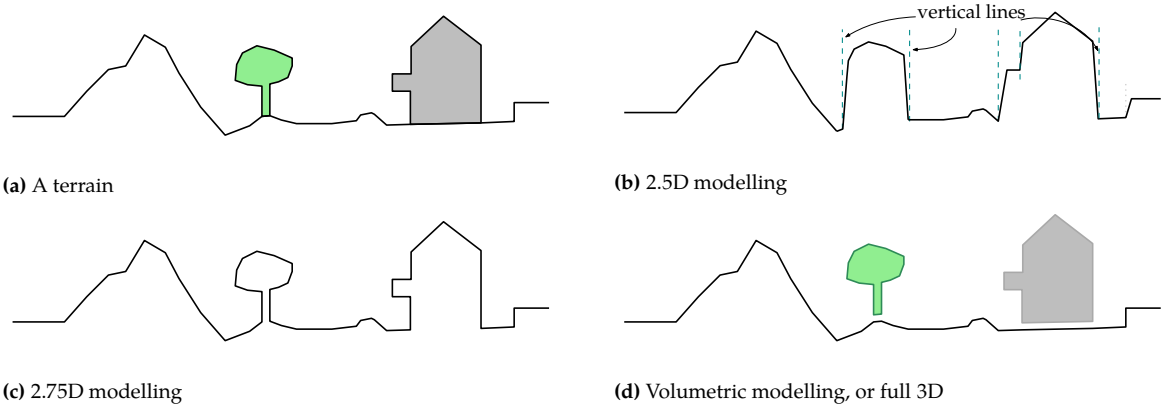
2.75D

The term "2.75D" refers to a surface (a 2-manifold) but unlike for the 2.5D case, the surface is not restricted to be projectable to the 2D plane (see Figure 1.2c). Thus, more than one z value is allowed for a given location (x, y). The term '2.75D' was coined because: it is more than 2.5D, but less than 3D. The surface represents the exterior of all objects/features together, and vertical walls are allowed. Surface modelling is popular in CAD, but in the GIS it is rather rare. We are not aware of any popular GIS software that allows us to model a terrain as a 2.75D and perform operations on it.

1.1.3 Full 3D, or volumetric modelling

volumetric modelling

This refers to the modelling of not only the boundaries of objects, but also of the interior of these. Notice for instance in Figure 1.2d that each building is represented with a solid. The volume of buildings can therefore be calculated (since the ground floor of buildings would be modelled for instance), while with the other variations it is not possible. Such a representation is usually done with a 2.5D terrain (although a 2.75D could also be used) and a set of buildings/objects that are connected to the terrain.



(a) A terrain

(b) 2.5D modelling

(c) 2.75D modelling

(d) Volumetric modelling, or full 3D

Figure 1.2: Different meanings for '3D GIS' in the context of terrains.

1.2 2.5D terrain == field

In the context of this course, we assume that a terrain is a 2.5D object, and therefore a terrain can be considered as a *field*. A field is a model of the spatial variation of an attribute a over a spatial domain, which in our case is $\mathbb{R}^2$, the two-dimensional Euclidean space. It is modelled by a function mapping one point p in $\mathbb{R}^2$ to the value of a, thus

field

$$a = f(p)$$

The function can theoretically have any number of independent variables, but in the context of a terrain the function is usually bivariate (x, y).

The representation of a field in a computer faces many problems. First, fields are continuous functions, and, by contrast, computers are discrete machines. Fields must therefore be *discretised*, ie broken into finite parts. Second, in practice it is usually impossible to measure continuous phenomena everywhere, and we have to resort to collecting samples at some finite locations and reconstructing fields from these samples. The discretisation task therefore begins at the acquisition phase, and is affected by the acquisition tools and techniques (more about this in Chapter 2). This fact is aggravated for fields as found in GIS-related disciplines because, unlike disciplines like medicine or engineering, we seldom have direct and easy access to the whole object of interest.

discretisation

1.2.1 What is needed to represent a field/terrain?

To represent a terrain in a computer, and be able to manipulate it (ie edit the terrain and extract information such as slope), two things are needed:

1. a set of samples that were collected to study the terrain, for instance a point cloud obtained from airbone laserscanning or photogrammetry (see Chapter 2 for details).
2. a set of rules to obtain one and only one elevation value at any location (x, y); in other words, to reconstruct the continuity of the surface from the discrete samples. This operation is referred to as spatial interpolation (Chapters 4 and 5).

1.2.2 Strategy #1: points + global interpolation function.

This means storing the original sample points with the parameters of the *global* spatial interpolation method that is best suited to the distribution of the samples and their accuracy. Global methods are for instance inverse-distance to a power, natural neighbours, or Kriging. This strategy is used because one can compactly represent a field (only the samples and a few parameters need to be stored).

Notice that this strategy permits us to reconstruct the continuity of a terrain from the samples by calculating the value of the elevation, but that this value is *not* persistently stored in memory. It is therefore less used in practice than the next strategy, which allows us to permanently store the terrain in a file and avoids us recomputing every time al the needed elevation values.

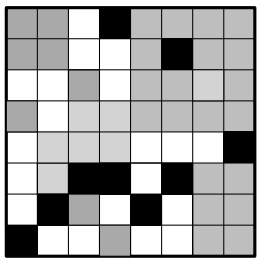
1.2.3 Strategy #2: piecewise spatial models

piecewise function

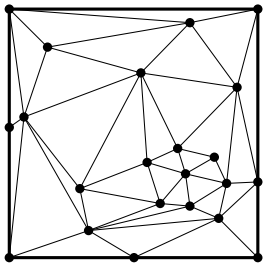
This means that the spatial interpolation function used is *piecewise* (instead of being global). That is, the two-dimensional domain of the terrain (the xy-plane) is tessellated, or partitioned, into several pieces, and for each of these we assign an interpolation function describing the spatial variation in its interior. This function is usually a simple mathematical function:

- constant function: the value of the modelled attribute is constant within one cell;
- linear function;
- higher-order function.

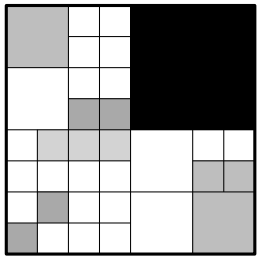
In general, we classify the tessellations of space into three categories (as shown in Figure 1.3): *regular*, *irregular*, and *hierarchical*.



regular



irregular



hierarchical

Figure 1.3: Type of tessellations.

Piecewise models typically imply that a supporting data structure is constructed, and stored, to represent the tessellation. Some of these tessellations partition arbitrarily the space, while some are based on the spatial distribution of the sample points.

1.3 Data models for representing terrains in a computer

1.3.1 Spatial data models ≠ data structures

In the GIS literature, there exists a confusion between the terms "spatial model" and "data structure". The confusion originates from the fact that object and field views of space are usually implemented in a GIS with respectively vector and raster models. However, this is not always the case as TINs can be used for fields for instance. A "spatial data model" offers an *abstract* view of a data structure, it is an abstraction of the reality. A data structure is the specific implementation of a spatial data model, and deals with storage, which topological relationships are explicitly stored, performance, etc. The same spatial data model can therefore be implemented with different data structures.

1.3.2 Regular Tessellations

pixel

As shown in Figure 1.3a, all the cells have the same shape and size. The most common regular tessellation in GIS and in terrain modelling is by far the grid (or raster representation), in which the cells are squares in 2D (usually called *pixels*, a portmanteau of 'picture' and 'element', as an analogy to digital images). However, while they are not common in practice, other regular shapes are possible, such as hexagons or triangles.

Observe that a regular tessellation often arbitrarily tessellates the space covered by the field without taking into consideration the objects embedded in it (the samples). This is in contrast with irregular tessellations in which, most of the time, the shape of the cells constructed depends on the samples.

In practice this means that, if we have a terrain stored as a regular tessellation we can assume that it was constructed from a set of samples by using spatial interpolation. Converting sample points to cells is not optimal because the original samples, which could be meaningful points such as the summits, valleys or ridges of a terrain, are not necessarily present in the resulting tessellation. There is a loss of information, since the exact location of the meaningful points are lost.

Concrete example: a 2D grid. A 2D grid, stored for instance with the GeoTIFF format, is thus a piecewise representation of a 2D field: a regular tessellation where each cell has a constant function. The value assigned to each cell is an estimation previously obtained by spatial interpolation. However, for a given grid, it is usually unclear if the value of a cell is for its centre, or for one of its vertices (and if it is the case, for which one?). Different formats have different rules, and converting a field represented with one format to another one (while retaining the same cell resolution and spatial extent) can shift the value from the centre to the top-left corner for instance.

The wide popularity of regular tessellations in terrain modelling is probably due to simplicity and to the fact that they permit us to easily integrate 2D remote sensing images and terrains. Indeed, a grid is naturally stored in a computer as an array (each cell is addressed by its position in the array, and only the value of the cell is stored), and thus the spatial relationships between cells are implicit. This is true for any dimensions, thus, contrary to other tessellations, grids are very easy to generalise to higher dimensions. The algorithms to analyse and manipulate (Boolean operations such as intersection or union) are also straightforwardly implemented in a computer.

On the other hand, grids also suffer problems. First, the size of a grid can become massive for data at a fine resolution; this problem gets worse in higher dimensions. Second, grids scale badly and are not rotationally invariant, ie if the coordinate reference system used is changed, then the grid needs to be reconstructed to obtain regular cells whose boundaries are parallel to the axes of the reference system. To assign a new value to the transformed cells, spatial interpolation is needed, which is often performed not by re-using the original samples, but by using the values of the neighbouring cells. Unfortunately, each time a grid is transformed its information is degraded because not the original samples are used, but interpolated values.

1.3.3 Irregular Tessellations

The cells of an irregular tessellation can be of any shape and size, and they usually 'follow'—or are constrained by—the samples points that were collected, albeit this is not a requirement. Subdividing the space based on the samples has the main advantage of producing a tessellation that is *adaptive* to the distribution of the samples. The subdivision is potentially better than that obtained with regular tessellations (which subdivide arbitrarily the space without any considerations for the samples).

The most known examples of the use of irregular tessellations in terrain modelling is the *triangulated irregular network*, or TIN.

TIN: triangulated irregular network

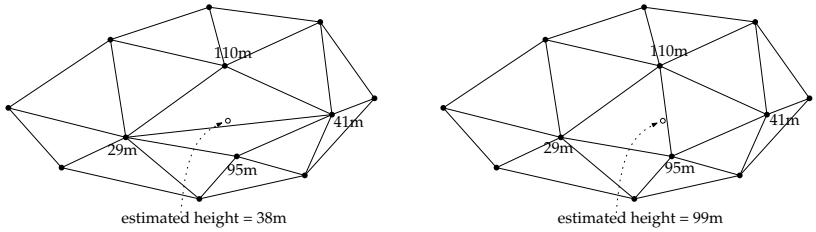


Figure 1.5: Two TINs (left is a DT, right has one non-Delaunay edge) and the result of estimating with linear interpolation in the TIN.

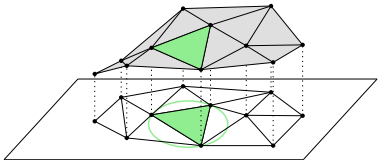


Figure 1.4: A TIN is obtained by lifting the vertices to their elevation. All the triangles are usually Delaunay, ie their circumcircle (green) is empty of any other points in the plane.

As shown in Figure 1.4, a TIN refers to an irregular tessellation of the xy-plane into non-overlapping triangles (whose vertices are formed by three sample points), and to the use of a linear interpolation function for each triangle. One way to explain the 2.5D properties of a TIN is as follows: if we project vertically to the xy-plane the triangles in 3D space forming the TIN, then no two triangles will intersect.

While not a requirement, the triangulation is usually a Delaunay triangulation (more about this in Chapter 3). The main reason is that Delaunay triangles are as "fat" as possible (long and skinny triangles are avoided), and thus they behave better for interpolation. As can be seen in Figure 1.5, the estimated value can be significantly different, and in this case the right one would make more sense since sample points that are closer to the interpolation location are used (in the TIN on the left, the value of 95m is not used).

Each of the points (which becomes vertices in the triangulation) are lifted to its elevation to create a surface, embedded in three dimensions, approximating the morphology of the terrain. The value of elevation at an unsampled location p is obtained by linearly interpolating on the plane passing through the three vertices of the triangle containing p. TINs are the most popular alternatives to 2D grids for modelling elevation; both representations have advantages and disadvantages.

A TIN in which a linear interpolation function is used yields a C^0 piecewise representation, ie it is a continuous function but at the edges of the triangles the first derivative is not possible. It is possible to use higher-order functions in each triangle of a TIN, to construct a C^1 or C^2 field, ie where the first and second derivative of the surface can be obtained. Chapter 4 gives more details.

1.3.4 Hierarchical tessellations

Hierarchical tessellations attempt to reduce the number of cells in a tessellation by merging the neighbouring cells having the same value (thus yielding cells of different sizes). While both regular and irregular tessellations can be hierarchical, in the context of the representation of terrains, the former is more relevant and is sometimes used in practice.

quadtree

A commonly used hierarchical structure in two dimensions is the *quadtree*, which is a generic term for a family of tessellations that recursively subdivide the plane into four quadrants. As is the case for grids, quadtrees are relatively easily implemented in a computer because they are trees in which each node has exactly four children, if any.

The shortcomings of regular hierarchical tessellations are similar to those of regular tessellations: the rotation and scaling operations are difficult to

handle. The main advantage of using them—saving memory space—is present only when there is spatial coherence between cells having the same attribute value, ie when they are clustered together. Indeed, the size of a quadtree is not dependent on the number of cells, but on their distribution. The quadtree of a 2D grid having no two adjacent cells with the same value (eg a checkers board) contains the same number of cells as the grid, and its size would most likely be worse because of the overhead to manage the tree. Another disadvantage is that the notion of neighbours, which is straightforward in regular tessellations, is less trivial.

1.3.5 Other common terrain representations used in GIS

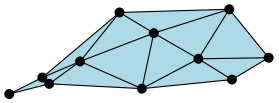
In the GIS literature, besides the ones above, different representations for terrains are often listed, the two most relevant being:

1. irregularly spaced sample points, such a point cloud;
2. contour lines.

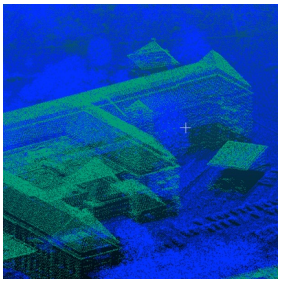
It should be noticed that these two are however *incomplete*: the set of rules to reconstruct the surface at unsampled locations is not explicitly given, they are not continuous surfaces. Conceptually speaking, these should therefore not be considered valid representations of a terrain. While this might seems odd, this is in line with the consensus among practitioners today, where a point cloud or contour lines would typically be used as an input to a process to generate a terrain.

We will nevertheless consider these in the course; the four representations we will use are shown in Figure 1.6.

raster



TIN



point cloud

contour lines

Figure 1.6: Four most common data models for terrains.

Contour lines. Given a bivariate field $f(x, y) = z$, an *isoline* (commonly named contour line) is the set of points in space where $f(x, y) = z_0$, where z_0 is a constant. Isolines have been traditionally used to represent the elevation in topographic maps and the depth in bathymetric maps for navigation at sea.

One particular property of an isoline is that its direction is always perpendicular to the direction of the steepest slope of the terrain. Another property that follows from the $2.5D$ property of the field is that contours neither intersect themselves nor each other.

The purpose of isolines on a map is to reveal the shape of the underlying terrain. By observing the shape and interrelation of neighbouring contours, the presence and significance of surface features becomes apparent; see Figure 1.7 for a few examples. It should be noticed that data between contours is absent in the contour map. Yet, in case of good contours the reader will still be able to deduct the general morphology of the field. It is even so that the use of contours will speed up the map reading process, as it conveys just that relevant bit of data to the map reader rather than 'flooding' the reader with information which essentially makes the user do his own cartographic selection. Contouring is a form of discretizing the field that makes it easier to use a map. Naturally, this comes at a price. The level of approximation of the field can (dramatically) differ between contours, the biggest error would be midway in between contour lines. But, depending on the relation between the spacing between contours

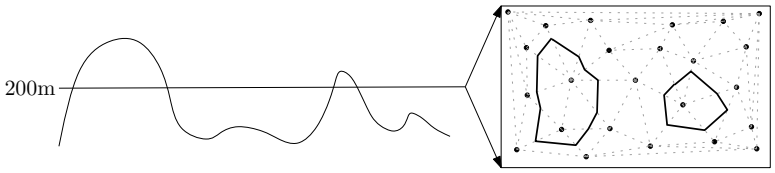


Figure 1.8: Cross-section of a terrain (left), and the 200m isoline extracted from a TIN (right).

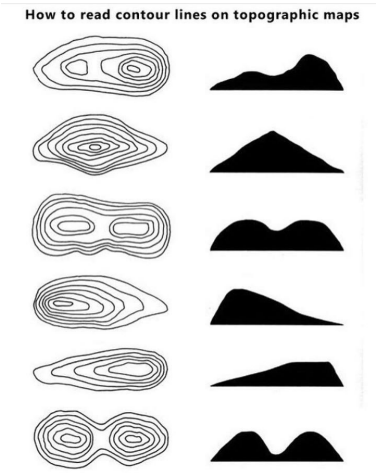


Figure 1.7: A few examples of terrain features and their contour lines. (Figure from Kjellstrom and Kjellstrom Elgin (2009))

(the *contour interval*) and the map scale, which in turn is dependent on the map application, this effect may be neglected.

In practice, isolines are only approximated from the computer representation of a field. They are usually extracted directly from a TIN or a regular grid. As shown in Figure 1.8, the idea is to compute the intersection between the level value (eg 200m) and the terrain, represented for instance with a TIN. Each triangle is scanned and segment lines are extracted to form an approximation of an isoline. Chapter 8 gives more details.

1.4 TIN versus raster for modelling terrains

There is an ongoing debate about whether TINs or rasters are the better data model to model terrains. Practitioners and scientists are probably split 50/50 on the issue.

A data model will be judged more *efficient* than another if it represents a surface more accurately within the same amount of storage space, measured in bytes. This of course depends on the data structure used to store that data model.

It should be said that both TIN and raster have advantages and disadvantages (as we will see during this course), and in practice one should choose the most appropriate model for the task at hand. This means converting between the two data models when it is necessary (topic of Chapter 8).

1.5 Notes and comments

Kumler (1994) carried out a 4-year comparison between TINs and rasters. He states that the common belief that a TIN is more space-efficient than raster is handicapped by the fact that a TIN must have *at least* 3 times less points to be of equal space. His conclusions are also that rasters can estimate elevation more accurately than comparably-sized TINs. However, he still finishes with by stating: "Yeah, well... TINs still *look* better."

Fisher (1997) discusses the disadvantages of rasters, in a GIS and remote sensing context.

Frank (1992) and Goodchild (1992) discuss at lenght the issue of data model, data structure and representation of reality.

Tse and Gold (2004) coined the term '2.75D GIS' and show an example of a where a triangulation is used to represent the surface of the Earth, with holes (for tunnels), cliffs and caves. The same idea is also referred to as a '2.8D GIS' by Gröger and Plümer (2005).

While more an academic exercise then something used in practice, multi-resolution triangulation have been described and used for some application by De Floriani and Magillo (2002).

Akima (1978) shows the advantages of using higher-order functions in each region of a TIN, to construct a C^1 or C^2 field.

Dakowicz and Gold (2003) demonstrate that using simple rules (nearest-neighbour for instance) yields fields that are not realistic and have bad slope, which is in practice problematic for several applications. Obtaining good slope from contour lines is possible, but is in practice a complex process.

1.6 Exercises

1. Explain in our own words why a point cloud (eg collected with airborne lidar) is not considered a complete representation of a terrain.
2. What is a bivariate function?
3. Assume you have a 2.75D terrain of an area. Is it possible to extract the isolines from it? What properties will these have? Will they intersect?

Acquisition of elevation measurements

2

The very first step in the process of terrain modelling is the acquisition of elevation measurements. Nowadays, these measurements are usually collected in large quantities using some form of remote sensing, ie sensors that measure—in our case—the distance to the Earth's surface from an airborne or even a spaceborne platform. In raw form, elevation measurements are typically stored as a point cloud, ie a collection of georeferenced 3D points with each point representing one elevation measurement on the Earth's surface.

There are a number of remote sensing techniques that are used to measure elevation on Earth or other planets. Typically, these techniques measure 1) the distance to the target surface and 2) their own position and orientation with respect to some global reference system. By combining these, we can compute the 3D coordinates of the measured location on the target surface.

In this chapter we will focus primarily on lidar, the most common acquisition technique for large scale terrain models with centimetre level accuracy. But we also give an overview of other acquisition techniques, for example photogrammetry, InSAR, and sonar. And to conclude we will look at typical artefacts that you might encounter while working with elevation data. This is because, as with any kind of real-world measurements, there are various uncertainties and restrictions in the acquisition process that lead to distortions—the *artefacts*—in the acquired data. These artefacts need to be taken into account when further processing and using the elevation data.

2.1 Principles of lidar

A lidar system[1] measures the distance to a target by illuminating it with pulsed laser light and measuring the reflected or *backscattered*[2] signal with a sensor (see Figure 2.1). By measuring the time-of-flight, ie the difference in time between emitting a pulse and detecting its return or *echo*, the distance to the target that reflected the pulse can be found using a simple formula. To be exact, the time-of-flight T is equal to

$$T = 2\frac{R}{c} \tag{2.1}$$

where c is the speed of light (approximately 300,000 km/s), and R is the distance or *range* between the lidar scanner and the target object that reflects the laser pulse. Therefore the range R can be found from the measured time-of-flight T using

$$R = \frac{1}{2}Tc.$$

1: While 'lidar' is often treated as the acronym of **li**ght **d**etection **a**nd **r**anging, it actually originated as a portmanteau of 'light' and 'radar'. (from Wikipedia)

2: Backscattering is the natural phenomenon of the reflection of (electromagnetic) waves or signals back to the direction they came from.

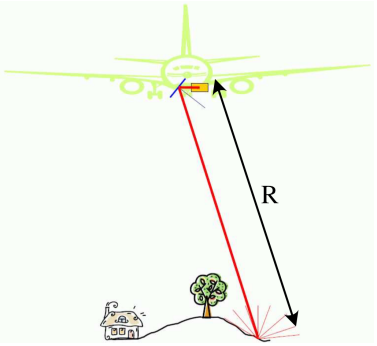


Figure 2.1: Lidar range measurment

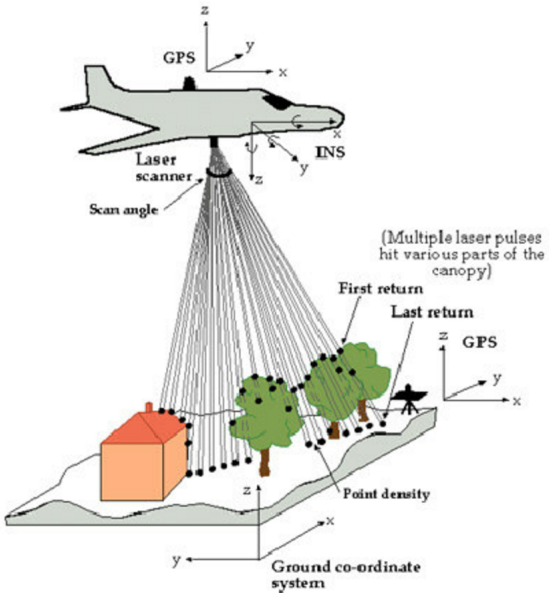


Figure 2.2: An airborne lidar system. Figure from Dowman (2004).

A typical lidar systems performs hundreds of thousands of such range measurements per second.

Lidar scanners exist in various forms. They can be mounted on a static tripod (terrestrial lidar) for detailed local scans, or on a moving platform such as a car (mobile lidar) or an aircraft (airborne lidar) for rapid scanning of larger areas. Nowadays, also hand-held lidar systems exist, and even some of the latest smartphones have a lidar sensor. Furthermore, lidar can also be used from a satellite in space[3].

3: NASA has used space lidar on Earth, on the Moon, and on Mars.

However, in the remainder of this text we will focus on airborne lidar.

2.1.1 Georeferencing the range measurements

inertial navigation system (INS)

Apart from the laser scanner itself, a lidar system uses a GPS receiver and an inertial navigation system (INS), see Figure 2.2. These devices, which respectively provide the global position and orientation of the laser scanner, are needed for georeferencing, ie to convert the range measurements of the laser scanner to 3D point measurements in a global coordinate system such as WGS84.

differential GPS

To obtain an accurate global position, *differential GPS* (DGPS) is employed. DGPS is a technique to enhance the accuracy of GPS by using GPS stations on the ground (one is visible in Figure 2.2). These DGPS stations have a known position and they broadcast the difference between that known position and the position at the station as indicated by GPS. This difference is essentially a correction for errors in the GPS signal. The aircraft receives these differences from nearby DGPS stations and uses them to correct the GPS position of the aircraft. Using DGPS the accuracy of the GPS position on the aircraft can be improved from around 15 meters to several centimetres.

To obtain the accurate orientation of the laser scanner, the INS of the aircraft is used. The INS accurately measures the orientation, ie the yaw, pitch and roll angles of the aircraft, by means of an inertial measurement unit (IMU)[4]. Only when we accurately know the orientation of the laser

4: `https://en.wikipedia.org/wiki/Inertial_measurement_unit`
inertial measurement unit (IMU)

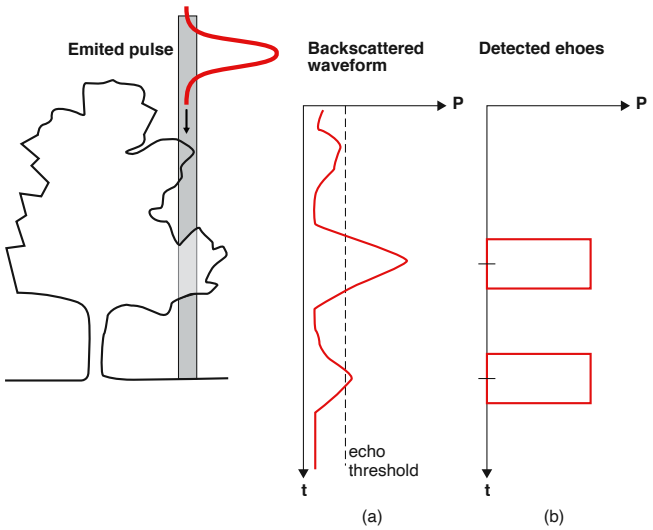


Figure 2.3: The emitted laser pulse, **(a)** the returned signal, and **(b)** the recorded echoes. Figure adapted from Bailly et al. (2012).

scanner, can we know the direction (in a global coordinate system) in which a laser pulse is emitted from the aircraft.

By combining the global position and the global orientation of the laser scanner with the range measurement from the laser scanner, the georeferenced 3D position of the point on the target object that reflected the lase pulse can be computed.

2.1.2 Echo detection

A lidar system performs ranging measurements using the time-of-flight principle that allows us to compute range from a time measurement using the known speed of light in the air. The time measurement starts when the laser pulse is emitted and is completed when a backscattered echo of that signal is detected. In practice one emitted pulse can even lead to multiple echoes in the case when an object reflects part of the laser pulse, but also allows part of the pulse to continue past the object. Notice that lidar pulses are typically emitted in a slightly divergent manner. As a result the footprint of the pules at ground level is several centimetres in diameter, which increases the likelihood of multiple echoes.

Figure 2.3 illustrates what the backscattered signal looks like when it hits a target object in the shape of a tree. A tree is particularly interesting because it often causes multiple echoes (one or more on its branches and one on the ground below). The lidar sensor observes a waveform that represents the received signal power (P) as a function of time (t). With the direct detection lidar systems that we focus on in this book, the echoes are derived from the backscattered waveform by using a thresholding technique. This essentially means that an echo is recorded whenever the power of the waveform exceeds a fixed threshold (see Figure 2.3b).

return

An echo can also be referred to as a *return*. For each return the return count is recorded, eg the first return is the first echo received from an emitted laser pules and the last return is the last received echo (see Figure 2.3). The return count can in some cases be used to determine if an echo was reflected on vegetation or ground (ground should then be the last return).

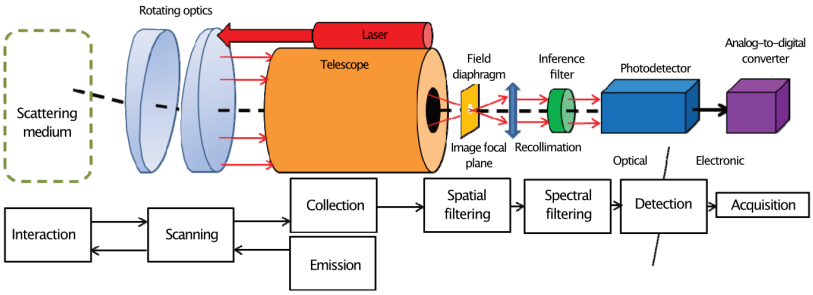


Figure 2.4: Conventional architecture of a direct detection lidar system. Figure from Chazette et al. (2016).

2.1.3 Anatomy of a lidar system

A lidar system consists of an optical and an electronic part. As shown in Figure 2.4, each part consists of several components.

In the optical part, a pulse of a particular wavelength (typically near-infrared) is generated by the laser source for each lidar measurement. It then passes through a set of optics (lenses and mirrors) so that it leaves the scanner in an appropriate direction. After the pulse interacts with the scattering medium, it is reflected back into the scanning optics which then directs the signal into a telescope. The telescope converges the signal through a field diaphragm (essentially a tiny hole around the point of convergence). The field diaphragm blocks stray light rays (eg sunlight reflected into the optics from any angle) from proceeding in the optical pipeline. Next, the light signal is recollimated so that it again consists only of parallel light rays. The final step of the optical part is the inference filter which blocks all wavelengths except for the wavelength of the laser source. This is again needed to block stray light rays from distorting the measurement.

The electronic part consists of a photodetector, which first transforms the light signal into an electrical current, which is then converted to a digital signal using the analogue-to-digital converter. Once the digital signal is available, further electronics can be used to interpret and record the signal.

2.1.4 Laser wavelength

atmospheric scattering

Choosing the optimal laser wavelength is a compromise of several different factors. One needs to consider atmospheric scattering, ie how much of the signal is lost simply by travelling through the atmosphere, and the absorption capacity of vegetation, ie how much of the signal is lost because it is absorbed by vegetation. In addition, there is the stray signal due to direct and scattered contributions of sunlight. While it is possible to filter such stray signals in the lidar system to some degree, it remains wise to choose a wavelength that is only minimally affected by it. Finally there are regulations that limit the laser radiance values permissible to the eye. This means that the power of emitted signal needs to be carefully controlled, and/or a wavelength must be chosen that is not absorbed by the eye so much.

As a result, most lidar systems use a wavelength in the near-infrared spectrum, usually between 600 and 1000 nm. A notable exception is made

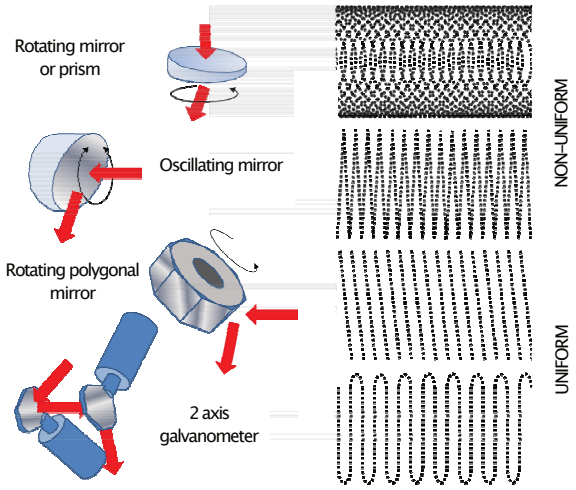


Figure 2.5: Different configurations of rotating mirrors and the associated scanning patterns from a moving platform. Arrows indicate the direction of the emitted laser signal. Figure from Chazette et al. (2016).

for bathymetric purposes, in which case a green (532 nm) laser is used because that has a greater penetration ability in water.

2.1.5 Scanning patterns

In order to improve the capacity to quickly scan large areas, a number of rotating optical elements are typically present in a lidar system. Using these optical elements, ie mirrors or prisms, the emitted laser pulse is guided in a cross-track direction (ie perpendicular to the along-track direction in which the aircraft moves, see Figure 2.1), thereby greatly increasing the scanned ground area per travelled meter of the aircraft. Figure 2.5 depicts a number of possible configurations of rotating optics and shows the resulting scanning patterns. It is clear that density of points on the ground is affected by the scanning pattern. The top example for example, yields much higher densities on edges of the scanned area. In practice more uniform patterns, such as the bottom two examples are often preferred.

To read or to watch.

This YouTube video explains the principles of an aerial LiDAR system:
`https://youtu.be/EYbhNSUnIdU`

2.2 Other acquisition techniques

Apart from lidar there are also other sensor techniques that can be used to acquire elevation data. Some of these are active sensors just like lidar (a signal is generated and emitted from the sensor), whereas others are passive (using the sun as light source). And like lidar, these sensors themselves only do range measurements, and need additional hardware such as a GPS receiver and an IMU to georeference the measurements. What follows is a brief description of the three other important acquisition techniques used in practice.

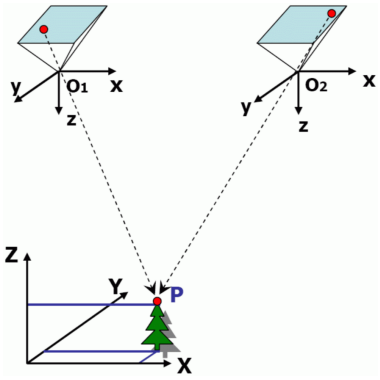


Figure 2.6: Photogrammetry

2.2.1 Photogrammetry

Photogrammetry allows us to measure the distance from overlapping photographs taken from different positions. If a ground point, called a *feature*, is identifiable in two or more images, its 3D coordinates can be computed in two steps. First, a viewing ray for that feature must be reconstructed for each image. A viewing ray can be defined as the line from the feature, passing through the projective centre of the camera, to the corresponding pixel in the image sensor (see Figure 2.6). Second, considering that we know the orientation and position of the camera, the distance to the feature (and its coordinates) can be computed by calculating the spatial intersection of several viewing rays.

dense image matching

The number of 3D point measurements resulting from photogrammetry thus depends on the number of features that are visible in multiple images, ie the so-called matches. With *dense image matching* it is attempted to find a match for every pixel in an image. If the ground sampling distance, ie the pixel size on ground level, is small (around 5cm for state-of-the-art systems), point densities of hundreds of points per square meter can be achieved, which is much higher than the typical lidar point cloud (typically up to dozens of points per square meter).

nadir images
oblique images

In photogrammetry we distinguish between *nadir* images, that are taken in a direction straight down from the camera, and *oblique* images that are taken at an angle with respect to the nadir direction. Vertical features such as building façades are only visible on oblique images. Therefore, oblique images are needed if one wants to see building façades in a dense image matching point cloud.

Because photography is used, photogrammetry gives us also the colour of the target surface, in addition to the elevation. This could be considered an advantage over lidar which captures several attributes for each point (eg the intensity of measured laser pulse and the exact GPS time of measurement), but colour is not among them.

Both airborne and spaceborne photogrammetry are possible.

2.2.2 InSAR

Interferometric synthetic aperture radar (InSAR) is a radar-based technique that is used from space in the context of terrain generation. It is quite different from airborne lidar or photogrammetry-based acquisition because of the extremely high altitude of the satellite carrying the sensor. Signals have to travel very long distances through several layers of unpredictable atmospheric conditions. As a result the speed of the radar signal is not known and the time-of-flight principle can not be used to get detailed measurements. However, by using a comprehensive chain of processing operations based on the measured phase shifts and the combination of multiple InSAR images, accurate elevation can still be measured. With InSAR it is possible to cover very large regions in a short amount of time, eg the global SRTM[5] dataset was generated with InSAR. Compared to dense image matching and lidar, InSAR-derived DTMs usually have a much lower resolution, eg SRTM has a pixel size of 30 meters.

5: https://en.wikipedia.org/wiki/Shuttle_Radar_Topography_Mission

To read or to watch.

Wikipedia page about *Interferometric synthetic-aperture radar*.

2.2.3 Echo sounding

Echo sounding is a form of sonar that can be used for bathymetry, ie mapping underwater terrains from a boat. Similar to lidar, it uses the time-of-flight principle to compute distance, but sound is used instead of light.

Single-beam and multi-beam echo sounders exist. Multi-beam systems are capable of receiving many narrow sound beams from one emitted pulse. As a result it measures the target surface much more accurately. For bathymetry usually a multi-beam echo sounder is used.

Chapter 13 describes techniques to process bathymetric datasets and create terrain of the seabed.

To read or to watch.

The principles of echo sounding.
`https://en.wikipedia.org/wiki/Echo_sounding`

2.3 Artefacts

In the acquisition process, there are many aspects—both under our control and not under our control— that affect the quality and usability of the resulting elevation data for a given application. Some examples are

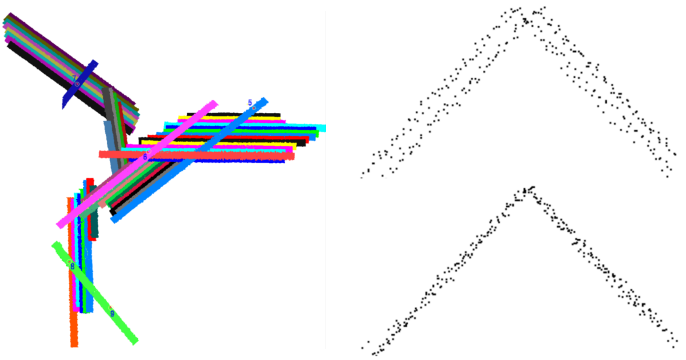
- the choice of the sensor technique,
- the sensor specifications, eg the resolution and focal length of a camera, or the scanning speed, the width of the swath, and scanning pattern of a lidar system,
- the flight parameters, eg the flying altitude and the distance and overlap between adjacent flights,
- atmospheric conditions,
- the physical properties of the target surface.

An artefact is any error in the perception or representation of information that is introduced by the involved equipment or techniques. Artefacts can result in areas without any measurements (eg the *no-data* values in a raster), or in so-called *outliers*, ie sample points with large errors in their coordinates.

outliers

We distinguish three types of artefacts,

1. those that occur due to problems in the sensor,
2. those that occur due to the geometry and material properties of the target surface,
3. those that occur due to post-processing steps.



(a) Plan view of the different strips of a lidar survey (Kornus and Ruiz, 2003)

(b) Cross-section of gable roof before (top) and after (bottom) strip adjustment (Vosselman, 2002)

Figure 2.7: Strip adjustment for lidar point clouds

2.3.1 Sensor orientation

The sensor position and orientation are continuously monitored during acquisition, eg by means of GNSS and an IMU for airborne and seaborne systems, and used to determine the 3D coordinates of the measured points. Consequently, any errors in the position and orientation of the sensor platform affect the elevation measurements. For this reason adjacent flight strips (see Figure 2.7a) often need to be adjusted to match with each other using ground control points. If the strip adjustments process fails or is omitted, a 'ghosting' effect can occur as illustrated in Figure 2.7b (top). Photogrammetry knows a similar process called aerial triangulation, in which camera positions and orientation parameters (one set for each image) are adjusted to fit with each other. Errors in the aerial triangulation can lead to a noisy result for the dense matching as seen in Figure 2.8.

2.3.2 Target surface

Many commonly occurring artefacts happen due to properties of the target surface. We distinguish three classes.

2.3.2.1 Geometry

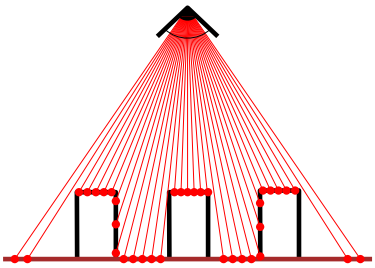


Figure 2.9: Point distribution and occlusion

The shape of the target surfaces in relation to the sensor position has a great effect on 1) local point densities and 2) occlusion. As you can see from Figure 2.9, which illustrates this for lidar, surfaces that are closest to the scanner and orthogonal to the laser beams will yield the highest point densities (see the rooftop of the middle house). Very steep surfaces on the other hand, yield relatively low point densities (see the façades of the buildings).

Occlusion happens when a surface is not visible from the scanner position. As a result there will be gaps in the point coverage, also visible in Figure 2.9. Notice how some steep surfaces and some of the adjacent ground are not registered at all by the scanner because it simply could not 'see' these parts.

The severity of both effects mostly depends on the geometry of the target objects and flight parameters such as the flying altitude and the

amount of overlap between flight strips. However, regardless of what flight parameters are chosen for a survey both effects are almost always visible somewhere in the resulting dataset, see for example Figure 2.10 for different lidar datasets for the same area.

2.3.2.2 Material properties

Depending on material properties of a target surface, signals may be reflected in a way that makes it impossible to compute the correct distance. Surfaces that act like a mirror are especially problematic, Figure 2.11 illustrates this. First, it may happen that a pulse is reflected away from the sensor, eg from a water surface, resulting in no distance measurement for that pulse. Or, in the case of photogrammetry, we will observe a different reflection in each image which heavily distorts the matching process, sometimes resulting in extreme outliers for water surfaces. In some cases, and only for active sensors, the reflected pulse does make its way back to the sensor, see for example the right half of Figure 2.11. However, it will have travelled a longer distance than it should have and the scanner only knows in which direction it emitted the pulse. This effect is called *multi-path* and the result is that points are measured at a distance that is too long and therefore they show up below the ground surface in the point cloud (see Figure 2.12).

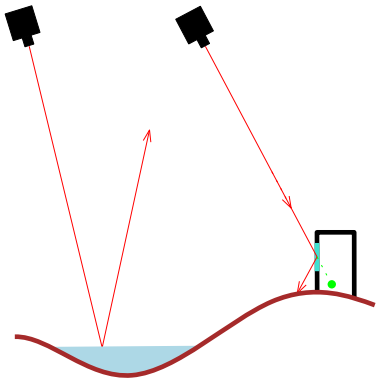


Figure 2.11: Reflection and multi-path

Photogrammetry suffers from a few other problems as well, such as surfaces that have a homogeneous texture that make it impossible to find distinguishing features that can be used for matching. This may also happen in poor lightning conditions, for example in the shadow parts of an image.

2.3.2.3 Moving objects

An example of moving objects are flocks of birds flying in front of the scanner. These can cause outliers high above the ground, as illustrated in Figure 2.12.

2.3.3 Processing

It is common to perform some kind of process after acquisition in order to fix errors caused by the reasons mentioned above. In most cases such processes are largely successful. For instance, one can attempt to fill the void regions, sometimes referred to as *no-data* regions, that are for instance due to pools of rainwater or occlusion, using an interpolation method (Figure 2.13a). Or, one can attempt to detect and remove outliers caused eg by multi-path effects or flocks of birds[6]. However, while the intention is always to reduce the number and severity of artefacts, these processes sometimes introduce distortions of their own. For example, an outlier detection algorithm may remove 'good' points if they look the same as outliers to the outlier detection algorithm (see eg Figure 2.13b). And void-filling is only effective if the void area is not too large, since interpolation methods always assume there is sufficient neighbourhood information to work with[7].

6: This is a topic of Chapter 12

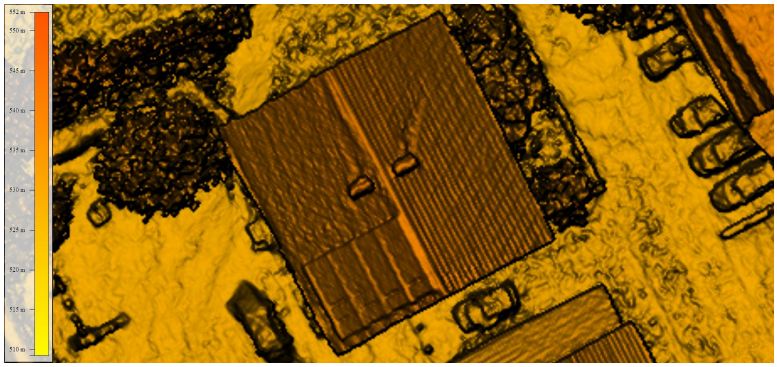
7: Chapters 4 and 5 explore the topic of spatial interpolation in detail.

To read or to watch.

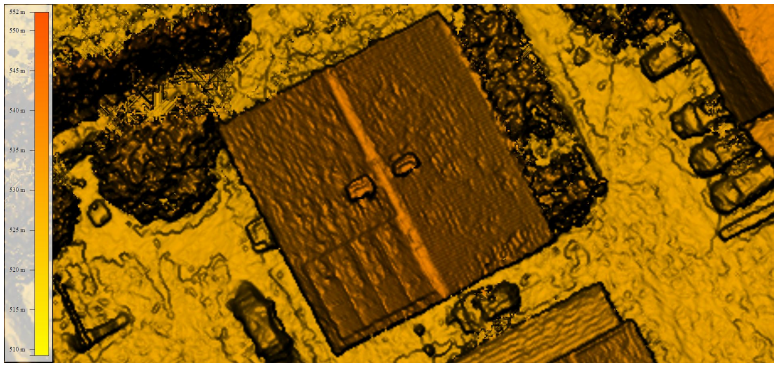
This is a paper that compares lidar and photogrammetry derived point clouds for the generation of a DEM. It shows that even when artefacts seem to be under control, both techniques may measure different elevations

C. Ressl et al. (2016). Dense Image Matching vs. Airborne Laser Scanning – Comparison of two methods for deriving terrain models. *Photogrammetrie - Fernerkundung - Geoinformation* 2016.2, pp. 57–73
PDF: https://3d.bk.tudelft.nl/courses/geo1015/data/others/Ressl16.pdf

(a) Nadir image

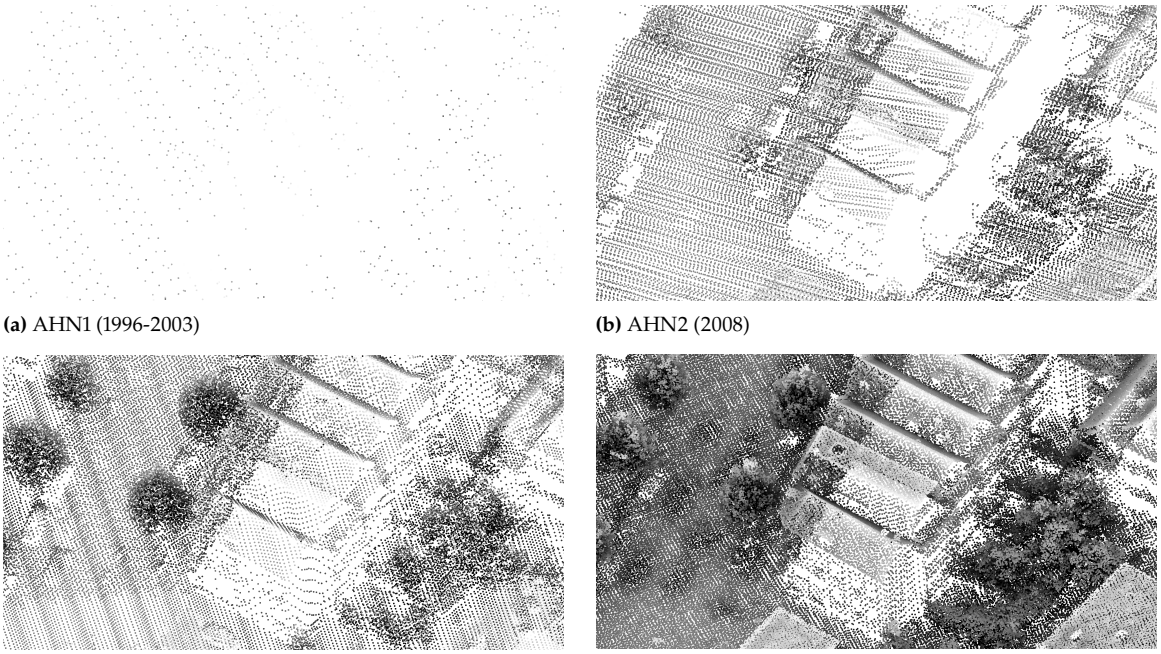


(b) DSM with good aerial triangulation



(c) DSM with poor aerial triangulation

Figure 2.8: Errors in aerial triangulation can lead to distortions in the DSM (derived from dense image matching). Images courtesy of Vermessung AVT.



(a) AHN1 (1996-2003)

(b) AHN2 (2008)

(c) AHN3 (2014)

(d) City of Rotterdam (2016)

Figure 2.10: Several lidar point clouds for the same area in the city of Rotterdam. Point distribution and occlusion effects vary.

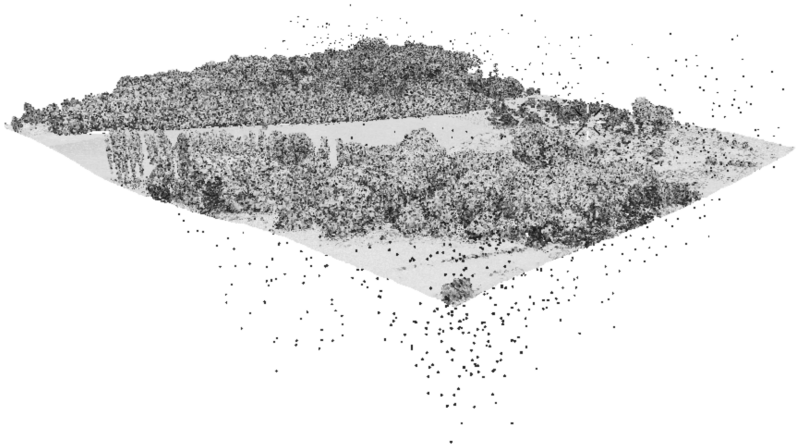
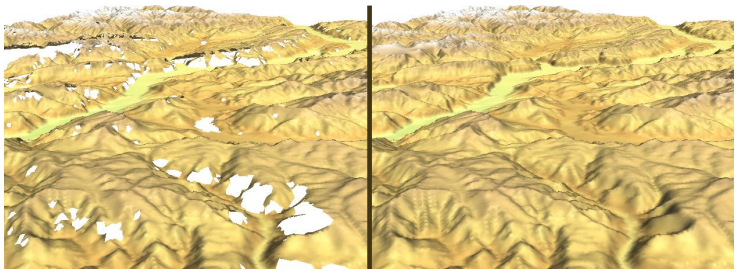
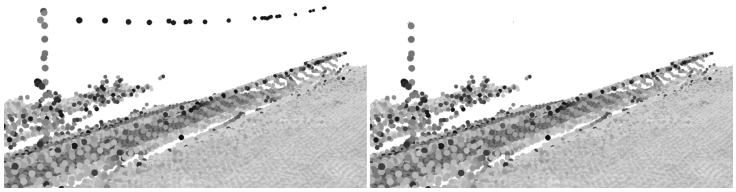


Figure 2.12: Outliers, below and above the ground, in a lidar point cloud dataset.



(a) Void-filling through interpolation in SRTM data



(b) Good points, ie those on the power line, may be removed during outlier detection

Figure 2.13: Post-processing aimed at correcting artefacts. Before processing (left) and after processing (right).

2.4 Notes and comments

If you would like to learn more about how a lidar scanner works, the chapter from Chazette et al. (2016) is recommended. More details on InSAR can be found in the manual from Ferretti et al. (2007).

Reuter et al. (2009) give an elaborate overview of the processing that needs to be done to derive a high quality (raster) DTM from raw elevation measurements.

2.5 Exercises

1. Name three differences between point cloud acquisition with lidar and with photogrammetry.
2. Explain what the time-of-flight principle entails.
3. How can you minimise occlusion effects in a point cloud during acquisition?
4. Why does positioning, using for instance GPS, play such an important role in acquisition?

Delaunay triangulations & Voronoi diagrams

3

Delaunay triangulations (DT) and Voronoi diagrams (VD) are fundamental data structures for terrains, both for their representation and for their processing (eg interpolation and several operations on terrains and point clouds are based on one of these structures).

This chapter formally defines the VD and DT in two dimensions, and introduces several concepts in computational geometry and combinatorial topology that are needed to understand, construct, and manipulate them in practice. Delaunay triangulations with constraints are also discussed.

3.1 Voronoi diagram

Let S be a set of points in $\mathbb{R}^2$ (the two-dimensional Euclidean space). The Voronoi cell of a point $p \in S$, defined $\mathcal{V}_p$, is the set of points $x \in \mathbb{R}^2$ that are closer to p than to any other point in S; that is:

$$\mathcal{V}_p = \{x \in \mathbb{R}^2 \mid \|x - p\| \leq \|x - q\|, \forall\, q \in S\}. \tag{3.1}$$

The union of the Voronoi cells of all generating points $p \in S$ form the Voronoi diagram of S, defined VD(S). If S contains only two points p and q, then VD(S) is formed by a single line defined by all the points $x \in \mathbb{R}^2$ that are equidistant from p and q. This line is the perpendicular bisector of the line segment from p to q, and splits the plane into two half-planes. $\mathcal{V}_p$ is formed by the half-plane containing p, and $\mathcal{V}_q$ by the one containing q. As shown in Figure 3.1, when S contains more than two points (let us say it contains n points), the Voronoi cell of a given point $p \in S$ is obtained by the intersection of $n - 1$ half-planes defined by p and the other points $q \in S$. That means that $\mathcal{V}_p$ is always convex. Notice also that every point $x \in \mathbb{R}^2$ has at least one nearest point in S, which means that VD(S) covers the entire space.

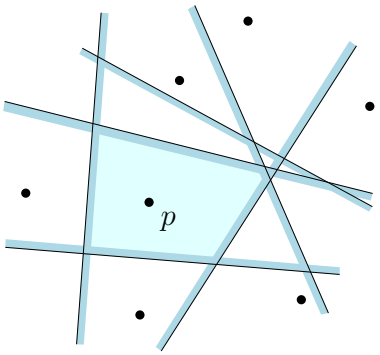


Figure 3.1: The Voronoi cell $\mathcal{V}_p$ is formed by the intersection of all the half-planes between p and the other points.

As shown in Figure 3.2, the VD of a set S of points in $\mathbb{R}^2$ is a planar graph. Its edges are the perpendicular bisectors of the line segments of pairs of points in S, and its vertices are located at the centres of the circles passing through three points in S. The VD in $\mathbb{R}^2$ can also be seen as a two-dimensional cell complex where each 2-cell is a (convex) polygon (see Figure 3.3). Two Voronoi cells, $\mathcal{V}_p$ and $\mathcal{V}_q$, lie on the opposite sides of the perpendicular bisector separating the points p and q.

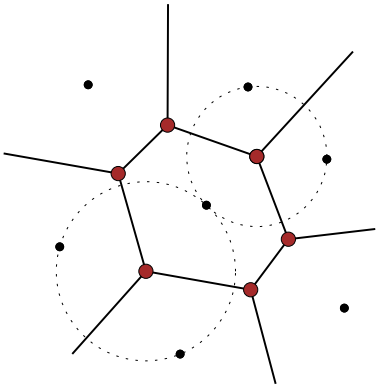


Figure 3.2: The VD for a set S of points in the plane (the black points). The Voronoi vertices (brown points) are located at the centre of the circle passing through three points in S, provided that this circle contains no other points in S in its interior.

The VD has many interesting properties, what follows is a list of the most relevant properties in the context of this course.

Size: if S has n points, then VD(S) has exactly n Voronoi cells since there is a one-to-one mapping between the points and the cells.

Voronoi vertices: a Voronoi vertex is equidistant from 3 data points. Observe for instance in Figure 3.2 that the Voronoi vertices are at the centre of circles.

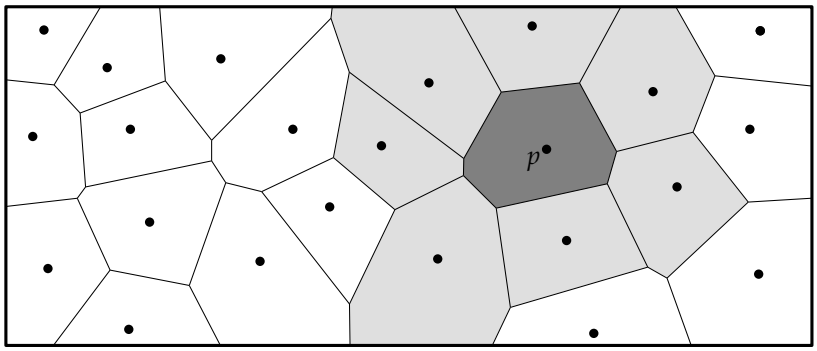


Figure 3.3: VD of a set of points in the plane (clipped by a box). The point p (whose Voronoi cell is dark grey) has seven neighbouring cells (light grey).

Voronoi edges: a Voronoi edge is equidistant from 2 points.

Convex hull: let S be a set of points in $\mathbb{R}^2$, and p one of its points. $\mathcal{V}_p$ is unbounded if p bounds conv(S). Otherwise, $\mathcal{V}_p$ is the convex hull of its Voronoi vertices. Observe that in Figure 3.2, only the point in the middle has a bounded Voronoi cell.

3.2 Delaunay triangulation

Let $\mathcal{D}$ be the VD of a set S of points in $\mathbb{R}^2$. Since VD(S) is a planar graph, it has a dual graph, and let $\mathcal{T}$ be this dual graph obtained by drawing straight edges between two points $p, q \in S$ if and only if $\mathcal{V}_p$ and $\mathcal{V}_q$ are adjacent in $\mathcal{D}$. Because the vertices in $\mathcal{D}$ are of degree 3 (3 edges connected to it), the graph $\mathcal{T}$ is a triangulation. $\mathcal{T}$ is actually called the Delaunay triangulation (DT) of S, and, as shown in Figure 3.4, partitions the plane into triangles—where the vertices of the triangles are the points in S generating each Voronoi cell—that satisfy the *empty circumcircle* test (a circle is said to be *empty* when no points are in its interior). If S is in general position, then DT(S) is unique.

3.2.1 Convex hull

The DT of a set S of points subdivides completely conv(S), ie the union of all the triangles in DT(S) is conv(S).

Let S be a set of points in $\mathbb{R}^2$, the *convex hull* of S, denoted conv(S), is the minimal convex set containing S. It is best understood with the elastic band analogy: imagine each point in $\mathbb{R}^2$ being a nail sticking out of the plane, and a rubber band stretched to contain all the nails, as shown in

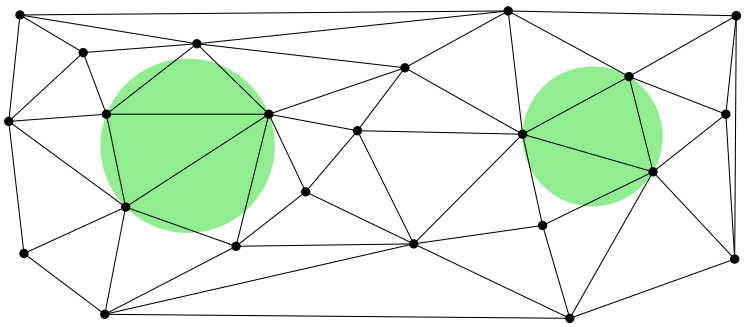


Figure 3.4: The DT of a set of points in the plane (same point set as Figure 3.3). The green circles show 2 examples of empty circumcircles.

Figure 3.5. When released, the rubber band will assume the shape of the convex hull of the nails. Notice that conv(S) is not only formed by the edges connecting the points (the rubber band), but all the points of $\mathbb{R}^2$ that are contained within these edges (thus the whole polygon).

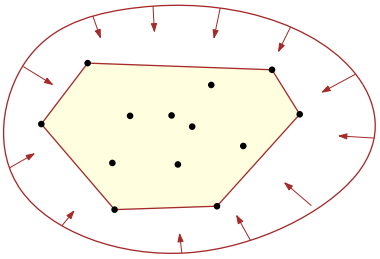


Figure 3.5: The convex hull of a set of points in $\mathbb{R}^2$.

3.2.2 Local optimality

Let $\mathcal{T}$ be a triangulation of S in $\mathbb{R}^2$. An edge σ is said to be *locally* Delaunay if it either:

(i) belongs to only one triangle, and thus bounds conv(S), or
(ii) belongs to two triangles τ_a and τ_b, formed by the vertices of σ and respectively the vertices p and q, and q is outside of the circumcircle of τ_a (see Figure 3.6).

Figure 3.6 gives an example that violates the second criteria: both p and q are contained by the circumcircles of their opposing triangles, ie of τ_b and τ_a respectively.

In an arbitrary triangulation, not every edge that is locally Delaunay is necessarily an edge of DT(S), but local optimality implies globally optimality in the case of the DT:

> Let $\mathcal{T}$ be a triangulation of a point set S in $\mathbb{R}^2$. If every edge of $\mathcal{T}$ is locally Delaunay, then $\mathcal{T}$ is the Delaunay triangulation of S.

This has serious implications as the DT—and its dual—are locally modifiable, ie we can theoretically insert, delete or move a point in S without recomputing DT(S) from scratch.

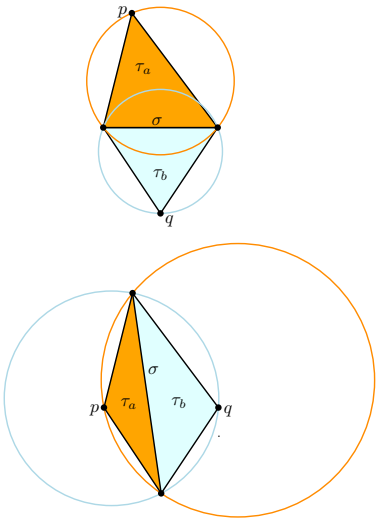


Figure 3.6: A quadrilateral that can be triangulated in two different ways. Only the top configuation is Delaunay. **(top)** σ is not locally Delaunay. **(bottom)** σ is not locally Delaunay.

3.2.3 Angle optimality

The DT in two dimensions has a very important property that is useful in applications such as finite element meshing or interpolation: the *max-min angle optimality*. Among all the possible triangulations of a set S of points in $\mathbb{R}^2$, DT(S) maximises the minimum angle (max-min property), and also minimises the maximum circumradii. In other words, it creates triangles that are as equilateral as possible. Notice here that maximising the minimum angle is not the same as minimising the maximum, and the DT only guarantees the former.

3.2.4 Lifting on the paraboloid

There exists a close relationship between DTs in $\mathbb{R}^2$ and convex polyhedra in $\mathbb{R}^3$.

Let S be a set of points in $\mathbb{R}^2$. The parabolic lifting map projects each vertex $v(v_x, v_y)$ to a vertex $v^+(v_x, v_y, v_x^2 + v_y^2)$ on the paraboloid of revolution in $\mathbb{R}^3$. The set of points thus obtained is denoted S^+. Observe that the paraboloid in three dimensions defines a surface whose vertical cross sections are parabolas, and whose horizontal cross sections are circles.

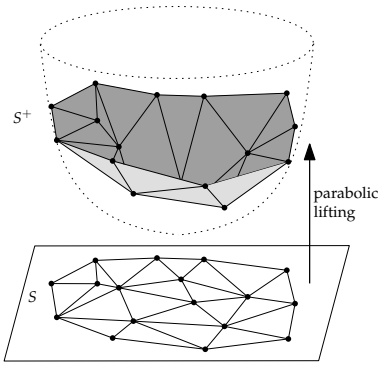


Figure 3.7: The parabolic lifting map for a set S of points $\mathbb{R}^2$.

The relationship is the following: every triangle of the lower envelope of conv(S^+) projects to a triangle of the Delaunay triangulation of S; this is illustrated in Figure 3.7 for a simple DT.

Construction of the two-dimensional DT can be transformed into the construction of the convex hull of the lifted set of points in three dimensions (followed by a simple project to the two-dimensional plane).

⚙ How does it work in practice?

Since it is easier to construct convex hulls (especially in higher dimensions, ie 4+), the DT is often constructed with this approach, even in 2D. One popular and widely used implementation is Qhull (`http://www.qhull.org/`).

3.2.5 Degeneracies

The previous definitions of the VD and the DT assumed that the set S of points is in general position, ie the distribution of points does not create any ambiguity in the two structures. For the VD/DT in $\mathbb{R}^2$, the degeneracies, or special cases, occur when 3 points lie on the same line and/or when 4 points are cocircular. For example, in two dimensions, when four or more points in S are cocircular there is an ambiguity in the definition of DT(S). As shown in Figure 3.8, the quadrilateral can be triangulated with two different diagonals, and an arbitrary choice must be made since both respect the Delaunay criterion (points should not be on the interior of a circumcircle, but more than three can lie directly on the circumcircle).

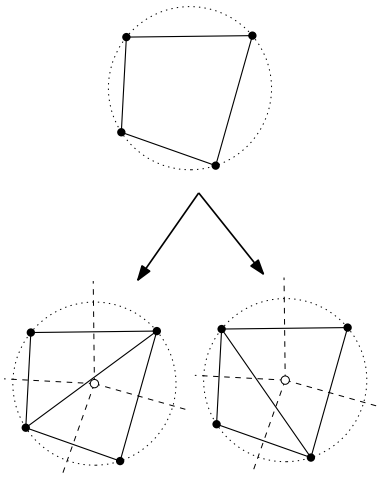


Figure 3.8: The DT for four cocircular points in two dimensions is not unique (but the VD is).

This implies that in the presence of four or more cocircular points, DT(S) is not unique. Notice that even in the presence of cocircular points, VD(S) is still unique, but it has different properties. For example, in Figure 3.8, the Voronoi vertex in the middle has degree 4 (remember that when S is in general position, every vertex in VD(S) has degree 3). When three or more points are collinear, DT(S) and VD(S) are unique, but problems with the implementation of the structures can arise.

3.3 Duality between the DT and the VD

Duality can have many different meanings in mathematics, but it always refers to the translation or mapping in a one-to-one fashion of concepts or structures. We use it in this course in the sense of the dual graph of a given graph. Let G be a planar graph, as illustrated in Figure 3.9 (black edges). Observe that G can also be seen as a cell complex in $\mathbb{R}^2$. The duality mapping is as follows (also shown in details in Figure 3.10) The dual graph $G^\star$ has a vertex for each face (polygon) in G, and the vertices in $G^\star$ are linked by an edge if and only if the two corresponding dual faces in G are adjacent (in Figure 3.9, $G^\star$ is represented with dashed lines). Notice also that each polygon in $G^\star$ corresponds to a vertex in G, and that each edge of G is actually dual to one edge (an arc in Figure 3.9) of $G^\star$ (for the sake of simplicity the dual edges to the edges on the boundary of G are not drawn).

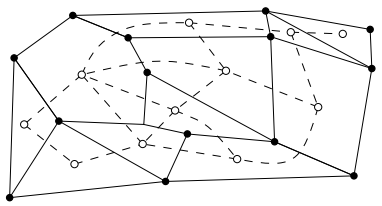
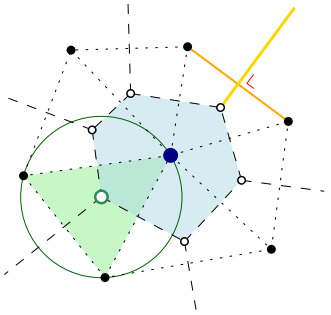


Figure 3.9: A graph G (black lines), and its dual graph $G^\star$ (dashed lines).



DT		VD
face	↔	vertex
vertex	↔	face
edge	↔	edge

Figure 3.10: Duality between the DT (dotted) and the VD (dashed).

The VD and the DT are the best example of the duality between plane graphs.

3.4 Incremental construction of the DT

Since the VD and the DT are dual structures, the knowledge of one implies the knowledge of the other one. In other words, if one has only one structure, she can always extract the other one. Because it is easier, from an algorithmic and data structure point of view, to manage triangles over arbitrary polygons (they have a constant number of vertices and neighbours), constructing and manipulating a VD by working only on its dual structure is simpler and usually preferred. When the VD is needed, it is extracted from the DT. This has the additional advantage of speeding up algorithms because when the VD is used directly intermediate Voronoi vertices—that will not necessarily exist in the final diagram—need to be computed and stored.

While there exists different strategies to construct at DT, we focus in this book on the *incremental* method since it is easier to understand and implement. An incremental algorithm is one where the structure is built incrementally; in our case this means that each point is inserted one at a time in a valid DT and the triangulation is updated, with respect to the Delaunay criterion (empty circumcircle), after each insertion. Observe that the insertion of a single point p in a DT modifies only *locally* the DT, ie only the triangles whose circumcircle contains p need to be deleted and replaced by new ones respecting the Delaunay criterion (see Figure 3.11 for an example).

In sharp contrast to this, other strategies to construct a DT (eg divide-and-conquer and plane sweep algorithms, see Section 3.7), build a DT in *one* operation (this is a batch operation), and if another point needs to be inserted after this, the whole construction operation must be done again from scratch. That hinders their use for some applications where new data coming from a sensor would have to be added.

The incremental insertion algorithm, and the other well-known algorithms, can all construct the DT of n points randomly distributed in the Euclidean plane in $\mathcal{O}(n \log n)$.

Figure 3.12 illustrates the steps of the algorithm, and Algorithm 1 its pseudo-code. In a nutshell, for the insertion of a new point p in a DT(S), the triangle τ containing p is identified and then split into three new triangles by joining p to every vertex of τ. Second, each new triangle

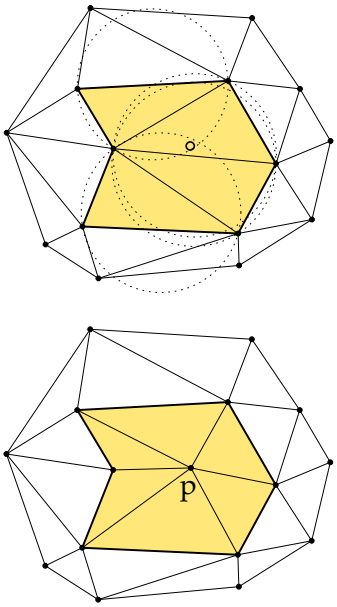


Figure 3.11: (top) The DT before and **(bottom)** after a point p has been inserted. Notice that the DT is updated only locally (only the yellow triangles are affected).

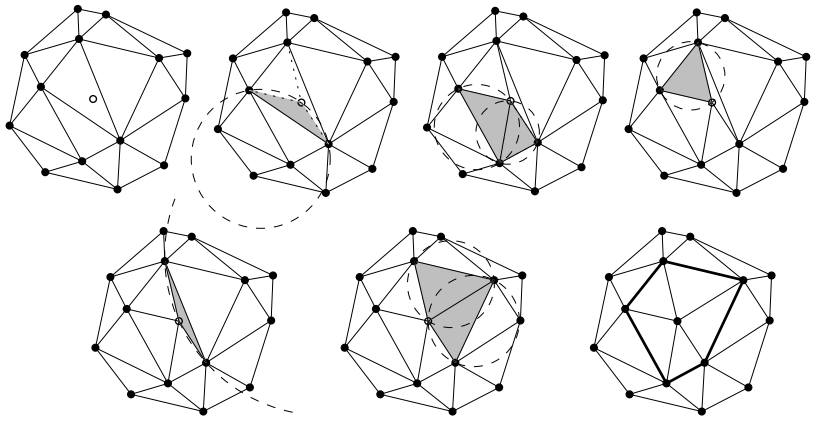


Figure 3.12: Step-by-step insertion, with flips, of a single point in a DT in two dimensions.

Algorithm 1: Algorithm to insert one point in a DT

```
1  Input: A DT(S) 𝒯, and a new point p to insert
   Output: 𝒯^p = 𝒯 ∪ {p} // the DT with point p
2  find triangle τ containing p
3  insert p in τ by splitting it in to 3 new triangles (flip13)
4  push 3 new triangles on a stack
5  while stack is non-empty do
6      τ = {p, a, b} ← pop from stack
7      τ_a = {a, b, c} ← get adjacent triangle of τ having the edge ab
8      if c is inside circumcircle of τ then
9          flip22 τ and τ_a
10         push 2 new triangles on stack
```

is tested—according to the Delaunay criterion—against its opposite neighbour (with respect to p); if it is not a Delaunay triangle then the edge shared by the two triangles is *flipped* (see below) and the two new triangles will also have to be tested later. This process stops when every triangle having p as one of its vertices respects the Delaunay criterion.

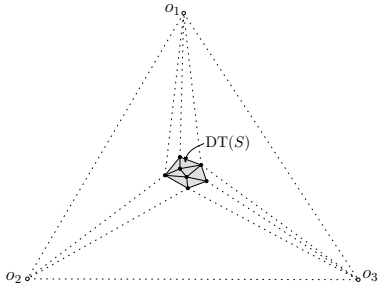


Figure 3.13: The set S of points is contained by a *big triangle* formed by the vertices o_1, o_2 and o_3. Many triangles outside conv(S) are created.

Initialisation: the big triangle. Most of the incremental DT/VD algorithms assume that the set S of points is entirely contained in a *big triangle* (τ_{big}) several times larger than the spatial extent of S; conv(S) therefore becomes τ_{big}. Figure 3.13 illustrates this. The construction of DT(S) is for example always initialised by first constructing τ_{big}, and then the points in S are inserted one by one.

Doing this has many advantages. First, when a single point p needs to be inserted in DT(S), this guarantees that p is always inside an existing triangle; we thus do not have to deal explicitly with vertices added outside the convex hull. Second, we do not have to deal with the (nasty) case of deleting a vertex that bounds conv(S). Third, since an edge is always guaranteed to be shared by two triangles, point location algorithms never "fall off" the convex hull. Fourth, identifying the vertices that bounds conv(S) is easy: they have one incident triangle that has one or more of the big triangle vertices. Fifth, the Voronoi cells of the points that bounds conv(S) will be bounded, since the only unbounded cells will be the ones of the four points of τ_{big}. This can help for some of the spatial analysis

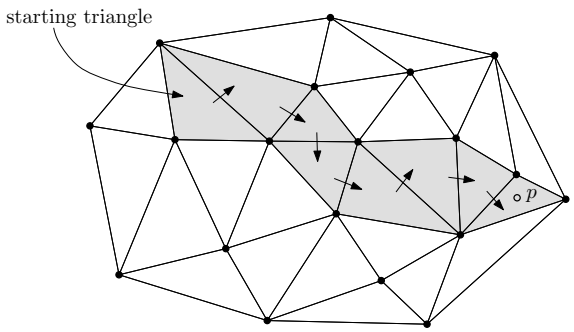


Figure 3.14: The Walk algorithm for a DT in two dimensions. The query point is p.

operations, for instance interpolation based on the VD (see Chapter 4).

The main disadvantage is that more triangles than needed are constructed. For example in Figure 3.13 only the shaded triangles would be part of DT(S). The extra triangles can nevertheless be easily marked as they are the only ones containing at least one of the three points forming τ_{big}.

How does it work in practice?

Several implementations of the DT use a big triangle, CGAL (`https://www.cgal.org/`) being one example, and these often label vertices and/or triangles as "infinite". It is therefore essential to understand the mecanism, even if one is not constructing the DT herself. There is also a variation on the big triangle: when an extra point is "at the infinity".

Point location with walking. To find the triangle containing the newly inserted point p, we can use the point-in-polygon test for every triangle (the standard GIS operation), but that brute-force operation would be very slow (complexity would be $\mathcal{O}(n)$ since each triangle must be checked).

A better alternative is to use the adjacency relationships between the triangles, and use a series of Orientation tests, as described below, to navigate from one triangle to the other. The idea, called "walking", is shown in Figure 3.14 and details are given in the Algorithm 2. The idea is as follows: in a DT(S), starting from a triangle τ (it can be any), we move to one of the adjacent triangle of τ (τ has three neighbours, we choose one neighbour τ_i such that the query point p and τ are on each side of the edge shared by τ and τ_i) until there is no such neighbour, then the simplex containing p is the current triangle τ. Notice that this algorithm is not affected by degenerate cases, and that if an Orientation test returns 0 (collinearity), then it is simply considered a positive result. This will ensure that if the query point p is located exactly at the same position as one point in S, then one triangle incident to p will be returned.

Flips. The *flip22* operation used to modify the triangulation is a simple *local* topological operation that modifies the configuration of two adjacent triangles. It is performed in constant time $\mathcal{O}(1)$. Consider the set $S = \{a, b, c, d\}$ of points in the plane forming a quadrilateral, as shown in Figure 3.15. There exist exactly two ways to triangulate S: the first one contains the triangles abc and bcd; and the second one contains the triangles abd and acd. Only the first triangulation of S is Delaunay

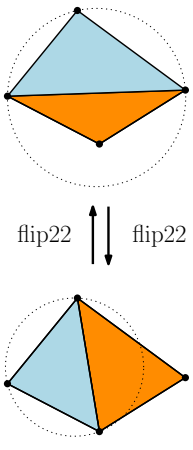


Figure 3.15: A flip22

Algorithm 2: Walk($\mathcal{T}$, τ, p)

Input: A DT(S) $\mathcal{T}$, a starting triangle τ, and a query point p
Output: τ_r: the triangle in $\mathcal{T}$ containing p

```
τ_r = None
while τ_r == None do
    visitededges = 0
    for i ← 0 to 2 do
        σ_i ← get edge opposite to vertex i in τ
        if Orientation(σ_i, p) < 0 then
            τ ← get neighbouring triangle of τ incident to σ_i
            break
        visitededges += 1
    if visitededges == 3 then
        // all the edges of τ have been tested
        τ_r = τ
Return(τ_r)
```

because d is outside the circumcircle of abc. A *flip22* is the operation that transforms the first triangulation into the second, or vice-versa.

Two other flips are possible: *flip13* and *flip31*; the numbers refer to the number of triangles before and after the flip. A flip13 refers to the operation of inserting a vertex inside a triangle, and splitting it into three triangles; and a flip31 is the inverse operation that deletes a vertex.

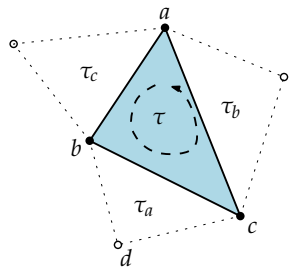
Controlling the flips. To control which triangles have to be checked and potentially flipped, we use a stack*. When the stack is empty, then there are no more triangles to be tested, and we are guaranteed that all the triangles in the triangulation have an empty circumcircle.

Predicates. Constructing a DT and manipulating it essentially require two basic geometric tests (called *predicates*): Orientation determines if a point p is left, right or lies on the line segment defined by two points a and b; and InCircle determines if a point p is inside, outside or lies on a circle defined by three points a, b and c. Both tests can be reduced to the computation of the determinant of a matrix:

$$\textsc{Orientation}(a,b,p) = \begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ p_x & p_y & 1 \end{vmatrix} \tag{3.2}$$

$$\textsc{InCircle}(a,b,c,p) = \begin{vmatrix} a_x & a_y & a_x^2+a_y^2 & 1 \\ b_x & b_y & b_x^2+b_y^2 & 1 \\ c_x & c_y & c_x^2+c_y^2 & 1 \\ p_x & p_y & p_x^2+p_y^2 & 1 \end{vmatrix} \tag{3.3}$$

* A data structure: https://en.wikipedia.org/wiki/Stack_(abstract_data_type)



triangle	v_1	v_2	v_3	adj_1	adj_2	adj_3
τ	a	b	c	τ_a	τ_b	τ_c
τ_a	b	d	c	$\tau_{...}$	τ	$\tau_{...}$
...	...	...	...	...	...	...

Figure 3.16: The triangle-based data structure to store efficiently a triangulation (and the adjacency relationships between the triangles).

3.5 Data structures for storing a DT

A triangulation is simply a subdivision of the plane into polygons, and thus any data structure used in GIS can be used to store a triangulation.

Simple Features: while many use this (PostGIS and any triangulation you see in Shapefiles), this is not smart: (1) the topological relationships between the triangles are not stored; (2) the vertices are repeated for each triangle (and we know that for a Poisson distribution of points in the plane a given point has exactly 6 incident triangles).

Edge-based structures: all the edge-based topological data structure used for storing planar graphs (eg DCEL, half-edge, winged-edge, etc) can be used. These usually lead to large storage space.

Observe that in practice, if only the DT is wanted (and not the constrained one, see below), practitioners will often simply store the sample points and reconstruct on-the-fly the DT, since it is unique (if we omit points not in general position that is).

However, because it is simpler to manage triangles over arbitrary polygons (they always have exactly 3 vertices and 3 neighbours), data structures specific for triangulations have been developed and are usually used.

The simplest data structure, as shown in Figure 3.16, considers the triangle as being its atom and stores each triangle with 3 pointers to its vertices and 3 pointers to its adjacent triangles. Observe that the order in which the vertices and adjacent triangles stored correspond to each other. This is an important property that allows an efficient retrieval of triangles in the Walk algorithm (Algorithm 2) for instance.

3.6 Constrained and Conforming Delaunay Triangulations

Given as input a set S of points and straight-line segments in the plane, different triangulations of S (so that the segments are respected) can be constructed. We are mostly interested in the *constrained Delaunay triangulation* (ConsDT) and the *conforming Delaunay triangulation* (ConfDT), see Figure 3.17 for one example.

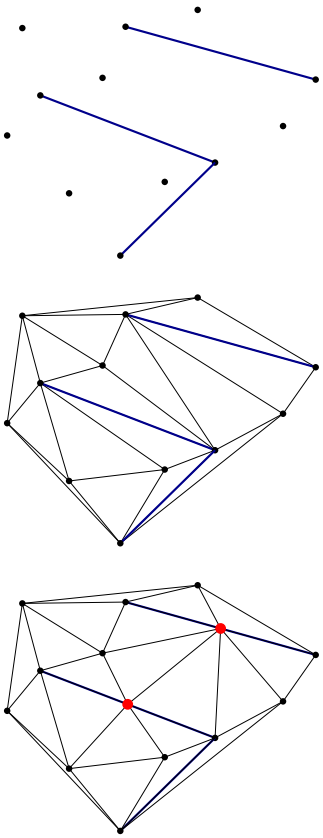


Figure 3.17: (top) A set S of points and straight-line segments. **(middle)** Constrained DT of S. **(bottom)** Conforming DT of S; the Steiner points added are in red.

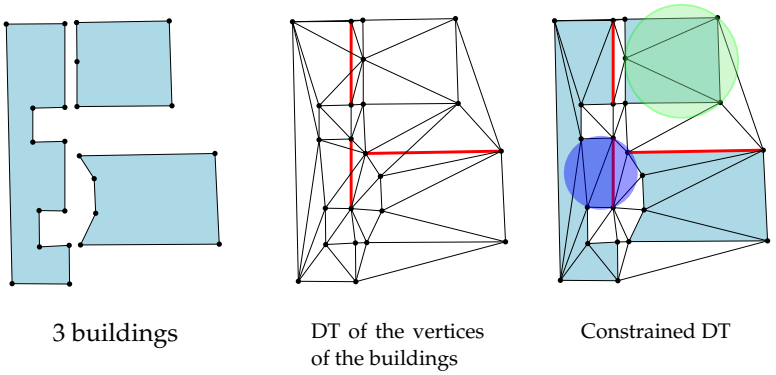


Figure 3.18: The ConsDT of a set of segments. On the right, the triangle whose circumcircle is green is a Delaunay (no other points in its interior) and so is the triangle whose circumcircle is in purple (there is one point in its interior, but it cannot be seen because of the constrained segment).

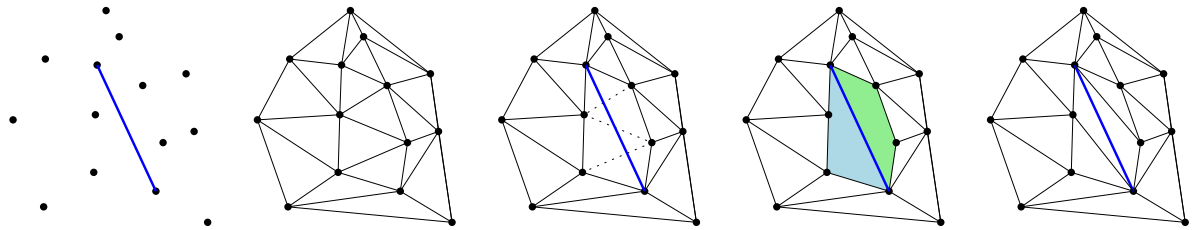


Figure 3.19: Steps to construct a ConsDT.

Constrained DT (ConsDT). Given a set S of points and straight-line segments in $\mathbb{R}^2$, the ConsDT permits us to decompose the convex hull of S into non-overlapping triangles, and every segment of S appears as an edge in ConsDT(S). ConsDT is similar to the Delaunay triangulation, but the triangles in ConsDT are not necessarily Delaunay (ie their circumcircle might contain other points from S). The empty circumcircle for a ConsDT is less strict: a triangle is Delaunay if its circumcircle contains no other points in S that are *visible* from the triangle. The constrained segments in S act as visibility blockers. Figure 3.18 shows one example.

Without going into details about one potential algorithm, one way to construct a ConsDT(S) is (see Figure 3.19):

1. construct DT(S^p), where S^p is the set containing all the points in S and the end points of the line segments (Figure 3.19b)
2. insert each line segment, each insertion will remove edges from DT(S^p). In Figure 3.19c 3 edges are removed.
3. this creates 2 polygons that need to be retriangulated, in Figure 3.19d there is a blue and a green one.
4. retriangulate each separately, the Delaunay criterion needs to be verified only for the vertices incident to the triangles incident to the hole/polygon.

Observe that the ConsDT can be used to triangulate polygons with holes (see Figure 3.20), it suffices to remove the triangle outside the exterior boundary, but inside the convex hull.

Conforming DT (ConfDT). A ConfDT adds new points to the input S (called *Steiner* points) to ensure that the input segments are present in the triangulation. As Figures 3.17 and 3.20 show, the input straight-line segments will be potentially split into several collinear segments. The

Steiner points have to be carefully chosen (where to put them is beyond the scope of this course).

Observe that each triangle in a ConfDT respect the Delaunay criterion, but that more triangles are present. If 2 segments are nearly parallel, many points could be necessary (for m segments, up to m^2 could be necessary).

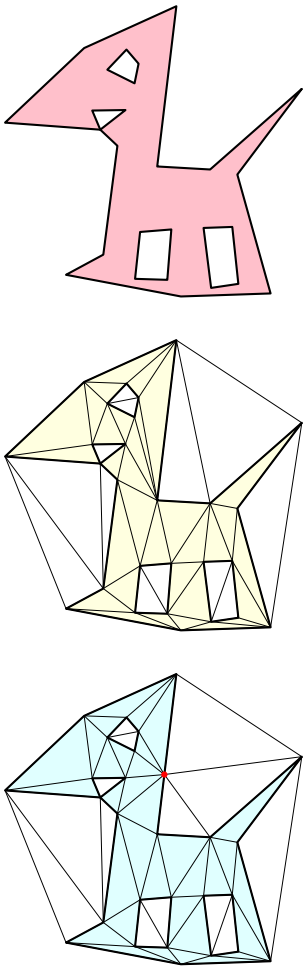


Figure 3.20: (top) One polygon with 4 holes (interior rings). **(middle)** its ConsDT. **(bottom)** its ConfDT (the Steiner point added is in red).

Steiner point

3.7 Notes and comments

The DT and the VD have been discovered, rediscovered and studied many times and in many different fields, see Okabe et al. (2000) for a complete history. The VD can be traced back to 1644, when Descartes used Voronoi-like structures in Part III of his *Principia Philosophiæ*. The VD was used by Dirichlet (1850) to study quadratic forms—this is why the VD is sometimes referred to as *Dirichlet tessellation*—but was formalised and defined by Voronoi (1908). The first use of the VD in a geographical context is due to Thiessen (1911), who used it in climatology to better estimate the precipitation average around observations sites; the DT was formalised by Delaunay (1934).

For the construction of the DT, the incremental algorithm was first described by Lawson (1972). Guibas and Stolfi (1985) describe a divide-and-conquer algorithm, and Fortune (1987) a sweep-line one.

The local optimality of a DT, which implies globally optimality in the case of the DT, was proven by Delaunay (1934) himself. The *max-min angle optimality* of the DT was firstly observed by Sibson (1978). This paraboloic lifting was first observed by Brown (1979) (who used a spherical transformation), further described by Seidel (1982) and Edelsbrunner and Seidel (1986).

Instead of using a *big triangle*, several will use an "infinite point" or a "far-away point", which is conceptually the same but numerically more stable (since the size of the big triangle does not need to be defined). Liu and Snoeyink (2005) explains the details.

The walking algorithm described, with a few modifications, can perform point location in $\mathcal{O}(n^{1/3})$. The walking in the triangulation to locate the triangle containing a newly inserted point is not the fastest solution, Mücke et al. (1999) and Devillers et al. (2002) discuss alternatives that are optimal (ie $\mathcal{O}(\log n)$). However, they both note that optimal algorithms do not necessarily mean better results in practice because of the amount of preprocessing involved, the extra storage needed, and also because the optimal algorithms do not always consider the dynamic case, where points in the DT could be deleted.

Several criteria for constructing data-dependent triangulations are discussed in Dyn et al. (1990). While these can be used, in practice it was proven that the Delaunay triangulation is still the triangulation that minimises the roughness of a surface (Wang et al., 2001; Rippa, 1990)

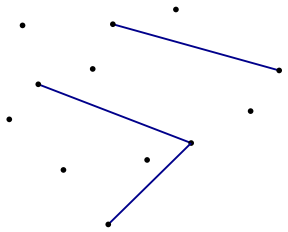
Shewchuk (1997) shows that while the triangle-based data structure requires twice as much code as with the quad-edge (to store and construct a ConsDT), the result is that the code runs twice as fast and the memory

requirement as about 2X less. CGAL (https://www.cgal.org/), among many others, uses the triangle-based data structure.

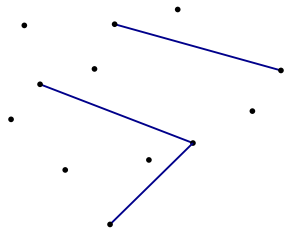
Since a DT can be locally modified by adding one point (and not reconstructing the whole structure from scratch, see Figure 3.11), it is also possible to delete/remove one vertex from a DT with only local operations. Mostafavi et al. (2003) and Devillers (2009) describe algorithms.

3.8 Exercises

1. A DT has 32 triangles and we insert a new point p that falls inside one of the triangles. If we insert and update the triangulation (for Delaunay criterion), what is the number of triangles?
2. If a given vertex v in a DT has 7 incident triangles, how many vertices will its dual polygon contain?
3. A DT has 6 vertices, and 3 of these are forming the convex hull. How many triangles does the DT have?
4. Assume you have 8 points located on a circle. Draw the DT and the VD of these 8 points.
5. When inserting points in a DT (Algorithm 1), what happens if a new points is inserted directly on an edge? Line 2 states that the triangle is split into 3 new triangles, does it still hold?
6. Given the input formed of elevation points and breaklines below (both projected to the xy-plane), draw both the constrained and conforming Delaunay triangulation (an approximation is fine).



Constained DT



Conforming DT

Spatial interpolation: deterministic methods

4

Given a set S of points p_i in $\mathbb{R}^2$ (also called samples or data points in the following) to which an attribute a_i is attached, spatial interpolation is the procedure used to estimate the value of the attribute at an unsampled location x. Its goal is to find a function $f(x, y)$ that fits (pass through, or close to, all the points in S) as well as possible. There is an infinity of such functions, some are global and some are piecewise, the aim is to find one that is best suited for the kind of datasets used as input. Interpolation is based on *spatial autocorrelation*, that is the attribute of two points close together in space is more likely to be similar than that of two points far from each other.

It should be noticed that the natural spatial extent of a set of sample is its *convex hull*, and that an estimation outside this convex hull is *extrapolation* (Figure 4.1). Extrapolating implies that more uncertainty is attached to the estimated value.

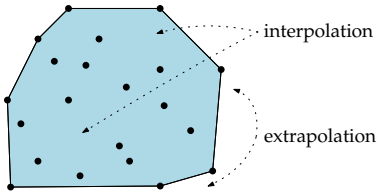


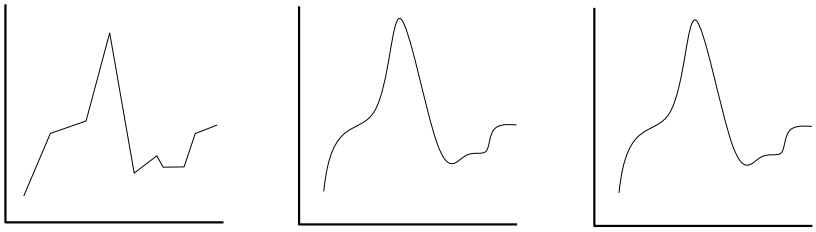
Figure 4.1: Spatial interpolation and extrapolation.

Spatial interpolation methods are crucial in the visualisation process (eg generation of contours lines), for the conversion of data from one format to another (eg from scattered points to raster), to have a better understanding of a dataset, or simply to identify 'bad' samples. The result of interpolation—usually a surface that represents the terrain—must be as accurate as possible because it often forms the basis for spatial analysis, for example runoff modelling or visibility analysis. Although interpolation helps in creating three-dimensional surfaces, in the case of terrains it is intrinsically a two-dimensional operation because only the (x, y) coordinates of each sample are used, and the elevation is the dependent attribute. Notice that the attribute used need not be only elevation, for other GIS applications one could use the spatial interpolation methods below for for instance rainfall amounts, percentage of humidity in the soil, maximum temperature, etc. Spatial interpolation in 3D is also possible (but out of scope for this course), in that case there are 3 independent variables (x, y, z) and one dependent variable, for instance the temperature of the sea at different depth, or the concentration of a certain chemical in the ground.

4.1 What is a good interpolation method for terrains?

The essential properties of an 'ideal' interpolation method for bivariate geoscientific datasets are as follows:

1. **exact**: the interpolant must 'honour' the data points, or 'pass through' them.
2. **continuous**: a single and unique value must be obtained at each location. This is called a C^0 interpolant in mathematics (see Figure 4.2).



(a) C^0 interpolant (b) C^1 interpolant (c) C^2 interpolant

Figure 4.2: C^0 interpolant is a function that is continuous but the first derivative is not possible at certain locations; C^1 interpolant has its first derivative possible everywhere; C^2 interpolant has its second derivative possible everywhere (this one is more difficult to draw).

3. **smooth**: it is desirable for some applications to have a function for which the first or second derivative is possible everywhere; such functions are respectively referred to as C^1 and C^2 interpolants.
4. **local**: the interpolation function uses only some neighbouring samples to estimate the value at a given location. This ensures that a sample with a gross error will not propagate its error to the whole interpolant.
5. **adaptability**: the function should give realistic results for anisotropic data distributions and/or for datasets where the data density varies greatly from one location to another.
6. **computationally efficient**: it should be possible to implement the method and get an efficient result. Efficiency is of course subjective. For a student doing this course, efficiency might mean that the method generates a result in matter of minutes or an hour on a laptop, for the homework dataset. For a mapping agency, running a process for a day on a supercomputer for a whole country might be efficient. Observe that the complexity of the algorithm is measured not only on the number n of points in the dataset, but how many neighbours k are used to perform one location estimation.
7. **automatic**: the method must require as little input as possible from the user, ie it should not rely on user-defined parameters that require *a priori* knowledge of the dataset.

4.2 Fitting polynomials

4.2.1 One global function

We know that if we have n points in S in in $\mathbb{R}^3$ (the samples are lifted to their elevation), there is one polynomial of degree at most $n - 1$.

This interpolant will be exact, continuous and smooth (at least C^2). However, it will not be local (which is problematic for terrains), and finding the polynomial of a high degree for large datasets might be impossible (or take a lot of time).

The biggest concern polynomials is probably that while the interpolant is exact (the surface passes through the sample points), higher-degree polynomials can oscillate between the samples and 'overshoot', ie be (far) outside the minimum or maximum z values of the the set S. This is known as the Runge's phenomenon in numerical analysis, and is shown in Figure 4.3.

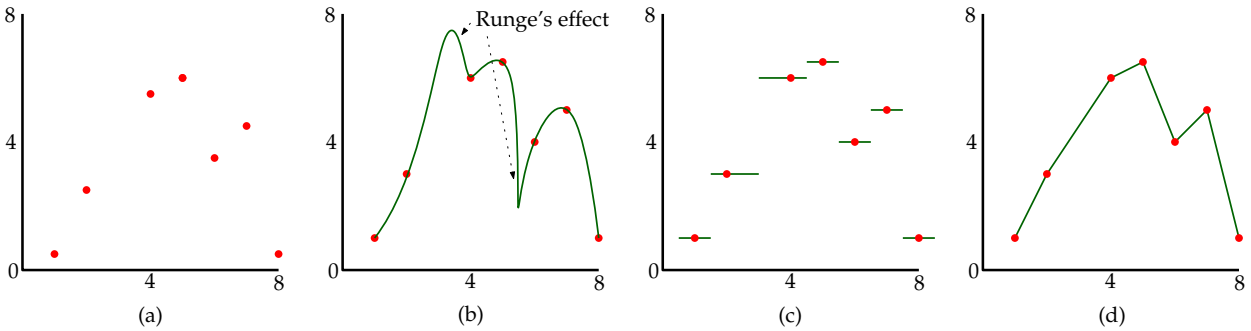


Figure 4.3: A few of the interpolation methods shown for a 1D dataset. **(a)** Input sample points. **(b)** Polynomial fitting, and the Runge's effect shown. **(c)** Natural neighbour. **(d)** Linear interpolation in TIN.

4.2.2 Splines: piecewise polynomials

Splines are piecewise polynomials, each piece is connected to its neighbouring piece in a *smooth* manner: along the edges and at the data points the function is usually still C^1 or C^2 (in other words, where 2 or more polynomials connect, they have the same values for their tangents).

In practice, for terrain modelling, splines are preferred over one polynomial function because of the reasons mentioned above (mostly Runge's effect) and because computing the polynomial for large datasets is very inefficient. The polynomials used in each piece of the subdivision is usually of low degree (≤ 5)

There are several types of splines (and variation of them, such as Bézier), and most of them are not suited for terrains. The most used spline in practice seems to be the *regularised spline with tension* (RST), where the dataset is decomposed into square pieces of a certain size. The Runge's effect (also called *overshoots*) are eliminated (since the degree is low), and the tension parameter can be tuned to obtain an interpolant that is smooth. The method is available in the GRASS GIS.

4.3 Weighted-average methods

The five interpolation methods discussed in this section are *weighted-average methods*. These are methods that use a subset of the sample points, to which a weight (importance) are assigned, to estimate the value of the dependent variable. The interpolation function f of such methods, with which we obtain an estimation $\hat{a}$ of the dependent variable a_i, have the following form:

$$f(x) = \hat{a_i} = \frac{\sum_{i=1}^{n} w_i(x)\, a_i}{\sum_{i=1}^{n} w_i(x)} \tag{4.1}$$

where $w_i(x)$ is the weight of each neighbouring data point p_i (with respect to the interpolation location x) used in the interpolation process. A neighbour p_i here is a sample point that is used to estimate the value of location x. In the context of terrain modelling, the attribute a is the elevation.

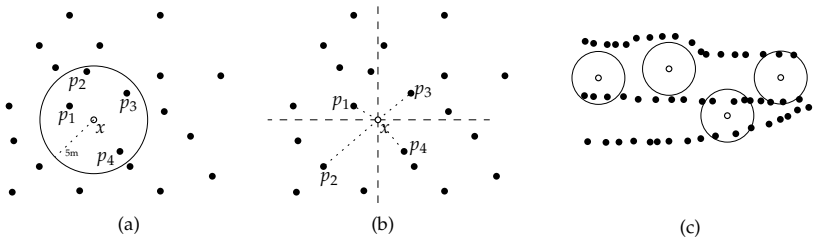


Figure 4.5: (a) IDW interpolation with a searching circle, and the weight assigned to each neighbour used in the estimation. **(b)** IDW by choosing the closest neighbour in each quadrant. **(c)** It has (serious) problems with datasets whose distribution of samples is anisotropic.

4.3.1 Nearest neighbour interpolation (nn)

Nearest neighbour, or closest neighbour, is a simple interpolation method: the value of an attribute at location x is simply assumed to be equal to the attribute of the nearest data point. This data point gets a weight of *1.0*. Given a set S of data points, if interpolation is performed with this method at many locations close to each other, the result is the Voronoi diagram (VD) of S, where all the points inside a Voronoi cell have the same value.

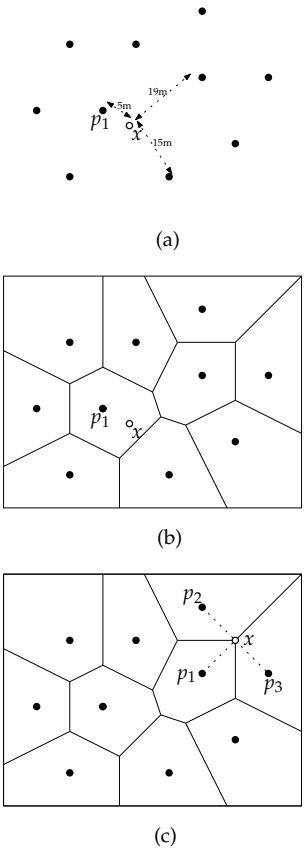


Figure 4.4: (a) Nearest neighbour: the estimated value at x is that of the closest data point. **(b)** the Voronoi diagram can be used. **(c)** Ambiguity because p_1, p_2, and p_3 are equidistant from x; this causes discontinuities in the resulting surface.

Although the method possesses many of the desirable properties (it is exact, local and can handle anisotropic data distributions), the reconstruction of continuous fields can not realistically be done using it since it fails lamentably properties 2 and 3. The interpolation function is indeed discontinuous at the border of cells; if the location x is directly on an edge or vertex of the VD(S), then which value should be returned?

The implementation of the method sounds easy: simply find the closest data point and assign its value to the interpolation location. The difficulty lies in finding an efficient way to get the closest data point. The simplest way consists of measuring the distance for each of the n points in the dataset, but this yields a $\mathcal{O}(n)$ behaviour for each interpolation, which is too slow for large datasets. To speed up this brute-force algorithm, auxiliary data structures that will spatially index the points must be used, see for instance the kd-tree in Section 10.2. This would speed up each query to $\mathcal{O}(\log n)$.

4.3.2 Inverse distance weighting (IDW)

Inverse distance weighting (IDW)—also called inverse distance to a power, or distance-based methods—is a family of interpolation methods using distance to identify the neighbours used, and to assign them weights. IDW is probably the most known interpolation method and it is widely used in many disciplines. As shown in Figure 4.5a, in two dimensions it often uses a 'searching circle', whose radius is user-defined, to select the data points p_i involved in the interpolation at location x. It is also possible to select for instance the 10 or 15 closest data points, or do that according to certain directions (ie you can select for example 3 data points in each quadrant; Figure 4.5b shows the case where the closest in each quadrant is used).

The weight $w_i(x)$ assigned to each p_i for a location x is:

$$w_i(x) = |xp_i|^{-h} \tag{4.2}$$

where h defines the power to be used, and $|ab|$ is the distance between a and b. The power h is typically 2, but other weights, such as 3, can also be used. A very high power, say 5, will assign very little importance to points that are far away.

It should be emphasised that the size of the radius of the searching circle influences greatly the result of the interpolation: a very big radius means that the resulting surface will be smooth or 'flattened'; on the other hand, a radius that is too small might have dramatic consequences if for example no data points are inside the circle (Figure 4.5c shows one example). A good knowledge of the dataset is thus required to select this parameter.

This method has many flaws when the data distribution varies greatly in one dataset because a fixed-radius circle will not necessarily be appropriate everywhere in the dataset. Figure 4.5c shows one example where one circle, when used with a dataset extracted from contour lines, clearly gives erroneous results at some locations. The major problem with the method comes from the fact that the criterion, for both selecting data points and assigning them a weight, is one-dimensional and therefore does not take into account the spatial distribution of the data points close to the interpolation location.

IDW is exact, local, and can be implemented in an efficient manner. However, finding all the points inside a given radius requires using an auxiliary data structure (such as a kd-tree, see Section 10.2) otherwise each interpolation requires $\mathcal{O}(n)$ operations. Also, as mentioned above, there are cases where IDW might not yield a continuous surface (nor smooth), it suffers from the distribution of sample points, and we cannot claim that it is automatic since finding the correct parameters for the search radius is usually a trial-and-error task. If the closest data points in each quadrant are used, then the method can be made automatic and continuous.

4.3.3 Linear interpolation in triangulation (TIN)

This method is popular for terrain modelling applications and is based on a triangulation of the data points. As is the case for the VD, a triangulation is a piecewise subdivision (tessellation) of the plane, and in the context of interpolation a linear function is assigned to each piece (each triangle). Interpolating at location x means first finding inside which triangle x lies, and then the height is estimated by linear interpolation on the 3D plane defined by the three vertices forming the triangle (the samples are lifted to their elevation value). The number of samples used in the interpolation is therefore always 3, and their weight is based on the barycentric value (see below). To obtain satisfactory results, this method is usually used in 2D with a Delaunay triangulation because, among all the possible triangulations of a set of points in the plane, it maximizes the minimum angle of each triangle.

The method is exact, continuous, local, adaptative, efficient, and automatic. Only the property #3 is not fulfilled (at the edges of the triangles).

If the point location strategy is used to identify the triangle containing x (Section 10), then $\mathcal{O}(n^{1/3})$ on average is used. The interpolation itself is performed in constant time.

Data-dependent triangulations. It was shown in Chapter 3 and in Figure 1.5 that, for terrain modelling, the Delaunay triangulation is preferred over other triangulations because it favours triangles that are as equilateral as possible. However, it should be noticed that the elevation of the vertices are not taken into account to obtain the DT, ie if we changed the elevation of the samples we would always get the same triangulation. One might therefore wonder whether the DT best approximates the morphology of a terrain.

A triangulation that considers the elevation (or any z coordinate) is called a *data-dependent triangulation*. The idea is to define a set of criteria (instead of the empty circumcircle). One example is trying to minimise the change in normals for the two incident triangles of an edge. While such methods will yield longer and skinnier triangles, these might better approximate the shape of the terrain for some specific cases. One drawback of these methods is that different criteria will be required for different cases, and that computing such triangulation can be computationally expensive. In practice, one would need to first compute the DT, and then take each edge (and the two incident triangles), and perform a local flip based on the elevation values; the final triangulation is obtained by optimisation the wished criterion.

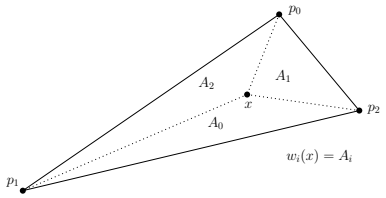


Figure 4.6: Barycentric coordinates. A_i defines the area of a triangle.

Barycentric coordinates. The linear interpolation in a triangle can be efficiently implemented by using barycentric coordinates, which are local coordinates defined within a triangle. Referring to Figure 4.6, any point x inside a triangle $p_1p_2p_3$ can be represented as a linear combination of the 3 vertices:

$$x = w_0p_0 + w_1p_1 + w_2p_2$$

and

$$w_0 + w_1 + w_2 = 1$$

The coefficients w_i are the barycentric coordinates of the point x with respect to the triangle $p_1p_2p_3$. Finding the coefficients w_0, w_1, and w_2 can be done by solving a system of linear equations. If we subtract p_2 from x, and we use $w_2 = 1 - w_0 - w_1$, we obtain

$$x - p_2 = w_0(p_0 - p_2) + w_1(p_1 - p_2)$$

We obtain 2 vectors ($p_0 - p_2$ and $p_1 - p_2$), which represent 2 edges of the triangle. This equation can be solved and we find that the 3 coefficients are equal of the area of the 3 triangle subdividing the original triangle (as shown in Figure 4.6).

Higher-order function in each triangle (TIN-c1). It is possible to modify the linear function inside each triangle by a higher-order function. As is the case for splines, there are *several* ways to achieve this, and the details of these is out of scope for this course. These methods are usually used more for finite element analysis where the flow of a certain fluid (eg wind) around or through a mechanical piece is studied.

Most methods would define a cubic Bézier polynomial inside each triangle (which is C^1), and then ensure that the function is C^1 along the edges and at the 3 vertices of the triangles. To achieve this the normals of each vertex is calculated by averaging the normals of the incident triangles, and the normal along an edge is computed similarly with the 2 incident triangles.

4.3.4 Natural Neighbour Interpolation (NNI)

This is a method based on the Voronoi diagram for both selecting the data points involved in the process, and assigning them a weight. It uses two VDs: one for the set S of data points (Figure 4.7), and another one where a point x is inserted at the estimation location (Figure 4.8). The insertion of x modifies *locally* a VD(S): the Voronoi cell $\mathcal{V}_x$ of x 'steals' some parts of some Voronoi cells of VD(S).

This idea forms the basis of natural neighbour coordinates, which define quantitatively the amount $\mathcal{V}_x$ steals from each of its natural neighbours (Figure **??**). Let $\mathcal{D}$ be the VD(S), and $\mathcal{D}^+ = \mathcal{D} \cup \{x\}$. The Voronoi cell of a point p in $\mathcal{D}$ is defined by $\mathcal{V}_p$, and $\mathcal{V}_p^+$ is its cell in $\mathcal{D}^+$. The natural neighbour coordinate of x with respect to a point p_i is

$$w_i(x) = \frac{Area(\mathcal{V}_{p_i} \cap \mathcal{V}_x^+)}{Area(\mathcal{V}_x^+)} \tag{4.3}$$

where $Area(\mathcal{V}_{p_i})$ represents the area of $\mathcal{V}_{p_i}$. For any x, the value of $w_i(x)$ will always be between 0 and 1: 0 when p_i is not a natural neighbour of x, and 1 when x is exactly at the same location as p_i. A further important consideration is that the sum of the areas stolen from each of the k natural neighbours is equal to $Area(V_x^+)$, in other words:

$$\sum_{i=1}^{k} w_i(x) = 1. \tag{4.4}$$

Therefore, the higher the value of $w_i(x)$ is, the stronger is the 'influence' of p_i on x. The natural neighbour coordinates are influenced by both the distance from x to p_i and the spatial distribution of the p_i around x.

Natural neighbour interpolation is based on the natural neighbour coordinates. The points used to estimate the value of an attribute at location x are the natural neighbours of x, and the weight of each neighbour is equal to the natural neighbour coordinate of x with respect to this neighbour.

The natural neighbour interpolant possesses all the wished properties from above, except that the first derivative is undefined at the data points. Its main disadvantage is that its implementation is rather complex, and obtaining an efficient one is not simple and involves complex manipulation of the VD. From Section 3.4 we know that one insertion of a single point p in a DT can be done in $\mathcal{O}(\log n)$, but the deletion of a point is a more complex operation (outside the scope of this book).

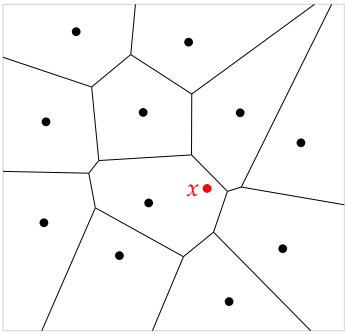


Figure 4.7: The VD of a set of points with an interpolation location x.

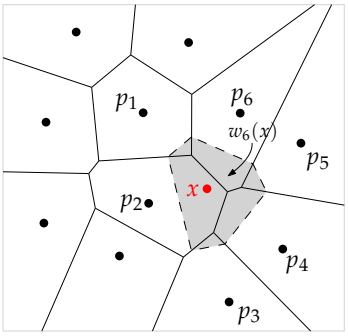
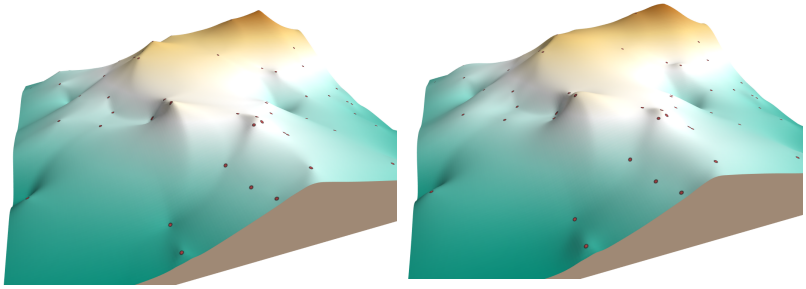


Figure 4.8: Natural neighbour coordinates in 2D for x. The shaded polygon is $\mathcal{V}_x^+$.



(a) NNI **(b)** NNI (C^1)

Figure 4.10: Notice how the **NNI** interpolant creates "inverted cups" around each sample point, and how **NNI-c1** results in a more rounded surface.

Higher-order function (NNI-c1). The NNI method can be thought of performing linear interpolation, in the 1D case (where we have one independent variable) then it is equivalent to a linear interpolant (see Figure 4.9).

It has been modified so that the first derivative is possible everywhere, including at the data points. This was achieved by modifying the weights so that they are not linear anymore. The gradient of the surface at each sample point is taken into account, ie for each data point we can estimate the slope (with a linear function, a plane) and modify the weights; how this is done is out of scope for this course. The resulting interpolant is C^1, and Figure 4.10

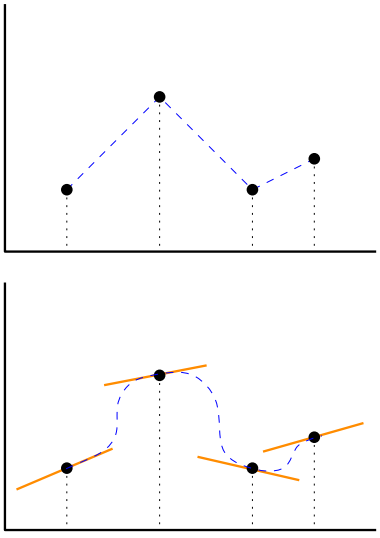


Figure 4.9: Top: The NNI interpolant in 1D is equivalent to a linear interpolation. **Bottom:** If the gradient at each sample points are calculated/estimated, then it is possible to modify the weights so that a C^1 interpolant is obtained.

4.3.5 Laplace interpolant

The Laplace interpolant, or non-Sibsonian interpolation, is a computationally faster variant of the natural neighbour interpolation method. It is faster because no (stolen) areas need to be computed, instead the lengths of the Delaunay and the Voronoi edges are used.

For a given interpolation location x, the natural neighbours p_i of x are used for the Laplace interpolant. The weight w_i of a p_i is obtained, as shown in the Figure 4.11, by:

$$w_i(x) = \frac{|edge_i(\mathcal{V}_x^+)|}{|xp_i|} \tag{4.5}$$

where $|edge_i(\mathcal{V}_x^+)|$ represents the length of the Voronoi edge between x and p_i (the orange edge in Figure 4.11 for one neighbour); and $|xp_i|$ the Euclidean distance (in 2D) between x and p_i (which is the Delaunay edge).

If we consider that each data point in S has an attribute a_i (its elevation), the interpolation function value at x is:

$$f(x) = \frac{\sum_{i=1}^{k} w_i(x)\, a_i}{\sum_{i=1}^{k} w_i(x)} \tag{4.6}$$

Note that the fraction becomes indeterminate when x equals one of the sample points p_i. In this case the Laplace interpolant therefore simply defines that $f(x) = a_i$.

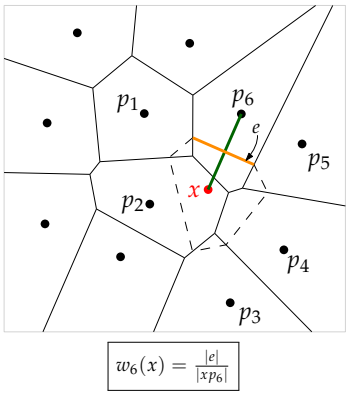


Figure 4.11: The weight for the Laplace interpolant for one neighbour.

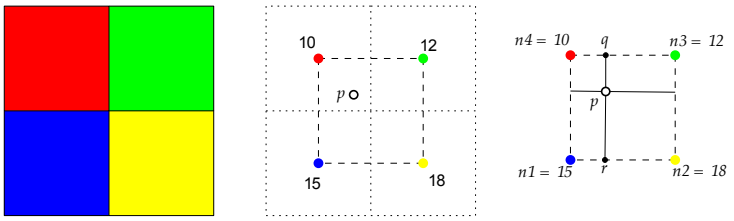


Figure 4.13: Bilinear interpolation.

Firstly the Laplace interpolant is exact: the interpolation method returns the exact value, rather than some estimate, of a sample point when it is queried at that precise location. Secondly, it is continuous and continuously differentiable (C^1) everywhere except at sites where finitely many Voronoi circles intersect. Thirdly, it is local, ie it uses only a local subset of data for the interpolation of a point. This limits the computational cost and supports efficient addition or removal of new data points. Finally, like the VD itself, it is adaptive to the spatial configuration of sample points. Unlike other methods such as IDW interpolation, the Laplace interpolant requires no user-defined parameters.

4.3.6 Bilinear Interpolation

When one wants to know the value of the elevation at a location p, she can simply look at the value of the pixel (which is equivalent to using nearest neighbour interpolation), but this method has many drawbacks, for example when one needs to *resample* a grid. Resampling means transforming an input grid so that the resolution and/or the orientation are different, see Figure 4.12.

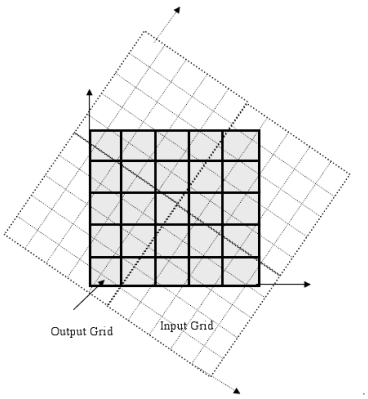


Figure 4.12: Resampling of an input grid, the output grid has different orientation.

Bilinear interpolation has been shown to give better results. The method, which can be seen as an "extension' of linear interpolation for raster data, performs linear interpolation in one dimension (say along the x axis), and then in the other dimension (y). Here one has to be careful about the meaning of a grid: does the value of a pixel represent the value of the whole pixel? or was the grid constructed by sampling the values at the middle of each pixel? In most cases, unless metadata are available, it is not known. But in the context of terrain modelling, we can assume that the value of a pixel represents the value at the centre of the pixel.

Suppose we have 4 adjacent pixels, each having an elevation, as in Figure 4.13. Bilinear interpolation uses the 4 centres to perform the interpolation at location $p = (p_x, p_y)$; it is thus a weighted-average method because the 4 samples are used, and their weight is based on the linear interpolation, as explained below. We need to linearly interpolation the values at locations q and r with linear interpolation, and then linearly interpolate along the y axis with these values. Also, notice that the result is independent of the order of interpolation: we could start with interpolating along the y axis and then the x axis and we would get the same result. For the case

in Figure 4.13, the calculation would go as follows:

$$q_z = \frac{p_x - n4_x}{n3_x - n4_x} \times (n3_z - n4_z) + n4_z$$

$$r_z = \frac{p_x - n1_x}{n2_x - n1_x} \times (n2_z - n1_z) + n1_z$$

$$p_z = \frac{p_y - r_y}{q_y - r_y} * (q_z - r_z) + r_z$$

4.4 Assessing the results of an interpolation method and/or fine-tuning the parameters

Finding the "best" interpolation method for a given dataset, and the most suitable parameters (if any are needed), can be a rather tricky task in practice because we most often do not have extra control points.

One simple technique, which is also very easy to implement, is called *jackknife*, or cross-validation. It is a simple statistics resampling technique to estimate the bias and the variance of an estimation.

Imagine you have a dataset S consisting of n sample points. The main idea is to remove/omit from S one sample point p and calculate the estimation $\hat{a}_p$ obtained for the elevation at the location (x, y) of p, and to compare this value with the real value of p (a_p). And then to repeat this for each of the n points in S; each estimation is thus obtained with $n - 1$ points.

One method (with given parameters) for a given dataset can be characterised by computing the root-mean-square error:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n} (\hat{z}_i - z_i)^2}{n}} \tag{4.7}$$

And it is a good idea to plot the results to observe where the largest differences between the estimation and the real values are obtained, this can help in idenfiying which parameters should be fine-tuned. See for instance one example in Figure 4.14. It can be seen in Figure 4.14c that the largest differences between the observed and estimated values are (mostly) concentrated around the two peaks of the terrain, which is not surprising. The differences in the lower areas (which is water) are smaller since these areas have a flatter morphology. Figure 4.14d shows the same absolute differences but in a scattered plot of the observed values versus the estimated one.

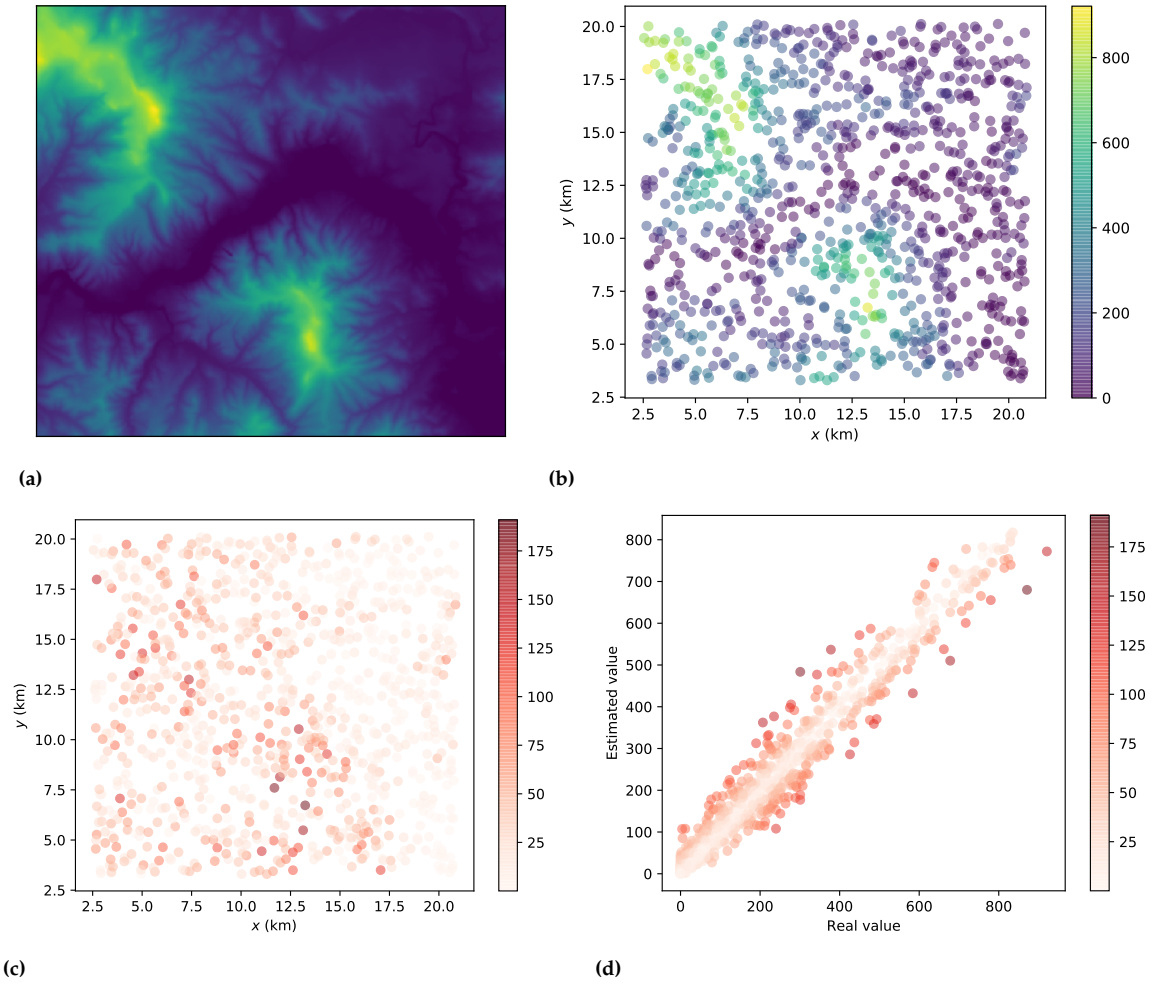
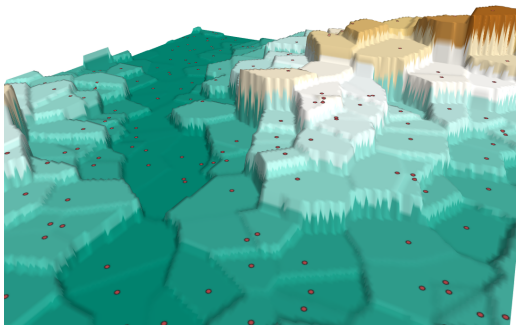


Figure 4.14: (a) A terrain of a given area containing 2 hills. **(b)** A sample of 1000 points of this terrain. **(c)** A plot of the errors (absolute values) obtained from the jackkknife (with IDW and a given search radius and power). **(d)** A plot of the absolute elevation versus the estimated ones .

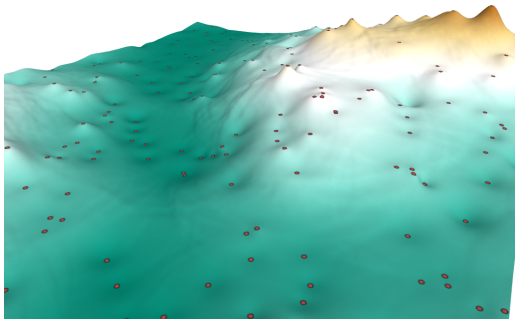
4.5 Overview of all methods

Figure 4.15 shows the result of 8 different interpolants for the same (real-world) sample points. You can better visualise these datasets, and download them, at `https://3d.bk.tudelft.nl/courses/geo1015/extra/interpol/`.

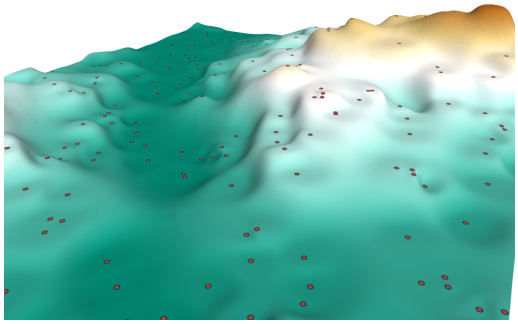
	exact	continuous	local	adaptable	efficient	automatic
global function	×	C^{2+}	×	–	–	×
splines	×	C^{2+}	depends	0	-	×
nearest neigh.	✓	×	✓	+	++	✓
IDW	✓	×	✓	-	0	×
TIN	✓	C^0	✓	+	++	✓
NNI	✓	C^0	✓	++	0	✓
NNI-c1	✓	C^1	✓	++	-	✓
Laplace	✓	C^0	✓	++	+	✓
bilinear	✓	C^0	✓	++	++	✓



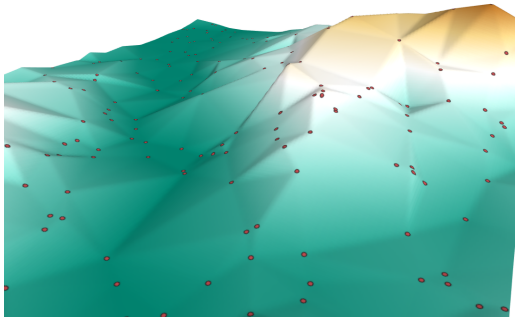
(a) Nearest neighour



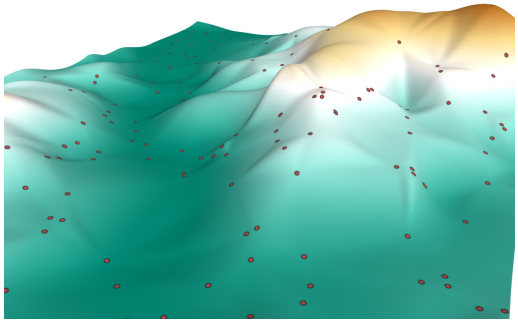
(b) IDW (radius=1500m; pow=2)



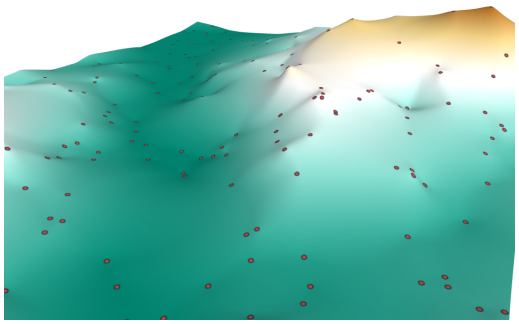
(c) IDW (radius=1500m; pow=4)



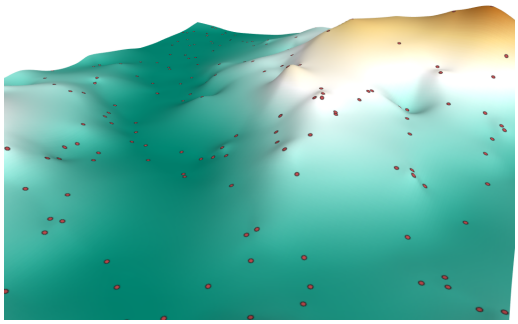
(d) TIN (linear)



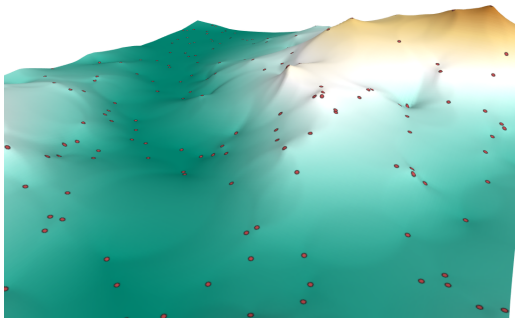
(e) TIN (C^1)



(f) Natural neighbours



(g) Natural neighbours (C^1)



(h) Laplace

Figure 4.15: Results of a few interpolation methods for the same dataset; the samples are shown on the surface (red dots).

4.6 Notes and comments

Watson (1992), in his authoritative book, lists the essential properties of an 'ideal' interpolation method for bivariate geoscientific datasets; we have added *computationally efficient* and *automatic* to the list.

Mitasova and Mitas (1993) gives a full description of the regularised splines with tension (RST) interpolation method. This method has also been implemented in the open-source GIS GRASS.

For a discussion about influence of the power in IDW on the resulting surface, please see Watson (1992).

The description of the barycentric coordinates is mostly taken from Eberly (2018).

The natural neighbour interpolation method is also called Sibson's interpolation, after the name of the inventor (Sibson, 1981).

An excellent summary of the methods to modify Equation **??** to obtain a continuous function is found in Flötotto (2003).

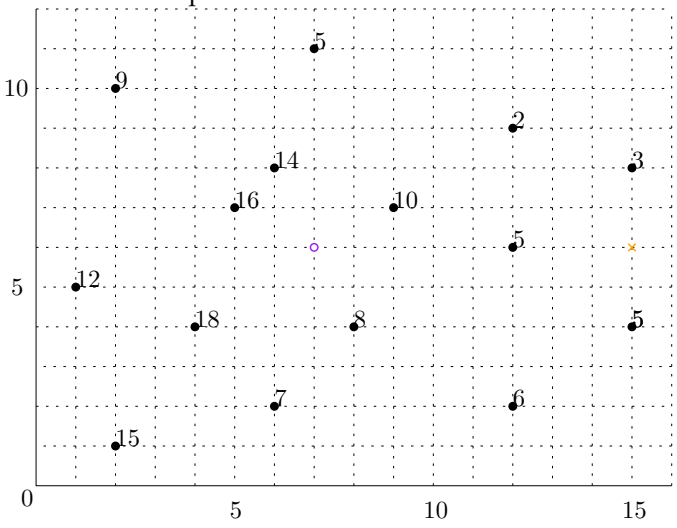
The construction of a polynomial inside each triangle of a TIN can be done with several methods. The simplest method is the Clough-Tocher method (Clough and Tocher, 1965; Farin, 1985). It splits each triangle into 3 sub-triangles (by inserting a temporary point at the centroid of the triangle) and a cubic function is built over each.

Dyn et al. (1990) shows how to obtain a data-dependent triangulation. Rippa (1990) proves that the DT is the triangulation that minimizes the roughness of the resulting terrain, no matter what the actual elevation of the data is. Here, roughness is defined as the integral of the square of the L^2-norm of the gradient of the terrain. Gudmundsson et al. (2002) shows that a variation of the DT (one where k vertices can be inside the circumcircle of a given triangle) can yield fewer local minima; whether it yields a "better' terrain is an open question.

4.7 Exercises

1. Given a triangle τ with coordinates (20.0, 72.0, 21.0), (116.0, 104.0, 32.0), and (84.0, 144.0, 26.0), estimate the elevation at x = (92.0, 112.0) with linear interpolation in the triangle (both by finding the equation of the plane and with barycentric coordinates).
2. What happens when the search distance is very large for inverse distance weighting interpolation (IDW)?
3. For grids, can IDW or others be used instead of bilinear? If yes, how does that work?
4. The 15 elevation samples below have been collected. You want to interpolate at two location:
   a) at location $(7, 6)$ with IDW (radius=3; power=2); the purple circle.
   b) at location $(15, 6)$ with linear interpolation in TIN; the orange cross.

   What are the respective answers?



Spatial interpolation: kriging | 5

Kriging is a spatial interpolation method that was developed mostly by Georges Matheron based on the earlier work of Danie Krige, who created it to estimate the gold yield of mines in South Africa. In contrast to most other spatial interpolation methods, it involves creating a custom statistical model for every dataset. In this way, different types of kriging can take into account the different characteristics that are specific to a dataset, such as highly unequal distributions of sample points, anisotropy (spatial correlation that varies according to a direction), or the varying uncertainty and spatial correlation of a set of sample points.

Like other geostatistical models, kriging is based on the fact that when one moves across space, values such as the gold content in rock or the elevation in a terrain have both a general spatial *trend* (eg a mean value, a fitted plane or a more complex polynomial) and a certain spatially correlated *randomness* (ie closer points tend to have more similar values). Both of these elements can be modelled in kriging.

trend

In the simplest case, when the trend is completely known, it is possible to use *simple kriging*. This is admittedly not very useful in practice, but we will nevertheless first cover it because it teaches the basics of the technique and is helpful to understand other types of kriging. Then, we will look at *ordinary kriging*, which attempts to estimate the trend as a constant and is the simplest case of kriging that is widely used in practice.

5.1 Statistical background

The physical processes that shape the world can be considered to be at least partly deterministic. In the case of a DTM, the elevation is determined by processes that we can model (more or less accurately), such as plate tectonics, volcanic activity and erosion. However, these processes are much too complex and not well-enough understood to use them to obtain accurate elevation values. Because of this, the value of sufficiently complex properties, such as the elevation of a terrain, can usually be defined as the result of *random processes*.

random process

Based on this inherent randomness, geostatistical models consider that the value of a property at a location is just one of the infinitely many values that are possible at that location. These possible values are not all equally likely, but they are instead represented by a *probability distribution*, which we can associate with a function (ie a *probability distribution function*) or with a set of standard statistical measures, such as the *mean* and *variance*. This situation is phrased in mathematical terms by saying that the value of the elevation property z at a location x is a *random variable* $Z(x)$. For the sake of simplicity, we will usually omit the location and denote it just as Z; or when working with multiple locations (eg x_i and x_j), we will

probability distribution

random variable

shorten their respective random variables using subscripts (eg Z_i and Z_j).

mean

variance

One way to express the general shape of the probability distribution of a random variable is in terms of its *mean* and its *variance*. In statistics, the mean, *expectation* or *expected value* of a random variable Z is a sort of probability-weighted average of its possible values and is denoted as $E[Z]$ or μ. Meanwhile, the *variance* of a random variable Z is a measure of how far the values of Z will usually spread from its mean, and it is denoted as $\text{var}(Z)$ or σ^2. A small variance thus means that a few random samples of Z will likely form a tight cluster around its mean, whereas a large variance will have sample values that are more spread out. Mathematically, the variance is defined as the expected value of the squared deviation from the expected value of Z, or:

$$\begin{aligned} \text{var}(Z) &= E\left[(Z - E[Z])^2\right] && (5.1)\\ &= E\left[Z^2 - 2ZE[Z] + E[Z]^2\right] \\ &= E\left[Z^2\right] - 2E[Z]E[Z] + E[Z]^2 \\ &= E\left[Z^2\right] - 2E[Z]^2 + E[Z]^2 \\ &= E\left[Z^2\right] - E[Z]^2. && (5.2) \end{aligned}$$

covariance

Next, it is important to define the covariance, denoted as $\text{cov}(Z_i, Z_j)$, or σ_{ij}, which expresses the joint variability of the random variables Z_i and Z_j. Thus, a positive covariance between Z_i and Z_j means that when one increases or decreases, the other is expected to increase/decrease in the same direction. Conversely, a negative covariance means that the variables tend to increase/decrease in opposite directions. The magnitude of the covariance is related to the magnitude of this increase or decrease. It is thus defined mathematically as the expected product of their deviations from their (individual) expected values, or:

$$\text{cov}(Z_i, Z_j) = E\left[(Z_i - E[Z_i])\left(Z_j - E[Z_j]\right)\right]. \quad (5.3)$$

Here, it is good to note that the covariance of Z_i with itself is equivalent to its variance:

$$\text{cov}(Z_i, Z_i) = E\left[(Z_i - E[Z_i])^2\right] = \text{var}(Z_i).$$

correlation coefficient

While not used further in this lesson, it is also good to know that the variance and the covariance can be used to calculate the Pearson correlation coefficient ρ_{ij}, which is one of the most common statistical measures that is applied to datasets:

$$\rho_{ij} = \frac{\text{cov}(Z_i, Z_j)}{\sqrt{\text{var}(Z_i)\text{var}(Z_j)}}.$$

Note that this is essentially just a normalised form of the covariance.

5.2 The standard geostatistical model

Geostatistics considers that a random variable Z, which represents a spatially correlated property at a given location, can be decomposed into two related variables: (i) a non-random spatial trend that can be modelled by the *expectation* $E[Z]$ (eg using a constant, a polynomial, a spline, etc.); and (ii) a random but spatially correlated deviation from this trend that is considered as a sort of error or residual term and is here denoted as R. In the case of elevation, the former would represent the general shape of the terrain, whereas the latter would represent local differences from it. We therefore have:

$$Z = E[Z] + R. \tag{5.4}$$

It is important to consider a couple important aspects here. First, note that since $R = Z - E[Z]$, the variance (Equation 5.1) and covariance (Equation 5.3) can also be defined in terms of the residuals:

$$\text{var}(Z) = E\left[R^2\right], \tag{5.5}$$
$$\text{cov}(Z_i, Z_j) = E\left[R_i \cdot R_j\right]. \tag{5.6}$$

Also, it is important to know that in order not to introduce any bias, the expected value of the residual must be zero. That is, $E[R] = 0$.

5.3 Simple kriging

Simple kriging starts from the assumption that in the geostatistical model from Equation 5.4, the expectation $E(Z)$ is the same everywhere, which is known as the *stationarity of the mean*, and that it is *known*. In the case of a DTM, that would mean that a terrain might be uneven with significant peaks and valleys, but that there is not a general trend across it (eg a clear slope with higher elevations on one side and lower elevations on the opposite side). Mathematically, we can express that as:

stationarity of the mean

$$E[Z(x+h)] = E[Z(x)], \tag{5.7}$$

where x is an arbitrary point in the domain (ie the area we want to interpolate), h is any vector from x to another point in the domain and $Z(x)$ is the value of a random variable at x (eg its elevation).

Next, simple kriging also makes the assumption that the covariance between a random variable at a pair of locations does not depend on the locations, but instead can be defined based only on the vector separating them. In the case of a DTM, this would mean that the likelihood of finding similar elevations at two points separated by a given distance and orientation does not change across the terrain. For instance, a terrain that goes from smooth on one side to rough on the other would not satisfy this assumption. Mathematically, we can express this as:

$$\operatorname{cov}(Z(x+h), Z(x)) = C(h), \tag{5.8}$$

where C is the covariance function. Since both the expectation (Equation 5.7) and the covariance (Equation 5.8) are translation invariant, this pair of assumptions are together known as *second-order stationarity*.

second-order stationarity

Simple kriging is similar to the other spatial interpolation methods that use a weighted average. However, since we are assuming that the expectation is the same everywhere, we will define it as a weighted average of residuals R of the form $Z - E[Z]$, after which the expectation needs to be added back in order to obtain the final value. It thus defines a function $\hat{R}_0$ that estimates the value of the residual R of the random variable Z at a location x_0 as a weighted average of its residuals at the n sample points x_i that we will use for the interpolation (where $1 \leq i \leq n$). We denote this as:

$$\hat{R}_0 = \hat{Z}_0 - E[Z_0] = \sum_{i=1}^{n} w_i \underbrace{(Z_i - E[Z_i])}_{R_i}. \tag{5.9}$$

Kriging has two distinguishing characteristics. First, that it is *unbiased*. This means that it creates a model where the expected value of the estimation at a location x_0 is equal to the expected value at that location. In mathematical terms, we can formulate this as:

unbiased

$$E\left[\hat{Z}_0 - Z_0\right] = 0 \qquad \text{or} \qquad E\left[Z_0\right] = E\left[\hat{Z}_0\right]. \tag{5.10}$$

In order to check this for simple kriging, we can put the weighted average from Equation 5.9 in this equation, which results in the following:

$$\begin{aligned} E\left[Z_0\right] &= E\left[E[Z_0] + \sum_{i=1}^{n} w_i R_i\right] \\ &= E[Z_0] + \sum_{i=1}^{n} w_i 0 E[R_i] \\ &= E[Z_0]. \end{aligned}$$

The second property of kriging is that it *minimises the variance of the estimation error*, which in this case is given by $\operatorname{var}\left(\hat{R}_0 - R_0\right)$. If we use the definition of the variance from Equation 5.1, this can be instead put in terms of an expectation:

minimisation of the variance

$$\operatorname{var}\left(\hat{R}_0 - R_0\right) = E\left[\left(\left(\hat{R}_0 - R_0\right) - E\left[\hat{R}_0 - R_0\right]\right)^2\right]$$

However, we know from the unbiased criterion from Equation 5.10 that $E\left[\hat{R}_0 - R_0\right] = 0$, and so we can simplify the previous equation as:

$$\operatorname{var}\left(\hat{R}_0 - R_0\right) = E\left[\left(\hat{R}_0 - R_0\right)^2\right].$$

If this is expanded, it results in:

$$\begin{aligned}\operatorname{var}\left(\hat{R}_0 - R_0\right) &= E\left[{\hat{R}_0}^2 - 2\hat{R}_0 R_0 + {R_0}^2\right] \\ &= E\left[{\hat{R}_0}^2\right] - 2E\left[\hat{R}_0 R_0\right] + E\left[{R_0}^2\right] \\ &= E\left[\sum_{i=1}^{n}\sum_{j=1}^{n} w_i w_j R_i R_j\right] - 2E\left[\sum_{i=1}^{n} w_i R_i R_0\right] + E\left[{R_0}^2\right] \\ &= \sum_{i=1}^{n}\sum_{j=1}^{n} w_i w_j E\left[R_i R_j\right] - 2\sum_{i=1}^{n} w_i E\left[R_i R_0\right] + E\left[{R_0}^2\right].\end{aligned}$$

Here, we can use the definitions of the variance based on residuals from Equations 5.5 and 5.6 together with our covariance formula from Equation 5.8, which yields:

$$\operatorname{var}\left(\hat{R}_0 - R_0\right) = \sum_{i=1}^{n}\sum_{j=1}^{n} w_i w_j \operatorname{cov}(R_i, R_j) - 2\sum_{i=1}^{n} w_i \operatorname{cov}(R_i, R_0) + \operatorname{cov}(R_0, R_0) \tag{5.11}$$

$$= \sum_{i=1}^{n}\sum_{j=1}^{n} w_i w_j C(x_i - x_j) - 2\sum_{i=1}^{n} w_i C(x_i - x_0) + C(x_0 - x_0). \tag{5.12}$$

In order to minimise this equation, we can find where its first derivative is zero. This is:

$$\frac{\partial \operatorname{var}\left(\hat{R}_0 - R_0\right)}{\partial w_i} = 2\sum_{j=1}^{n} w_j C(x_i - x_j) - 2C(x_i - x_0) = 0 \qquad \text{for all } 1 \leq i \leq n,$$

which yields the set of n simple kriging equations:

$$2\sum_{j=1}^{n} w_j C(x_i - x_j) = 2C(x_i - x_0). \tag{5.13}$$

While these equations can be used to perform simple kriging, it is often easier to deal with these in matrix form:

$$\underbrace{\begin{pmatrix} C(x_1 - x_1) & \cdots & C(x_1 - x_n) \\ \vdots & \ddots & \vdots \\ C(x_n - x_1) & \cdots & C(x_n - x_n) \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} w_1 \\ \vdots \\ w_n \end{pmatrix}}_{w} = \underbrace{\begin{pmatrix} C(x_1 - x_0) \\ \vdots \\ C(x_n - x_0) \end{pmatrix}}_{d} \tag{5.14}$$

which is known as the *simple kriging system*. Finally, if we invert the

simple kriging system

matrix A, the interpolation weights are given by:

$$w = A^{-1}d. \tag{5.15}$$

Now, the obvious remaining questions are: (i) what expectation $E[Z]$ to use, and (ii) what covariance function C to use. Without an external evaluation of the dataset, there are no optimal answers for either of those questions, which is the main weakness of simple kriging and the reason why it is not used widely in practice. That being said, a reasonable solution could be to use the average value of z_i for all points as $E[Z]$, and an arbitrary covariance function, such as the exponential, gaussian and spherical functions that we discuss in the following section.

5.4 The variogram

As we saw in simple kriging, the theoretical definitions of the variance and covariance functions imply that we know the expected value (ie $E[Z]$ in Equation 5.1, and $E[Z_i]$ and $E[Z_j]$ in Equation 5.3), which in practice is not realistic. In order to avoid this problem, most other forms of kriging rely instead on what is known as a *variogram*. The variogram $\gamma(h)$ is a function that expresses the average dissimilarity of the value of a random variable Z between sample points at different distances. It is defined as:

$$\gamma(h) = \frac{1}{2}(Z(x+h) - Z(x))^2, \tag{5.16}$$

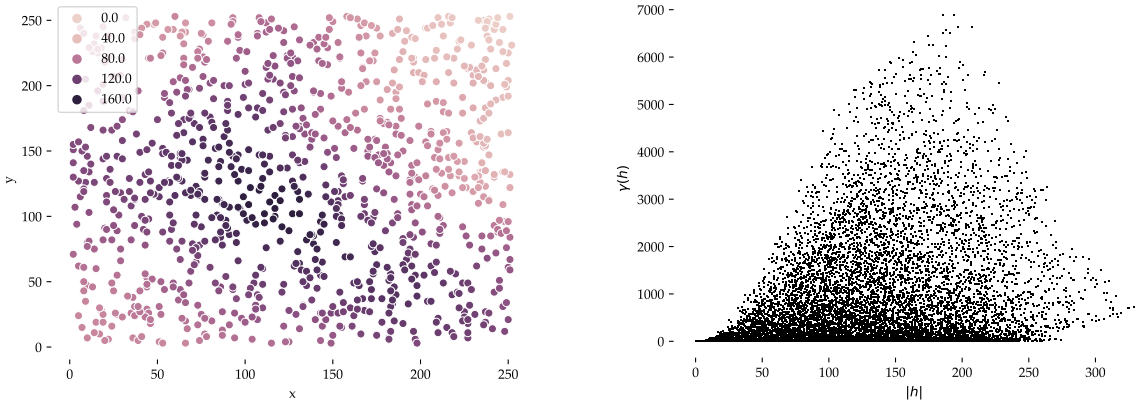
where x is a sample point, h is a vector from x to another sample point and $Z(x)$ is the value of a random variable at x (eg its elevation).

When this is done with every possible pair of sample points in a dataset, or with a representative subset in order to speed up the process as it is usually done in practice, $|h|$ (ie the magnitude of the vector h) and $\gamma(h)$ can be put into a scatter plot to show how the average dissimilarity of a value changes with the distance between the sample points. The result of such a plot is what is known as a *variogram cloud* (Figure 5.1).

variogram cloud

In this figure, it is possible to see some typical characteristics of a variogram cloud. Since nearby sample points tend to have similar values, the dissimilarity tends to increase as the distance between sample points increases. However, it is worth noting that since the farthest away pairs of sample points have similar values in this specific dataset, the dissimilarity also decreases at the highest distances.

Since most of the time there is a wide variation between the dissimilarities shown at all distances in a variogram cloud, the next step is to average the dissimilarity of the pairs of sample points based on distance intervals. Mathematically, a series of averages of dissimilarities $\gamma^\star(h)$ can be created by computing the average dissimilarities for all vectors whose lengths are within a series of specified intervals (generally known as *bins*). Given



(a) dataset

(b) variogram cloud

Figure 5.1: Starting from (a) a sample dataset, (b) the variogram cloud can be computed. In this case, only 1% randomly selected point pairs were used.

a set $\mathfrak{h}$ containing the vectors for a length interval, the average for its dissimilarity class is computed as:

$$\gamma^\star(\mathfrak{h}) = \frac{1}{2n} \sum (z(x+h) - z(x))^2 \qquad \text{for all } h \in \mathfrak{h} \tag{5.17}$$

where n is the number of sample point pairs in $\mathfrak{h}$.

This computation results in much smoother values for the dissimilarity, and when the results of $|h|$ and $\gamma^\star(h)$ are put into a scatter plot (Figure 5.2), the result is what is known as an *experimental variogram*. Experimental variograms are based on a few parameters (Figure 5.2b illustrates these):

experimental variogram

- the *sill*, which is the upper bound of $\gamma^\star(h)$; — sill
- the *range*, which is the value of $|h|$ when it converges; — range
- the *nugget*, which is the value of $\gamma^\star(h)$ when $|h| = 0$. — nugget

Note that in order to avoid the unreliable dissimilarities that are common at large distances between sample points, it is usual practice to only compute the experimental variogram for distances up to half of the size of the region covered by the dataset.

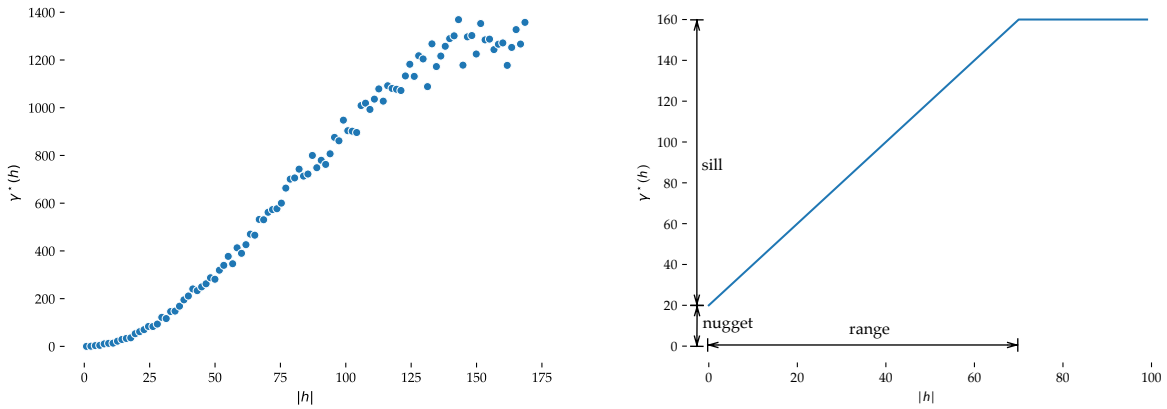
Finally, the last step is to replace the experimental variogram with a *theoretical variogram* function that approximates it and which can be more easily evaluated for further calculations. Depending on the shape of the variogram, there are various functions that can be used. Some examples are:

theoretical variogram

$$\gamma_{\text{exponential}}(h) = s\left(1 - e^{\frac{-3|h|}{r}}\right) + n \tag{5.18}$$

$$\gamma_{\text{gaussian}}(h) = s\left(1 - e^{\frac{-(3|h|)^2}{r^2}}\right) + n \tag{5.19}$$

$$\gamma_{\text{spherical}}(h) = \begin{cases} s\left(\frac{3|h|}{2r} - \frac{|h|^3}{2r^3}\right) + n & \text{if } |h| \le r \\ s + n & \text{if } |h| > r \end{cases} \tag{5.20}$$



(a) experimental variogram

(b) parameters

Figure 5.2: (a) The experimental variogram is usually described in terms of (b) its parameters.

where s is the *sill*, set to roughly the value of $\gamma^\star(h)$ when $\gamma^\star(h)$ is flat; r is the *range*, roughly the minimum value of $|h|$ where $\gamma^\star(h)$ is flat, and n is the nugget, which is the starting value of $\gamma^\star(h)$. Figure 5.3 shows the result of fitting the three example theoretical variogram functions, exponential, gaussian and spherical. Note how the gaussian function appears to be a better fit in this case.

These functions can be used as covariance functions for simple kriging, taking into account that $\gamma(h) = C(0) - C(h)$. Note that this means that the covariance is high when $|h|$ is small and it decreases as $|h|$ increases.

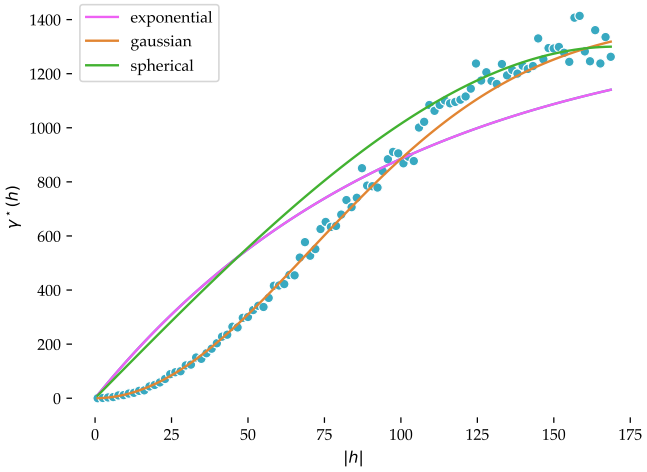


Figure 5.3: Three possible theoretical variogram functions

5.5 Ordinary kriging

Ordinary kriging is similar to simple kriging and to other spatial interpolation methods that use a weighted average (see Equation 5.9). It thus defines a function $\hat{Z}_0$ that estimates the value of the random variable Z at a location x_0 as a weighted average of its value at the n sample points x_i that we will use for the interpolation (where $1 \leq i \leq n$). We denote this as:

$$\hat{Z}_0 = \sum_{i=1}^{n} w_i Z_i. \tag{5.21}$$

Like simple kriging, ordinary kriging is *unbiased*. This means that it creates a model where the expected value of the estimation at a location x_0 is equal to the expected value at that location. In practice, this means that if we put the weighted average from the previous equation in Equation 5.10, it results in the following:

$$E\left[Z_0\right] = E\left[\sum_{i=1}^{n} w_i Z_i\right] = \sum_{i=1}^{n} w_i E\left[Z_i\right]. \tag{5.22}$$

Here, *ordinary kriging* also makes the assumption that the expectation is the same everywhere (stationarity of the mean). Therefore, $E\left[Z_i\right]$ has the same value everywhere (ie it is a constant) and we can thus move it outside of the summation:

$$E\left[Z_0\right] = E\left[Z_i\right] \sum_{i=1}^{n} w_i, \tag{5.23}$$

and since also $E\left[Z_i\right] = E\left[Z_0\right]$ (because of the stationarity of the mean), the two terms cancel each other out in the previous equation, which

means that the unbiased property in ordinary kriging implies that the interpolation weights must add up to one:

$$\sum_{i=1}^{n} w_i = 1. \tag{5.24}$$

Fulfilling this criterion means that we can use the variogram in ordinary kriging, which is not true for simple kriging.

Like simple kriging, ordinary kriging *minimises the variance of the estimation error*, which is given by $\text{var}\left(\hat{Z}_0 - Z_0\right)$. For this, we can use the same derivation as for simple kriging up to Equation 5.11 but using the variogram for the final step. This is:

$$\begin{aligned}\text{var}\left(\hat{R}_0 - R_0\right) &= \sum_{i=1}^{n}\sum_{j=1}^{n} w_i w_j \text{cov}(R_i, R_j) - 2\sum_{i=1}^{n} w_i \text{cov}(R_i, R_0) + \text{cov}(R_0, R_0) \\ &= -\sum_{i=1}^{n}\sum_{j=1}^{n} w_i w_j \gamma(x_i - x_j) + 2\sum_{i=1}^{n} w_i \gamma(x_i - x_0) - \gamma(x_0 - x_0).\end{aligned} \tag{5.25}$$

Using the previous equation and the unbiased criterion from Equation 5.10, we can apply the minimisation method known as Lagrange multipliers* and arrive at the set of $n + 1$ ordinary kriging equations:

$$\begin{aligned}\sum_{j=1}^{n} w_i \gamma(x_i - x_j) + \mu(x_0) &= \gamma(x_i - x_0) \qquad \text{for all } 1 \leq i \leq n \\ \sum_{j=1}^{n} w_i &= 1\end{aligned} \tag{5.26}$$

where $\mu(x_0)$ is a Lagrange parameter that was used in the minimisation process.

Like with simple kriging, these equations can be used to perform ordinary kriging, but it is often easier to deal with these in matrix form:

$$\underbrace{\begin{pmatrix} \gamma(x_1 - x_1) & \cdots & \gamma(x_1 - x_n) & 1 \\ \vdots & \ddots & \vdots & 1 \\ \gamma(x_n - x_1) & \cdots & \gamma(x_n - x_n) & 1 \\ 1 & \cdots & 1 & 0 \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} w_1 \\ \vdots \\ w_n \\ \mu(x_0) \end{pmatrix}}_{w} = \underbrace{\begin{pmatrix} \gamma(x_1 - x_0) \\ \vdots \\ \gamma(x_n - x_0) \\ 1 \end{pmatrix}}_{d} \tag{5.27}$$

which is known as the *ordinary kriging system*.

* https://en.wikipedia.org/wiki/Lagrange_multiplier

Finally, if we invert the matrix A, the weights and the Lagrange multipliers are given by:

$$w = A^{-1}d \tag{5.28}$$

5.6 Implementation

Kriging can be directly applied to any point on the plane, yielding a result such as the one in Figure 5.4. However, much like other interpolation methods, kriging is only reliable in the domain (ie roughly the convex hull of the points). It can extrapolate (often with negative weights), but that does not mean that the results outside the domain are accurate.

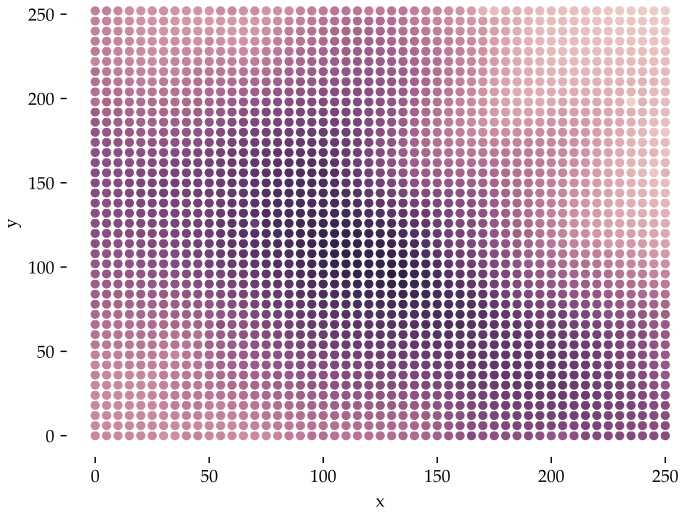


Figure 5.4: The result of using ordinary kriging to interpolate on a grid of points using the sample dataset using only the sample points within 15 units of each interpolated point.

Finally, it is also important to consider some computational aspects into account. If a large number of sample points are used to interpolate every point, kriging can be *very slow*. The reason for this is because matrix A will be very large, and inverting a matrix is a computationally expensive process. However, inverting very large matrices is not really needed in practice. When a sample point is far from the interpolated point, its weight will be low, and it will thus have only a small influence on it. Because of this, it is usually best to only take into account a few sample points in the calculation, either by limiting the sample points to those within a given search radius, or by selecting only a given number of its closest sample points. However, you should note that this can cause artefacts in the final result.

5.7 Notes and comments

Krige (1951) is the original publication by Danie Krige, which was later formalised by Georges Matheron (Matheron, 1962; Matheron, 1965). How this came to be is best explained in Cressie (1993).

If you have trouble following the derivations of the kriging equations or want to know more about them, Lichtenstern (2013) explains this well. If you feel like your statistics background is a bit weak, you first might want to have a look at Fewster (2014), particularly Chapter 3.

A relatively simple explanation of Kriging with agricultural examples that is accessible from the campus or VPN is given by Oliver and Webster (2015). A standard reference textbook that is good but not so easy to follow is Wackernagel (2003).

Two other good Youtube videos that explain kriging:

- `https://www.youtube.com/watch?v=CVkmuwF8cJ8`
- `https://www.youtube.com/watch?v=98zz25kTteQ`

5.8 Exercises

1. Why can using a search radius create artifacts in the interpolated terrain?
2. If kriging generally provides better results than other interpolation methods, why would you use something else (eg IDW)?
3. What does a nugget of zero say about a dataset? What about a large nugget?
4. What kind of dataset would yield a flat variogram (ie a horizontal line)?

Topographic features 6

While a DTM is a (2.5D) surface, it can also be conceptualised as an aggregation of many *topographic features* that are inter-related. Common examples of features are peaks, ridges, valleys, lakes, cliffs, etc., but one can think of application-specific ones such as the navigational channels in bathymetry, buildings in city modelling, or dikes for flood modelling.

Identifying the different features forming a terrain enhances our understanding of the raw dataset. To help us extract and identify features, some operations/characteristics need to be extracted from terrains, eg the slope, the aspect, the curvature, the roughness, etc.

We describe in this chapter a few key features in terrains, and describe how they can be extracted. Since these differ from the data model used (TINs vs grids), we give examples for both.

6.1 Slope

The slope at a given location p on a terrain is defined by the plane H that is tangent at p to the surface representing the terrain (see Figure 6.1). What we casually refer to as 'slope' has actually two components: (1) gradient; (2) aspect (see Figure 6.2).

Gradient. The gradient at a given point p is the maximum rate of change in elevation. It is obtained by the angle α between H and the horizontal plane (Figure 6.2). From a mathematical point-of-view, the gradient is the maximum value of the derivative at a point on the surface of the terrain (maximised over the direction). The gradient will often be expressed in degrees, or in percentage.

Notice that if we calculate the gradient at every location for a terrain, then we obtain a new field since the gradient is a continuous phenomena (values from 0% to 100% for instance). This means in practice that for a given terrain in raster, calculating its gradient will create a new raster file that can be further processed.

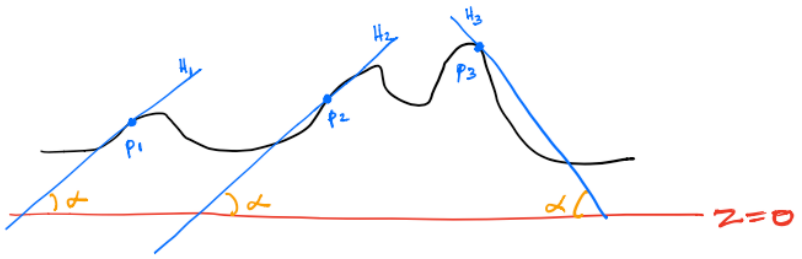


Figure 6.1: The slope at a given location p_i is defined by the tangent plane H_i to the surface. Here are 3 examples for a profile view of a terrain.

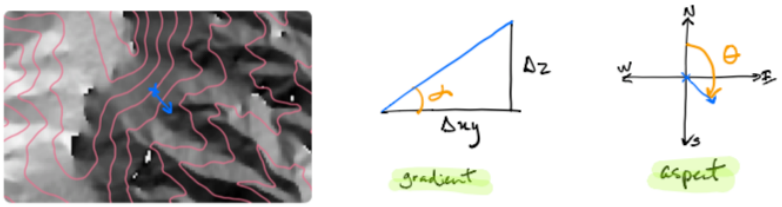


Figure 6.2: One DTM with contour lines, and the gradient and aspects concepts for a given location (blue cross).

Aspect At a given point p on the terrain the gradient can be in any direction, the aspect is this direction projected to the xy-plane. It is basically a 2D vector telling us the direction of the steepest slope at a given point. It is usually expressed in degrees from an arbitrary direction (most often the north). Observe that for the parts of the terrain that are horizontal (eg a lake) the value of the aspect is unknown. Also observe that at a given location the aspect will always be perpendicular to the contour line.

6.1.1 Slope in TINs

Calculating the slope in a TIN is fairly straightforward: for a point $p = (x, y)$ find the triangle τ containing this point, and compute the normal vector $\vec{n}$ of τ (pointing outwards). The projection of $\vec{n}$ on the xy-plane is the aspect (this is done by simply ignore the z-component of the vector). And the gradient is obtained easily.

If p is directly on a edge of the TIN then the solution cannot be obtained directly; it is common practice to calculate the normal vector of the 2 incident triangle and average them to obtain one $\vec{n}$. The same is applied if p is directly on a vertex v of the TIN: the average of all the normal vectors of all the incident triangle to v is used.

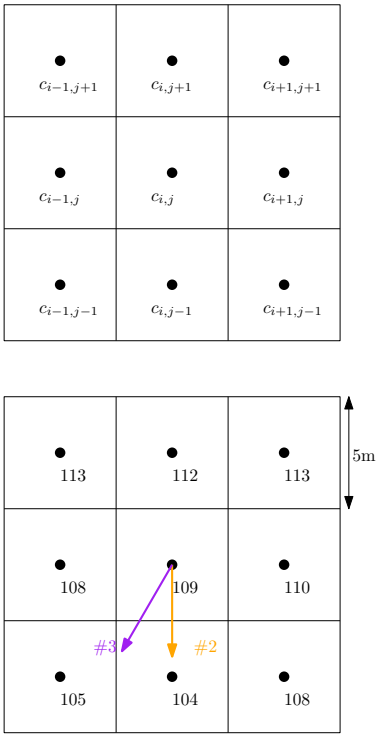


Figure 6.3: (top) Given a cell $c_{i,j}$, the 3×3 kernel and its 8 neighbours. **(bottom)** A hypothetical case with some elevations; orange = aspect for method #2 below, purple = aspect for method #3 below.

6.1.2 Slope in grids

If the terrain is represented as a regular grid (say of resolution r), then there exist several algorithm to obtain the slope at a given cell $c_{i,j}$. We list here a few common ones. It should be noticed that most algorithms use a 3×3 kernel, ie the value for the gradient/aspect at cell $c_{i,j}$ is computed by using (a subset of) the 8 neighbours.

1. Local triangulation + TIN method. It is possible to locally triangulate the 9 points, calculate the normal of the 8 triangles, and then use the method above for TINs.

2. Maximum height difference. This method simply picks the maximum height difference between $c_{i,j}$ and each of its 8 neighbours, the maximum absolute value is the direction of the aspect and the gradient can be trivially calculated. Notice that this means that there are only 8 possibilities for the slope (at 45° intervals). For the case in Figure 6.3, the aspect would be facing south (180°) and the gradient would be 45°.

3. Finite difference. With this method, the height differences in the x-direction (west-east) and in the y-direction (south-north) are calculated separately, and then the 2 differences are combined to obtain the slope. This means that only the direct 4-neighbours of $c_{i,j}$ are used.

$$\frac{\partial z}{\partial x} = \frac{z_{i+1,j} - z_{i-1,j}}{2\,r}, \frac{\partial z}{\partial y} = \frac{z_{i,j+1} - z_{i,j-1}}{2\,r}$$

The gradient is defined as:

$$\tan \alpha = \sqrt{(\frac{\partial z}{\partial x})^2 + (\frac{\partial z}{\partial y})^2}$$

and the aspect as:

$$\tan \theta = \frac{\frac{\partial z}{\partial y}}{\frac{\partial z}{\partial x}}$$

For the case in Figure 6.3, the aspect would be 194.0° and the gradient would be 39.5°.

4. Local polynomial fitting. Based on the 9 elevation points, it is possible to fit a polynomial (as explained in Chapter 4) that approximate the surface locally; notice that the polynomial might not pass through the point if a low-degree function is used.

A quadratic polynomial could for instance be defined:

$$f(x, y) = ax^2 + by^2 + cxy + dx + ey + d$$

, and thus:

$$\frac{\partial f}{\partial x} = 2ax + cy + d$$
$$\frac{\partial f}{\partial y} = 2by + cx + e$$

and if a local coordinate system centered at $c_{i,j}$ is used, then $x = y = 0$, and thus $\frac{\partial f}{\partial x} = d$ and $\frac{\partial f}{\partial y} = e$.

⚙ How does it work in practice?

The GDAL utility `gdaldem` (https://www.gdal.org/gdaldem.html) does not have the best documentation and does not explicitly mention which method is used.

After some searching, we can conclude that the method "4. Local polynomial fitting" is used by default for slope/aspect, and specificially the Horn's method is used (Horn, 1981). This uses a 3×3 window, and fits a polynomial; the centre pixel value is not used.

If the option `-alg ZevenbergenThorne` is used, then the algorithm of Zevenbergen and Thorne (1987) is used. This uses only the 4 neighbours, and is a variation of the method "3. Finite difference" above.

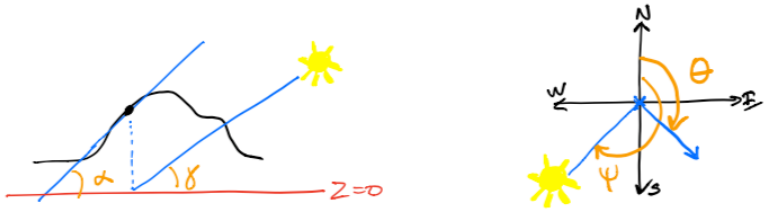


Figure 6.5: The 4 parameters necessary to calculate the hillshade at a location (black point on the terrain).

The documentation of `gdaldem` states that: "literature suggests Zevenbergen & Thorne to be more suited to smooth landscapes, whereas Horn's formula to perform better on rougher terrain."

6.1.3 Hillshading

Hillshading is a technique used to help visualise the relief of a DTM (see Figure 6.4 for an example). It involves creating an image that depicts the relative slopes and highlights features such as ridges and valleys; a hillshade does not depict absolute elevation. This image assumes that the source of light (the sun) is located at a given position.

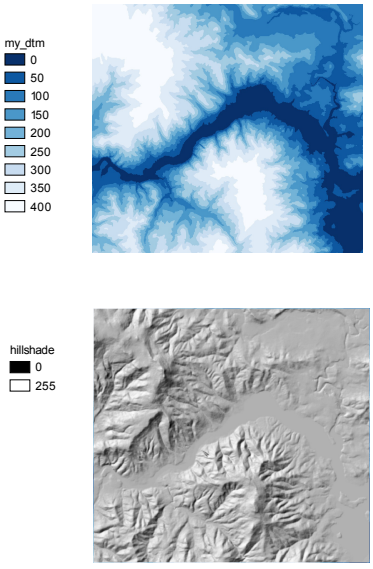


Figure 6.4: Top: a DTM visualised with height as a shade of blue. **Bottom**: when hillshading is applied.

While it would be possible to use advanced computer graphics methods (see Chapter 11) to compute the shadows created by the terrain surface, in practice most GIS implements a simplified version of it which can be computed very quickly.

Given a regular gridded DTM, hillshading means that each cell gets a value which depicts the variation in tone, from light to dark. The output of a hillshade operation is thus a regular gridded DTM, usually with the same extent and resolution as the original gridded DTM (for convenience). The values computed for each cell need as input the gradient and the aspect of the DTM. The formula to compute the hillshade of a given cell $c_{i,j}$ differs from software to software, and we present here one (it is used in ArcGIS for example, and surely others). It assumes that the output hillshade value is an integer in the range $[0, 255]$ (8-bit pixel), and that the direction (azimuth) and the height (given as an angle) of the illumination source is known. Notice that the position of the sun is relative to the cell, its position thus changes for different cells of a DTM. As above and in Figure 6.5, for a cell $cell_{i,j}$, its gradient is $\alpha_{i,j}$, its aspect is $\theta_{i,j}$, the azimuth of the sun is ψ (angle clockwise from the north, like the aspect), and the height of the sun is γ (0 rand is the horizon, π rand is the zenith).

$$hillshade_{i,j} = 250 \times ((\cos\gamma \times \cos\theta_{i,j}) + (\sin\gamma \times \sin\theta_{i,j} \times \cos(\psi - \alpha_{i,j})))$$

(all angles need to be radians)

6.2 Features

6.2.1 Characteristic points on terrains

Peak. A point p whose surrounding is formed only of points that are of lower elevation is a peak. The size and shape of the surrounding is

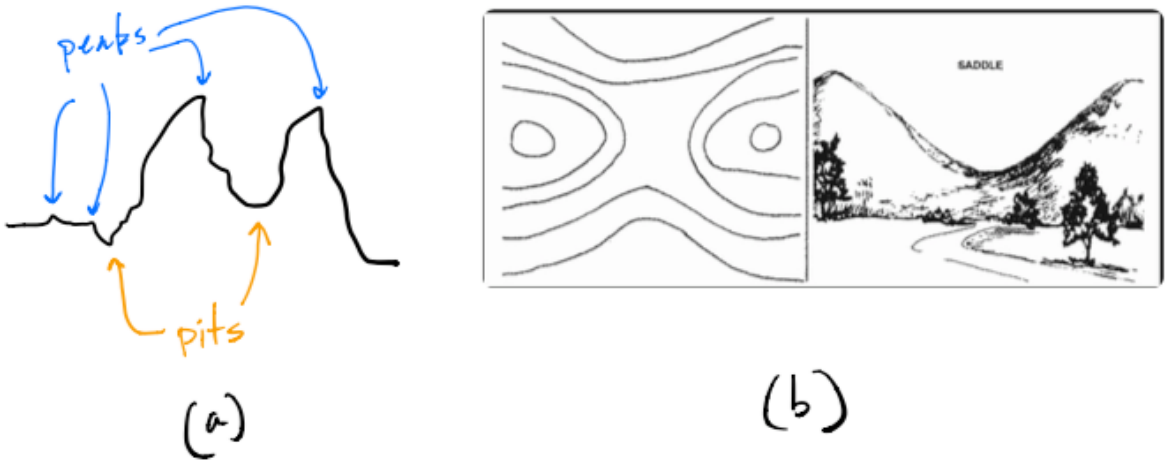


Figure 6.6: (a) Peaks and pits. **(b)** A saddle point (Figure from https://www.armystudyguide.com)

dependent on the application and on the data model used to represent the terrain. If a grid is used, this surrounding could be the 8 neighbours; if a TIN is used they could be the vertices that of the triangles incident to p. Observe that a peak can be local, that is one point that happens to be a few centimetres higher than all its neighbours would be classified as a peak (the small terrain in Figure 6.6 contains several peaks), while if we consider a hill we would surely consider only the top as the peak. A peak is therefore on the scale of the data.

The contour line through the p does not exist.

Pit. A point p whose surrounding is formed only of points that are of higher elevation is a pit. The same remarks as for peak apply here. The contour line through the p does not exist.

Saddle point. As shown in Figure 6.6b, a saddle point, also called a pass, is a point whose neighbourhood is composed of higher elevations on two opposite directions, and 2 lower elevations in the other two directions. From a mathematics point-of-view, it is a point for which the derivatives in orthogonal directions are 0, but the point is not maximum (peak) or a minimum (pit).

If we consider the contour line of a saddle point p, then there are 4 or more contour line segments meeting at p; for most point in a terrain this will be 2, except for peaks/pits where this is 0. Figure 6.7 shows an example for a point with an elevation of 10m, the contour lines at 10m is drawn by linearly interpolating along the edges of the TIN of the surrounding.

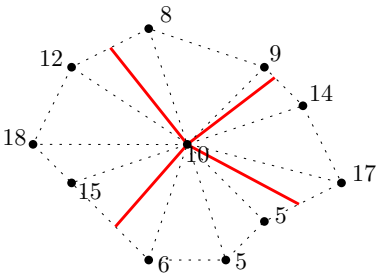


Figure 6.7: A saddle point at elevation 10m, and its surrounding points. The triangulation of the area is created and used to extract the contour line segments at 10m (red lines).

6.2.2 Valleys, ridges

Valleys and ridges are 1-dimensional features. If a terrain is represented as a TIN, we can extract the edges of the triangles that form a valley or a ridge. An edge e, incident to 2 triangles, is considered a valley-edge if the

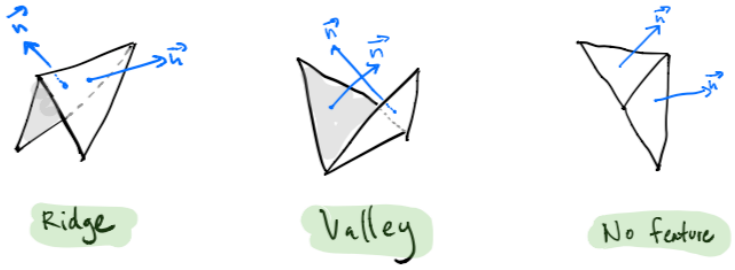


Figure 6.8: Edges in a TIN can be classified as valley, ridge, or neither

projection of the 2 normals of the triangles, projected to the xy-plane, point to e. If the 2 normals projected point in the opposite direction, then e is a ridge. If they point is different directions, then e is neither.

6.3 Curvature

The curvature is the 2nd derivative of the surface representing the terrain, it represents the rate of change of the gradient. We are often not interested in the value of the curvature itself ($\frac{\circ}{m}$) but whether the curvature is: convex, concave, or flat.

The curvature at a point p is often decomposed into types:

1. **profile curvature:** the curvature of the vertical cross-section through p perpendicular to the contour line passing through p (or of the vertical plane along the 2D vector of the aspect at p)
2. **plan curvature:** the curvature along the contour line passing through p (or along the line segment perpendicular to the 2D vector aspect and passing through p)

Because there are 2 types of curvatures and each have 3 potential values, there are 9 possible options (as Figure 6.9 shows).

6.3.1 Computing for grids

Computing the curvature is a complex operation and we will not describe one specific method. The idea is to reconstruct *locally* the surface (eg with the polynomial fitting from Section 6.1.2 above, or with a TIN), and then verify whether the 2 curvature types are convex/concave/flat. Observe that the curvature, as it is the case for the slope, is heavily influenced by the scale of the terrain (its resolution) and thus having a 3×3 kernel might be influenced by the noise in the data, or by small features.

6.3.2 Computing for TINs

For a TIN, it is possible to define for each vertex v the profile and the plan curvatures by using the triangles that are incident to v and extract the contour line for the elevation of v (as is shown in Figure 6.7). The idea is to classify each vertex into one of the 9 possibilities in Figure 6.9.

If there is no contour segment, then v is either a peak or a pit. A peak will be profile and plan convex; a pit will be profile and plan concave.

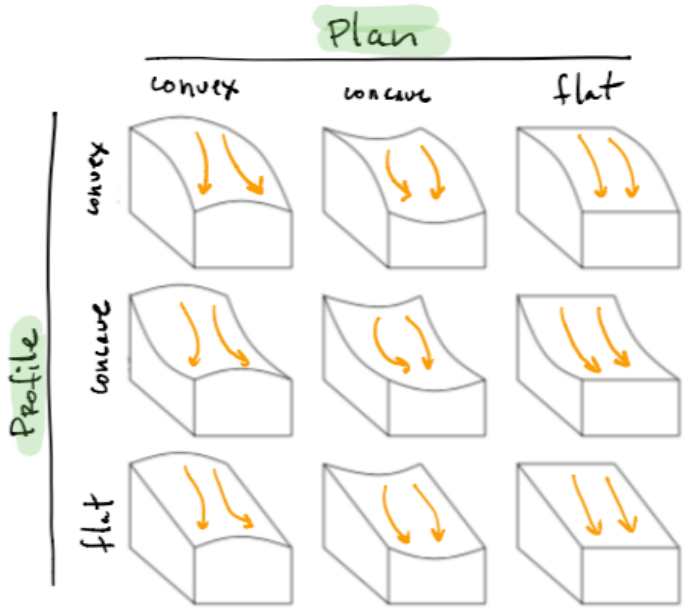


Figure 6.9: Nine curvatures (Figure adapted from Kreveld (1997)).

If there are 2 segments, then we can use these to estimate the direction of the aspect, it will be perpendicular (thus the bisector between the 2 segments is a good estimate) in the direction or lower elevations. If we simply look at the elevations higher and lower than v along this direction, then we can easily verify whether v is profile convex or concave. For the plan curvature, we can simply walk along one of the 2 edges so that higher elevations are on our left, v is plan convex if the contour line makes a left turn at v, if it makes a right turn it is concave, and if it is straight then it is plan flat.

If there are > 2 segments, then v is a saddle point and thus no curvatures can be defined.

When each point has been assigned a curvature—a pair $(profile, plan)$—we can use for instance the Voronoi diagram, as shown in Figure 6.10. It suffices to remove the Voronoi edges incident to cells having the same label, and polygonal zones are obtained.

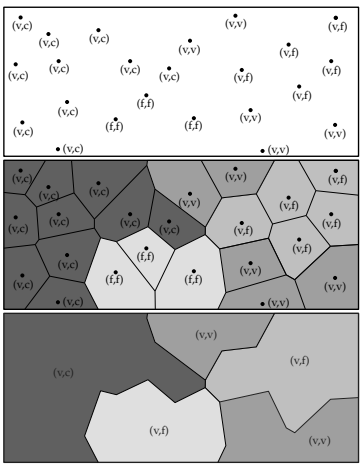


Figure 6.10: (top) Points from a TIN classified according to their curvatures (convex, concave, flat). **(middle)** The VD of the points. **(bottom)** The Voronoi edges between the cells having the same label are removed, to create polygons.

6.4 Notes and comments

The polynomial fitting method for computing the slope is from Evan (1980) and Wood (1996). Skidmore (1989) carried out a comparison of 6 methods to extract slope from regular gridded DTMs, and concluded that methods using 8 neighbours perform better than those using only 4 or the biggest height difference. He did not however investigate how the resolution of the grid influences the results.

The formula to calculate the hillshade for one cell in a gridded DTM is from Burrough and McDonnell (1998), and the ArcGIS describes it in details (`https://desktop.arcgis.com/en/arcmap/10.3/tools/spatial-analyst-toolbox/how-hillshade-works.htm`).

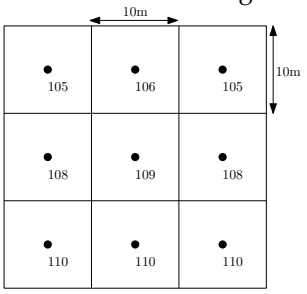
Some algorithms have been developed to identify the features forming a DTM: Kweon and Kanade (1994) and Schneider (2005) can identify simple

features in DEMs (if they are pre-processed into bilinear patches); and Magillo et al. (2009) and Edelsbrunner et al. (2001) describe algorithms to perform the same, but directly on TINs.

The algorithm to extract profile and plan curvatures from a TIN is taken from Kreveld (1997).

6.5 Exercises

1. What is the missing word? The _____ is the 2nd derivative of the surface representing the terrain, it represents the rate of change of the gradient.
2. Given a raster, how to identify a valley and a ridge?
3. If we want to compute the slope (gradient + aspect) for the cell at the centre of this 3×3 DTM with the 'finite difference method', what results will we get?



Runoff modelling 7

Many interesting DTM operations are based on runoff modelling, ie the computation of the flow and accumulation of water on a terrain. Examples include: knowing where streams will form in the case of heavy rainfall, finding the areas that will be affected by a waterborne pollutant, tracing the areas that could become submerged by floodwater, or calculating the rate of erosion or sedimentation in a given area.

In hydrology, runoff modelling can be very complex (Figure 7.1). Hydrological models usually consider different precipitation scenarios, model various types of overland and subsurface flows, and take into account many location- and time-dependent factors, such as the depth of the water table and the permeability of the soil. Such models can be quite accurate, but they require high-resolution data that is often not available, they are difficult to create without specialised knowledge, and they involve substantial manual work.

By contrast, the simpler *GIS models of runoff* can be performed automatically in large areas with only a DTM. These models mostly use gridded raster terrains, and so we will generally refer to these in this chapter, but the methods described here mostly work just as well with other representations (eg a TIN). In order for the GIS models of runoff to achieve their results, two big assumptions are usually made:

1. that all water flow is *overland*, thus ignoring all subsurface flows and dismissing factors such as evaporation and infiltration; and
2. that a good estimate for the total flow at any point is the drainage area upstream from it, ie the area above the point which drains through/to it, which is roughly equivalent to rain that is falling evenly all over a terrain.

Based on these assumptions, runoff modelling is simplified by considering only two values, which are computed for every cell in a DTM:

Flow direction Given a DTM cell, towards which nearby cells and in which proportions does water flow from it?

flow direction

Flow accumulation Given a DTM cell, what is the total water flow that passes through it?

flow accumulation

We look at a few different methods to compute these values in the next two sections.

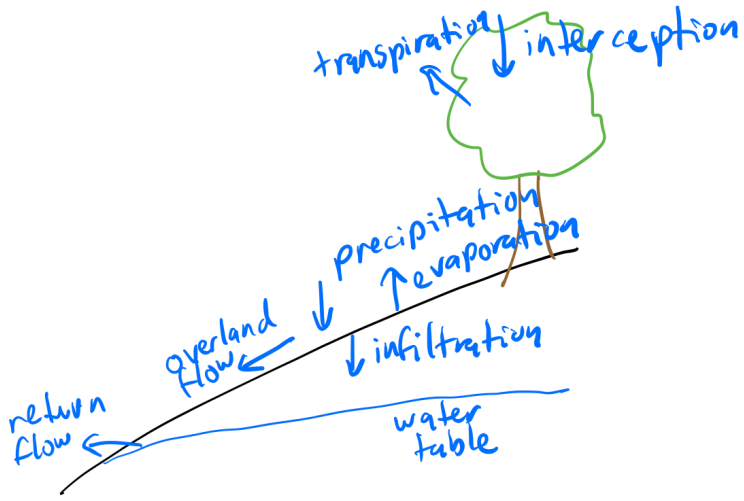


Figure 7.1: Different types of water flows as modelled in hydrology. Based on Beven (2012).

7.1 Computing the flow direction

Theoretically, the flow direction of a point is the direction with the steepest descent at that location, which does correspond to the direction towards which water would naturally flow. However, the discretisation of a terrain into DTM cells means that some kind of an approximation needs to be made. There are two broad approaches that can be followed to do this: computing a single flow direction, which assumes that all the water in a DTM cell flows to one other cell, or multiple flow directions, which assumes that the water in a DTM cell can flow towards multiple other cells.

7.1.1 Single flow direction

single flow direction (SFD)

D8 flow direction

The earliest and simplest method to compute the flow direction of a cell is to compute the slope between the centre of the cell and the centre of all its neighbouring cells (using the distance between the centres and the difference in elevation), then assign the flow direction towards the neighbour with the steepest descent. The method is known as the *single flow direction (SFD)* approach, and when applied to a raster grid, it usually considers that there are eight neighbours to each pixel (left, right, up, down and the diagonals). For this reason, it is also known as the *eight flow directions (D8)* approach.

On one hand, the method is very fast and easy to implement, and it avoids dispersing the water flow between multiple cells. On the other hand, it can have significant errors in the flow direction, and it does not allow for divergent flows. For instance, in a square grid, the errors can be of up to 22.5° (because the method is forced to choose a neighbouring cell in increments of 45°). This method can therefore easily create artefacts in certain geometric configurations (Figure 7.2).

rho8 (ρ8)

Many of these artefacts can be eliminated by using the rho8 (ρ8) method, which modifies D8 to assign the flow direction to one of its lower neighbours randomly with probability proportional to the slope. However, it

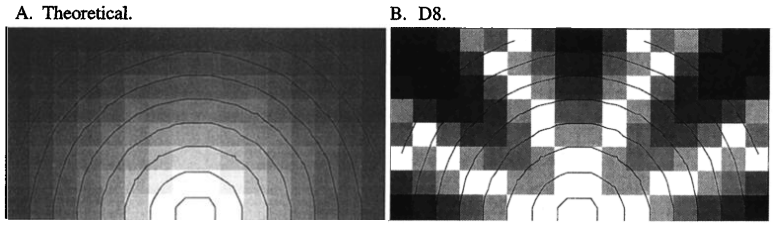


Figure 7.2: The D8 method creates artefacts when water is draining from a circular cone. From Tarboton (1997).

produces non-deterministic results, which is often a sufficient reason not to use it.

Despite its age and limitations, the SFD method is still widely used and available in many GIS tools.

7.1.2 Multiple flow directions

In an attempt to overcome the limitations of the SFD method, a variety of methods assign the flow direction of a DTM cell fractionally to some or all of its lower neighbouring cells according to some criteria. These methods are collectively known as multiple flow directions (MFD), and they usually use a variation of this equation:

multiple flow directions (MFD)

$$F_i = \frac{(L_i \tan \alpha_i)^x}{\sum_{j=1}^{n} (L_j \tan \alpha_j)^x} \tag{7.1}$$

where F_i is the flow towards the i-th neighbouring cell, L_i is the flow width (Figure 7.3), α_i is the gradient towards the i-th neighbouring cell (and so $\tan(\alpha_i)$ is the slope), x is an exponent that controls the dispersion, and n is the number of neighbours of the cell.

To read or to watch.

Look at Figures 4–9 in the following paper. These show the results of using a few flow direction computation methods in different terrains. Pay attention to how these differ from the theoretical values (if given), how some methods tend to create artefacts (eg D8 in Figure 4), and how dispersion affects some others (eg Quinn et al. (1991) in Figure 7).

D. G. Tarboton (1997). A new method for the determination of flow directions and upslope areas in grid digital elevation models. *Water Resources Research* 33.2, pp. 309–319
PDF: https://doi.org/10.1029/96WR03137

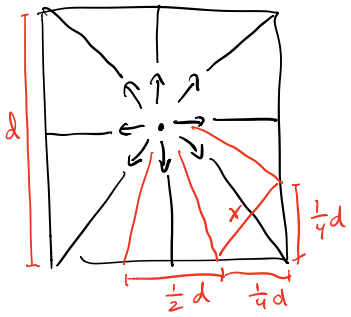


Figure 7.3: The flow width L can be computed using the geometry of the DTM cells. In the case of a square grid with spacing d, it is $\frac{\sqrt{2}}{4}d$ for the diagonals (L_2) and $\frac{1}{2}d$ for the adjacencies (L_1), where d is the grid spacing. Based on Quinn et al. (1991).

7.2 Computing the flow accumulation

After the flow directions in all the cells of a DTM have been computed, the usual next step is to use this information to compute the flow accumulation in all of them. As stated in the assumptions we make for GIS models of runoff, the flow accumulation at a given DTM cell can be

estimated by the area that drains to it. Note that in the case of a square grid, it is simply the number of cells that drain to it.

In practical terms, the flow accumulation is defined based on a recursive operation:

$$A_0 = a_0 + \sum_{i=1}^{n} p_i A_i \tag{7.2}$$

where A_0 is the accumulated flow for a cell, a_0 is the area of the cell, p is the proportion of the i-th neighbour that drains to the cell, A_i is the accumulated flow for the i-th neighbour, n is the total number of the neighbouring cells. Note that this calculation can be sped up substantially by: (i) storing the accumulated flows that have already been computed, and (ii) not following the recursion when $p_i = 0$.

7.3 Solving issues with sinks

sink

Sinks, which are also known as depressions or pits, are areas in a DTM that are completely surrounded by higher terrain. Some of these are natural features that are present in the terrain (eg lakes and dry lakebeds) and where water would flow towards (and stagnate) in reality, and are thus not a problem for runoff modelling. However, they can also be artefacts of the DTM (eg noise and areas without vegetation can create depressions), or they can be very small areas that easily flooded, after which water would flow out of them. In the latter case, we need to implement a mechanism to route water flows out of these depressions. We will look at two common options to solve this problem: modifying a DTM by filling in (certain) sinks, and implementing a flow routing algorithm that allows water to flow out of sinks.

7.3.1 Filling in sinks

The aim of the algorithms to fill in sinks is to increase the elevation of certain DTM cells in a way that ensures that all the cells in the DTM can drain to a cell on its boundary (or possibly to a set of cells that are known to be valid outlets, eg lakes and oceans). At the same time, the elevation increases should be minimised in order to preserve the original DTM as much as possible.

To read or to watch.

Section 3 in the following paper. It explains the priority-flood algorithm, which is an efficient method to fill in sinks. In short, it keeps a sorted list of DTM cells that are known to drain to the boundary (and possibly other cells that are known to drain), which is initialised with the cells on the boundary of the DTM (and possibly other cells). Then, it iteratively: (i) removes the lowest cell from this list, (ii) increases the elevation of its neighbours that are not yet known to drain to the level of the cell, (iii) adds the neighbours that are not yet known to drain to the list.

R. Barnes et al. (2014b). Priority-flood: An optimal depression-filling and watershed-labeling algorithm for digital elevation models. *Computers & Geosciences* 62, pp. 117–127
PDF: https://doi.org/10.1016/j.cageo.2013.04.024

7.3.2 Least-cost (drainage) paths

An alternative to modifying a DTM to eliminate sinks is to implement a more complex water routing algorithm that allows water to flow out of sinks. For this, the usual approach is to implement a variation of the A^* search algorithm, which in this context is known as the least-cost paths (LCP) algorithm.

least-cost paths (LCP)

To read or to watch.

Section 2.1 in the following paper. It explains the LCP algorithm and how it is implemented in GRASS. In short, it keeps a sorted list of DTM cells that are known to drain to the boundary (and possibly other cells that are known to drain), which is initialised with the cells on the boundary of the DTM (and possibly other cells). Then, it iteratively: (i) removes the lowest cell from this list, (ii) sets the drainage direction of its neighbours that are not yet known to drain towards itself, (iii) adds the neighbours that are not yet known to drain to the list. Note the similarity with the algorithm in Barnes et al. (2014b).

M. Metz et al. (2011). Efficient extraction of drainage networks from massive, radar-based elevation models with least cost path search. *Hydrology and Earth System Sciences* 15, pp. 667–678
PDF: https://doi.org/10.5194/hess-15-667-2011

7.4 Assigning flow direction in flats

Flats are areas in a DTM that have the same elevation. They therefore do not have a well-defined flow direction, which causes problems for many water routing algorithms. Flats can sometimes occur naturally, but they are more often the result of precision limits, noise removal, or sink filling algorithms.

flat

It is thus often necessary to apply a method that assigns a flow direction to flats, either by: (i) modifying the DTM to eliminate them, and then assigning them a flow direction in the usual way, or (ii) assigning them a flow direction directly.

To read or to watch.

Section 2 in the following paper. It describes an algorithm to directly assign a flow direction to flats using a combination of: (i) a gradient away from higher terrain (ie terrain goes down as we move away from higher terrain), and (ii) a gradient towards lower terrain (ie terrain

goes down as we move closer to lower terrain).

R. Barnes et al. (2014a). An efficient assignment of drainage direction over flat surfaces in raster digital elevation models. *Computers & Geosciences* 62, pp. 128–135
PDF: https://doi.org/10.1016/j.cageo.2013.01.009

7.5 Drainage networks and basins

drainage network

Interpreting DTM cells as nodes and the flow direction as directed edges connecting them yields the *drainage network* of a DTM. However, it is usually best to filter out the least important parts of the network using a flow accumulation threshold. A good rule of thumb for this threshold is the mean flow accumulation in the DTM, but an exact value is usually set by trial and error until the desired parts of the network are kept.

drainage basin

drainage divide

Based on a computed drainage network, it is then possible to extract the *drainage basins* of a DTM by considering the areas that are drained by one or more nodes of the network (Figure 7.4). This operation can be performed in many different places, such as the end node of a river (yielding its river basin), the nodes just before junctions in the network (yielding the drainage basins of the tributaries of a river), or the end nodes of a selected part of the network (yielding the drainage basin of a sea or ocean). The lines that separate adjacent drainage basins are *drainage divides*, which form topographical ridges.

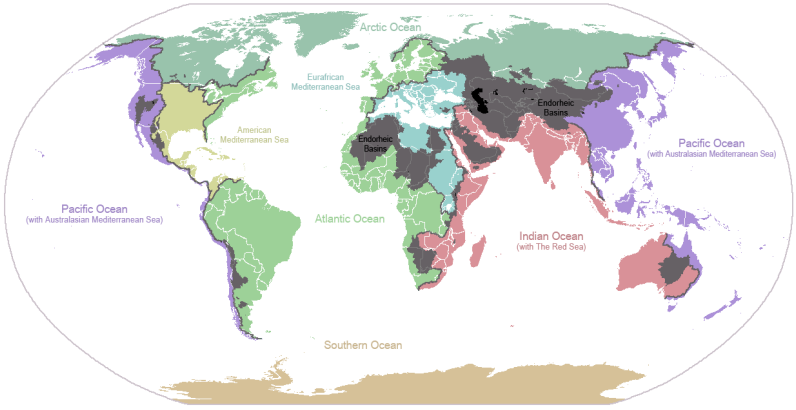


Figure 7.4: The areas that drain to all the oceans can be computed by selecting the DTM cells on the coastline of these oceans and finding the areas that drain through them. Note the endorheic basins that drain to none of these cells. These actually form sinks in the DTM. From Wikimedia Commons.

7.6 Notes and comments

Beven (2012) is a good reference book on hydrology. It covers how to make much more complex runoff models than the ones described here.

O'Callaghan and Mark (1984) was the original paper to describe the D8 method. Fairfield and Leymarie (1991) modify D8 into the stochastic rho8 method. Quinn et al. (1991) describes the original MFD method. Tarborton (1997) describes the alternative (D^{∞}) MFD method and contains nice figures comparing multiple methods.

Barnes et al. (2014b) describes how to fill in sinks, while Metz et al. (2011) describes how to use a variation of A^* search algorithm to route water out of them. Barnes et al. (2014a) describes how to assign the drainage direction over flats.

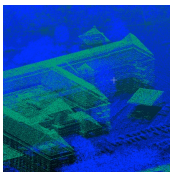
7.7 Exercises

1. Given a raster map of precipitation values, how would you be able to improve the flow accumulation estimates?
2. Why is the flow width important?
3. You have a cycle in your drainage network. How can that happen? How would you solve it?
4. How can you detect endorheic basins without finding all other basins first?

Conversions between terrain representations

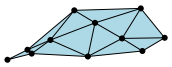
8

We consider in this chapter the following four terrain representations and discuss the conversions between them:



point cloud (PC)

raster



TIN

isolines

from/to	PC	raster	TIN	isolines
PC	—	interpolate at middle points of cells (§8.1)	create DT using 2D projection of points (ie using x and y only)	convert to TIN + extract from triangles (§8.2.2) + structure output (§8.2.3)
raster	keep middle points only	—	create TIN using middle points of cells + TIN simplification (§8.3)	extract from grid cells (§8.2.1) + structure output (§8.2.3)
TIN	keep only vertices	interpolate at middle points of cells (§8.1)	—	extract from triangles (§8.2.2) + structure output (§8.2.3)
isolines	keep only vertices — *warning: 'wedding cake' effect* (§8.4)	convert lines to points + interpolate (§8.1) — *warning: 'wedding cake' effect* (§8.4)	create DT using points — *warning: 'wedding cake' effect* (§8.4)	—

8.1 Conversion of PC/TIN to raster

As shown in Figure 8.1, this step is trivial: one needs to interpolate at the locations of the centre points of the raster cells. The interpolation method can be any of the ones described in Chapters 4 and 5.

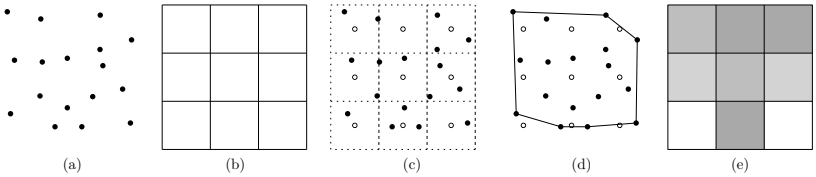


Figure 8.1: (a) input sample points. **(b)** size/location of output raster. **(c)** 9 interpolations must be performed (at locations marked with ○): at the middle of each cell. **(d)** the convex hull of the sample points show that 2 estimations are outside, thus no interpolation. **(e)** the resulting raster.

8.2 Conversion to isolines

Reading a contour map requires some skill, however it is considerably easier to learn to interpret a contour map than to manually draw one from a limited set of sample points. Yet this was exactly the task of many cartographers in the past couple of centuries: it was intuitively done by imagining a local triangulation of sample points.

Isolines are usually directly extracted from either a TIN or a grid representation of a terrain. The basic idea, as shown in Figure 8.2, is to compute the intersection between a level value (eg 200m) and each cell of the terrain (triangle or grid cell in our case). Notice that the cells are 'lifted' to their elevation. Each cell of the terrain is thus visited, one after the other, and for each cell if there is an intersection (which forms a line segment) then it is extracted. The resulting set of segment lines forms an approximation of the isoline. This process is then repeated for every level value. Notice that an isoline can have several *components*, for instance when the terrain has more than one peak.

isoline component

Therefore the number and size of the line segments in the resulting isoline are dependent on the resolution of the data representation.

The basic algorithm for extracting one isoline is shown in Algorithm 3. Note that since the algorithm contours every grid cell or triangle individually and requires only local information, it is very easy to parallelise. It is thus a scalable algorithm. Its time complexity is $\mathcal{O}(c)$, where c is the number of cells. Recall from Chapter 3 that for n points a DT contains about $2n$ triangles.

The same idea can be used to extract all the isolines: for each triangle/cell and each level value, extract all the necessary line segments.

8.2.1 Conversion of raster to isolines

Intersections are computed by linearly interpolating the elevations of the vertex pairs along the edges of this grid. Figure 8.3 illustrates the different possible configurations. The top-left case indicates the case for which there are no intersections: all vertices are either higher or lower than z_0.

Observe that two vertices are exactly at z_0, then the extraction of these is in theory not necessary because the neighbouring cell could also extract them. However, we do not want to obtain an output with duplicate line segments, and thus a simple solution to this is to only extract such line segments if they are for instance the lower and/or left segments of a given cell.

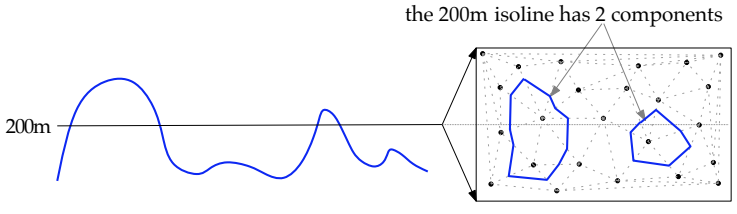


Figure 8.2: Vertical cross-section of a terrain (left), and a 2D projection of the terrain TIN with the extracted 200m isoline (right).

Algorithm 3: Simple extraction of one isoline

Input: a planar partition E formed of cells (either rectangular or triangular cells); the elevation value z_0
Output: a list of unstructured line segments representing the contour lines at z_0

```
1 segmentList ← [ ];
2 for e ∈ E do
3     if z0 intersects e then
          /* See Figures 8.3 and 8.4                    */
4         extract intersection χ of z0 with e;
5         add χ to segmentList;
```

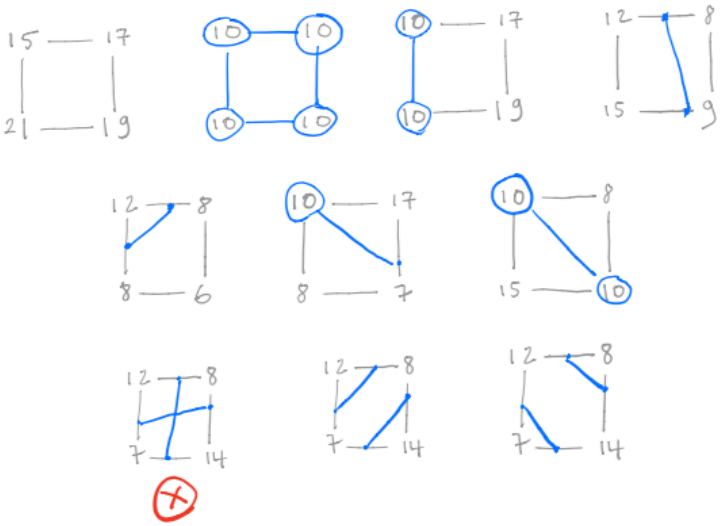


Figure 8.3: Different cases when extracting an isoline at elevation 10m for a regular grid. The grey values are the elevation of the vertices forming one regular cell, and the blue lines and vertices are the ones extracted for that cell.

The most interesting case is the bottom-left one in Figure 8.3, it occurs when the two pairs of opposing points are respectively higher and lower than z_0. This forms a saddle point. The ambiguity arises here since there are two ways to extract a valid pair of contour line segments (only one of the 2 options must be extracted). This can be resolved by simply picking a random option or consistently choose one geometric orientation.

8.2.2 Conversion TIN to isolines

Since a triangle has one fewer vertex/edge than a square grid cell, there are less possible intersection cases (Figure 8.4) and, more importantly, there is no ambiguous case. The worst case to handle is when there are horizontal triangles at exactly the height of the isoline. Otherwise, the intersection cases are quite similar to the raster situation and they can be easily implemented.

To avoid extracting twice the same line segment when 2 vertices are at z_0 (case on the right in Figure 8.4), then we can simply look at the normal of the edge segment: if its y-component is positive then it can be added, if $y = 0$ then only add if the x-component is positive.

Observe that since the algorithm is simpler than that for a raster dataset, one way to extract isolines from a raster dataset is by first triangulating

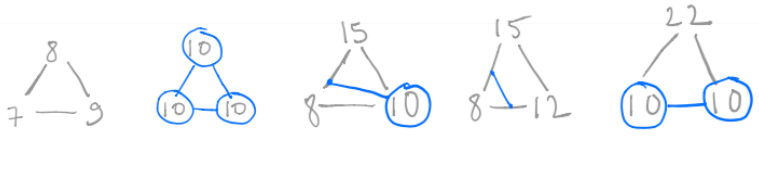


Figure 8.4: Different cases when extracting an isoline at elevation 10m for a TIN. The grey values are the elevation of the vertices forming one triangle, and the blue lines and vertices are the ones extracted for that triangle.

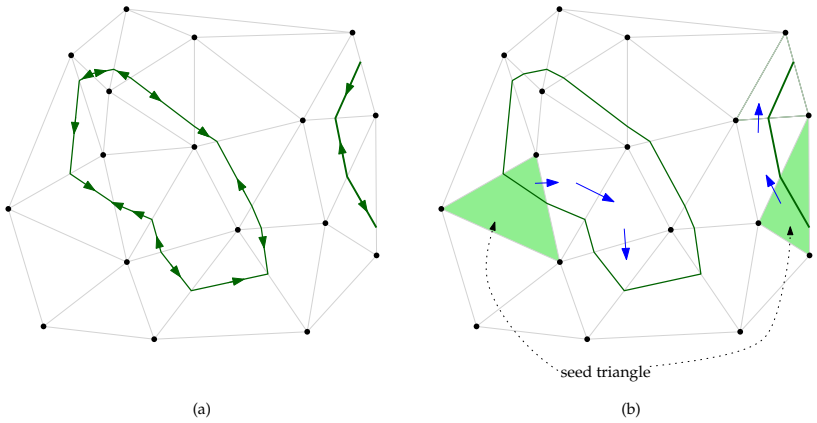


Figure 8.5: (a) The isoline segments extracted with Algorithm 3 do not have a consistent orientation. **(b)** Algorithm 3 can be sped up by starting at a seed triangle and 'tracing' the isoline; the order is shown by the blue arrows.

it: each square cell is subdivided into 2 triangles (simply ensure that the diagonal is consistent, eg from lower-left to top-right).

8.2.3 Structuring the output

The line segments obtained from the simple algorithms above are not structured, ie they are in an arbitrary order (the order in which we visited the triangles/cells) and are not connected. Furthermore, the set of line segments can form more than one *component*, a set of segments forming a closed polygon (unless they are at the border of the dataset). Perhaps the only application where having unstructured line segments is fine is for visualisation of the lines. For most other applications this can be problematic, for instance:

1. if one wants to know how many peaks above 1000m there are in a given area;
2. if smoothing of the isolines is necessary, with the Douglas-Peucker algorithm for instance;
3. if a GIS format requires that the isolines be closed polylines oriented such that the higher terrain is on the left for instance, such as for colouring the area enclosed by an isoline.

To obtain structured segments, the simplest solution is to merge, as post-processing, the line segments based on their start and end vertices. Observe that the line segments will not be consistently oriented to form one polygon (see Figure 8.5a), that is the orientation of the segments might need to be swapped. This can be done by simply starting with a segment ab, and searching for the other segment having b as either start or end vertex, and continue until a component is formed (a polygon is formed), or until no segment can be found (the border of the dataset is reached, as shown in Figure 8.5a).

As shown in Figure 8.5b, another solution is to find *one* cell τ_0 intersecting the isoline at a given elevation, 'tracing' the isoline by navigating from $\tau_0$to the adjacent cell, and continuing until τ_0 is visited again (or the

border of the dataset is reached). To navigate to the adjacent cell, it suffices to identify the edge ϵ intersecting the isoline, and then navigating to the triangle/cell that is incident to ϵ. It is possible that there is no adjacent cell, if the boundary of the convex hull is reached in a TIN for instance. This requires that the TIN be stored in a topological data structure in which the adjacency between the triangles is available (for a grid this is implied).

The main issue is finding the starting cells (let us call them seed triangles). Obviously, it suffices to have one seed for each of the component of the isolines (there would be 2 seeds in Figure 8.5b). An easy algorithm to extract all the components of an isoline requires visiting all the cells in a terrain, and keeping track of which triangles have been visited (simply store an Boolean attribute for each triangle, which is called a *mark bit*). Simply visit triangle sequentially and mark them as 'visited', when one triangle has an intersection then start the tracing operation, marking triangles as visited as you trace.

8.2.4 Smoothness of the contours

The mathematical concept of the *Implicit Function Theorem* states that a contour line extracted from a field f will be no less smooth than f itself. In other words, obtaining smooth contour lines can be achieved by smoothing the field itself. Robin Sibson[1] goes further in stating that:

> *'The eye is very good at detecting gaps and corners, but very bad at detecting discontinuities in derivatives higher than the first. For contour lines to be accepted by the eye as a description of a function however smooth, they need to have continuously turning tangents, but higher order continuity of the supposed contours is not needed for them to be visually convincing.'*

1: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.51.63

In brief, in practice we should use interpolant functions whose first derivative is continuous (ie C^1) if we want to obtain smooth contours. C^0 interpolants are not enough, and C^2 ones are not necessary.

8.3 Simplification of a TIN

The TIN simplification problem is:

> Given a TIN formed by the Delaunay triangulation of a set S of points, the aim is to find a subset R of S which will approximate the surface of the TIN as accurately as possible, using as few points as possible. The subset R will contain the 'important' points of S, ie a point p is important when the elevation at location p can not be accurately estimated by using the neighbours of p.

The overarching goal of TIN simplification is always to (smartly) reduce the number of points in the TIN. This reduces memory and storage requirements, and speeds up TIN analysis algorithms.

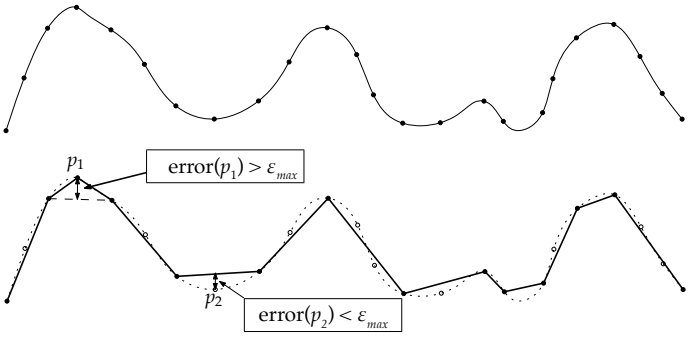


Figure 8.6: The importance measure of a point can be expressed by its vertical error. When this error is greater than a given threshold $\epsilon_{\max}$, the point is kept (p_1), else it is discarded (p_2).

Observe that the simplification of a TIN can be used to simplify a raster terrain: we can first obtain the triangulation of the middle points of each cell, and then simplify this TIN to obtain a simplified terrain.

8.3.1 The importance of a point

The importance of a point is a measure that indicates the error in the TIN when that point would not be part of it. Imagine for instance a large flat area in a terrain. This area can be accurately approximated with only a few large triangles, and inserting points in the middle of such an area does not make the TIN more accurate. An area with a lot of relief on the other hand can only be accurately modelled with many small triangles. We can therefore say that the points in the middle of the flat area are less important than the points in the area with relief.

The importance of a point—or importance measure—can be expressed in several ways, eg based on an elevation difference or the curvature of the point. Here we focus on the *vertical error* which has proven to be effective in practice.

The vertical error of a point p is the elevation difference between p itself and the interpolated elevation in the TIN $\mathcal{T}$ at the (x, y) coordinates of p (see Figure 8.6). Notice that $\mathcal{T}$ does not contain p as a vertex.

8.3.2 TIN simplification algorithms

There are two main approaches to TIN simplification: decimation and refinement. In a decimation algorithm, we start with a TIN that contains all the input points, and gradually remove points that are not important. In a refinement algorithm, we do the opposite: we start with a very simple TIN, and we gradually refine it by adding the important points.

8.3.2.1 TIN simplification by refinement

Here we describe an iterative refinement algorithm based on greedy insertion*. It begins with a simple triangulation of the spatial extent and, at each iteration, finds the input point with highest importance—the

* A greedy algorithm is one that divides a complex problem into a series of easier steps, then solves it by making the locally optimal choice at each step, and never goes back on this choice. In our case the heuristic is the importance measure, ie the vertical error. See https://en.wikipedia.org/wiki/Greedy_algorithm.

Algorithm 4: TIN simplification by refinement

Input: A set of input points S, and the simplification threshold $\epsilon_{\max}$
Output: A triangulation $\mathcal{T}$ that consists of a subset of S and that satisfies $\epsilon_{\max}$

```
1  Construct an initial triangulation T that covers the 2D bounding box
     of S ;
2  ε ← ∞ ;
3  while ε > ε_max do
4      ε ← 0 ;
5      q ← nil ;
6      for all p ∈ S do
7          τ ← the triangle in T that contains p ;
8          ε_τ ← the vertical error of p with respect to τ ;
9          if ε_τ > ε then
10             ε ← ε_τ ;
11             q ← p ;
       /* insert the point q that has the largest error:   */
12     insert into T the point q ;
13     remove q from S ;
```

highest vertical error—in the current TIN and inserts it as a new vertex in the triangulation. The algorithm stops when the highest error of the remaining input points with respect to the current TIN is below a user-defined threshold $\epsilon_{\max}$. Algorithm 4 shows the pseudo-code. It is also possible to insert only a certain percentage of the number of input points, eg we might want to keep only 10% of them.

8.3.2.2 TIN simplification by decimation

The implementation of the decimation algorithm is similar to the refinement algorithm. The main differences are

1. we start with a full triangulation of all the input points, instead of an empty triangulation;
2. instead of iteratively adding the point with the highest importance, we iteratively remove the point with the lowest importance, and
3. in order to compute the importance of a point we actually need to remove it *temporarily* from the triangulation before we can decide if it should be permanently removed. In other words: we need to verify what the vertical error would be if the point was not present.

Algorithm 5 shows the pseudo-code for the TIN decimation algorithm. It should be noticed that the implementation of this algorithm requires a method to delete/remove a vertex from a (Delaunay) triangulation, and that many libraries do not have one[†].

Observe that the Algorithms 4 and 5 both state that the importance of the points must be completely recomputed after each iteration of the algorithms (either one removal or one insertion), but that in practice several of these will not have changed. As can be seen in Chapter 3, the insertion/deletion of a single point/vertex will only *locally* modify the

[†] The implementation in SciPy does not allow to remove one vertex, but CGAL does (www.cgal.org)

Algorithm 5: TIN simplification by decimation

Input: A set of input points S, and the simplification threshold $\epsilon_{\max}$
Output: A triangulation $\mathcal{T}$ that consists of a subset of S and satisfies $\epsilon_{\max}$

```
1  T ← a triangulation of S ;
2  ε ← 0 ;
3  while ε < ε_max do
4      ε ← ∞ ;
5      q ← nil ;
6      for all p ∈ T do
7          remove p from T ;
8          τ ← the triangle in T that contains p ;
9          ε_τ ← the vertical error of p with respect to τ ;
10         if ε_τ < ε then
11             ε ← ε_τ ;
12             q ← p ;
13         put back p into T ;
       /* remove the point q that has the smallest error:  */
14     remove from T the point q ;
```

triangulation, and it is thus faster from a computational point of view to flag the vertices incident to the modified triangles, and only update these.

8.3.3 Comparison: decimation versus refinement

While both methods will allow us to obtain similar results, the properties of the resulting terrain are different. Consider the threshold $\epsilon_{\max}$ that is used to stop the simplification process. If the refinement method is used, then it is guaranteed that the final surface of the terrain will be at a maximum of $\epsilon_{\max}$ (vertical distance) to the 'real surface' because all the points of the input are considered. However, with the decimation method, after a vertex is deleted from the TIN, it is never considered again when assessing whether a given vertex has an error larger than $\epsilon_{\max}$. It is thus possible that the final surface does not lie within $\epsilon_{\max}$, although for normal distribution of points, it should not deviate too much from it.

In practice, refinement is often computationally more efficient than decimation because we do not need to first build a TIN from all input points before removing several of them again. However, decimation could be more efficient when you already have a detailed TIN, stored in a topological data structure, that just needs to be slightly simplified.

8.4 Conversion isolines to TIN/raster creates the 'wedding cake effect'

If the input is a set of isolines, then the simplest solution is, as shown in Figure 8.7a, to convert these to points and then use any of the interpolation methods previously discussed. This conversion can be done by either

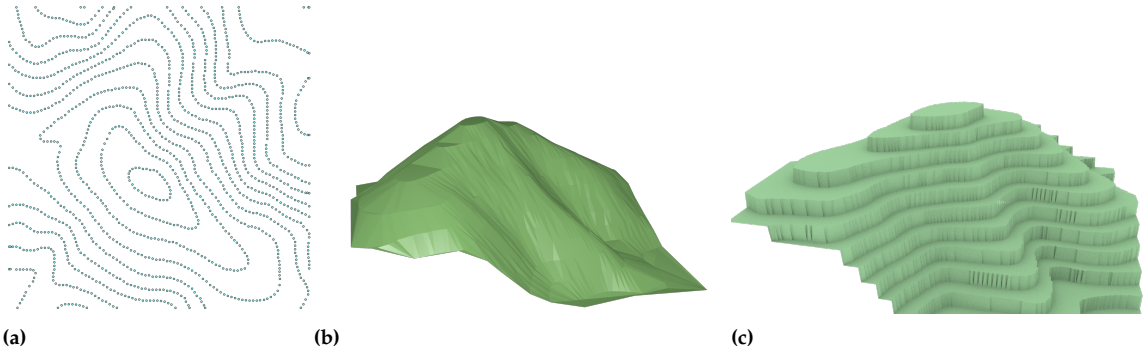


Figure 8.7: The 'wedding cake' effect. **(a)** The input isolines have been discretised into sample points. **(b)** The TIN of the samples creates several horizontal triangles. **(c)** The surface obtained with nearest-neighbour interpolation.

keeping only the vertices of the polylines, or by sampling points at regular intervals along the lines (say every 10m). However, one should be aware that doing so will create terrains having the so-called *wedding cake effect*. Indeed, the TIN obtained with a Delaunay triangulation, as shown in Figure 8.7b, contains several horizontal triangles; these triangles are formed by 3 vertices from the same isoline. If another interpolation method is used, eg nearest neighbour (Figure 8.7c), then the results are catastrophic.

Solving this problem requires solutions specifically designed for such inputs. The main ideas for most of them is to add extra vertices between the isolines, to avoid having horizontal triangles. One strategy that has proven to work is to add the new vertices on the *skeleton*, or medial-axis transform, of the isolines, which are located 'halfway' between two isolines. The elevation assigned to these is based on the elevations of the isolines.

8.5 Notes & comments

The *Implicit Function Theorem* is further explained in Sibson (1997).

Dakowicz and Gold (2003) describe in details the skeleton-based algorithm to interpolate from isolines, and show the results of using different interpolation methods.

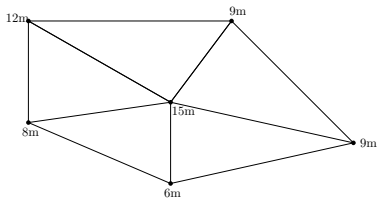
The basic algorithm to extract isolines, which is a brute-force approach, can be slow if for instance only a few isolines are extracted from a very large datasets: all the n triangles/cells are visited, and most will not have any intersections. To avoid this, Kreveld (1996) build an auxiliary data structure, the *interval tree*, which allows us to find quickly which triangles will intersect a given elevation. It is also possible to build another auxiliary structure, the contour tree, where the triangle seeds are stored (Kreveld et al., 1997). Such methods require more storage, but can be useful for interactive environment where the user extracts isolines interactively.

Garland and Heckbert (1995) elaborate further on different aspects of TIN simplification, such as different importance measures, the differences between refinement and decimation, and the usefulness of data-dependent triangulations. They also show how Algorithm 4 can be made a lot faster

by only recomputing the importance of points in triangles that have been modified.

8.6 Exercises

1. When converting isolines to a TIN, what main "problem" should you be aware of? Describe *in details* one algorithm to convert isolines (given for instance in a *shapefile*) to a TIN and avoid this problem.
2. How would the isocontours of a 2.75D terrain look like?
3. In Section 8.2.3, it is mentioned that merging the segments will form on polygon. But how to ensure that the orientation of that resulting curve is consistent, that it is for instance having higher terrains on the right?
4. Given a raster terrain (GeoTiff format) that contains several cells with `no_data` values, describe the methodology you would use to extract contour lines from it. As a reminder, contours lines should be closed curves, except at the boundary of the dataset.
5. Assume you have the small terrain formed of 3 triangles below, draw the isoline in this TIN for an elevation of 10m.



Spatial extent of a point cloud

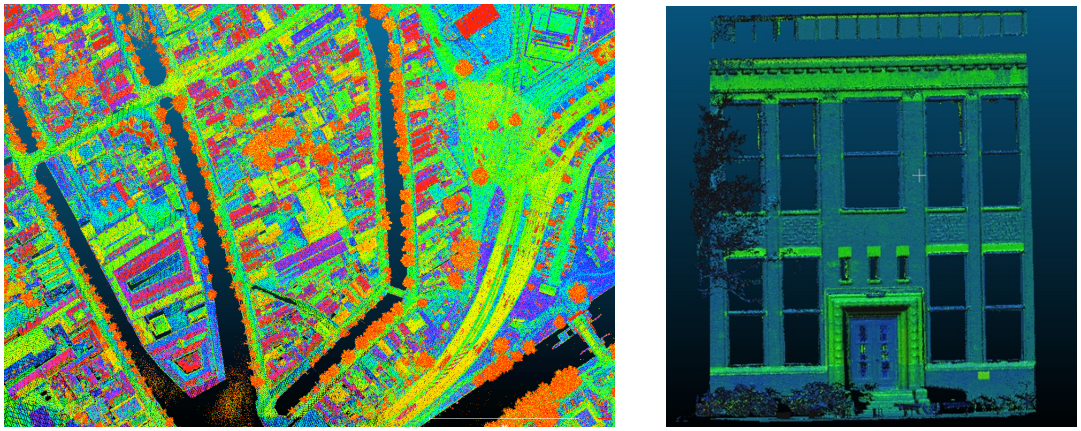
9

Given a point cloud, one operation that practitioners often need to perform is to define the spatial extent of the dataset. That is, they need to define the shape of the region that best abstracts or represents the set of points. As seen in Figure 9.1, this region is often in two dimensions, for example in the case of an aerial lidar datasets we want to know where the ground is (after removing the points on the water), or in the case of the scanning of the façade of a building, we would like to obtain a polygon that represents where the wall is (omitting the windows).

Calculating the spatial extent is useful to calculate the area covered by a dataset, to convert it to other formats (eg raster), or to get an overview of several datasets it is faster to load a few polygons instead of billions of points, etc.

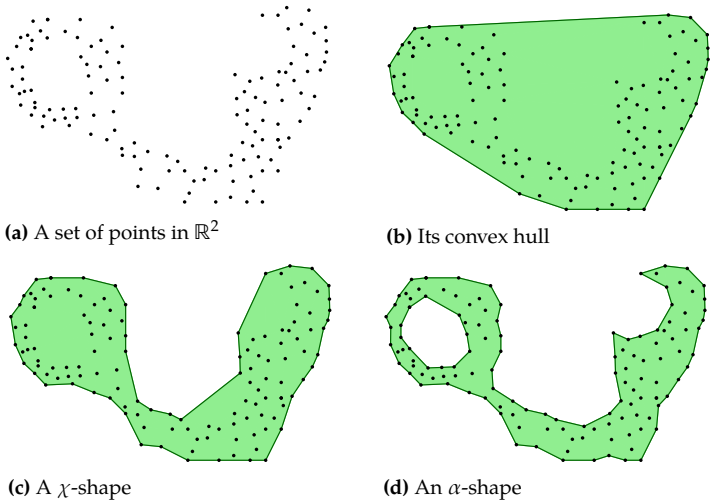
The spatial extent is often called by different names, for instance: envelope, hull, concave hull, or footprints. It is important to notice that the spatial extent is not uniquely defined and that it is a vague concept. As Figure 9.2 shows, there are several potential regions for a rather simple set of points, and most of these could be considered 'correct' by a human.

In this chapter we present methods that are used in practice to define the spatial extent of a set of points in $\mathbb{R}^2$, which implies that the points in a point cloud are first projected to a two-dimensional plane.



(a) **(b)**

Figure 9.1: Two point cloud datasets for which we would like to find the spatial extent. **(a)** An aerial point cloud with several canals (dark colour). **(b)** A scan of a façade containing several windows.



(a) A set of points in $\mathbb{R}^2$ **(b)** Its convex hull **(c)** A χ-shape **(d)** An α-shape

Figure 9.2: Different methods to obtain the spatial extent of a given set of points in the plane.

9.1 Properties of the region

Let S be a set of points in $\mathbb{R}^2$, and $R(S)$ the region that characterise the spatial extent of S. The region is potentially formed by a set of polygons (if S forms two distinct clusters for instance), and in practice most algorithms will compute a linear approximation of $R(S)$, so the polygons will have straight edges as boundaries.

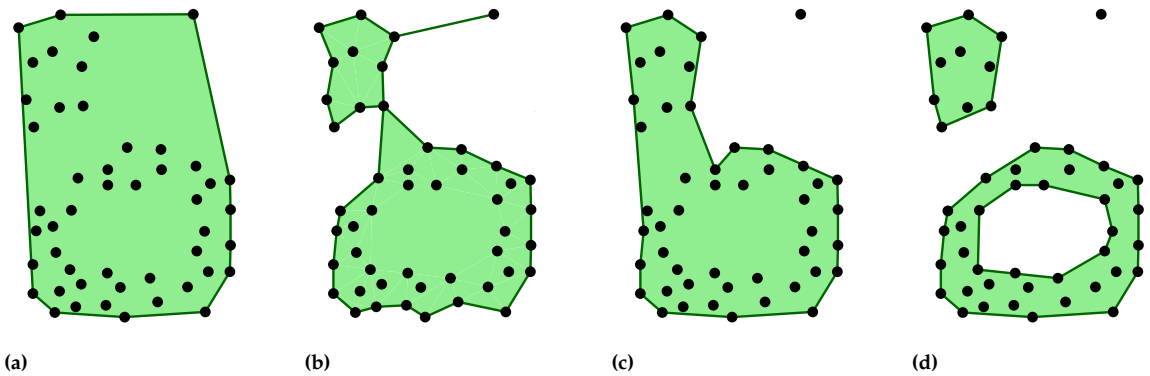
To evaluate the different algorithms to create R(S), we list here different properties that one must consider when defining the spatial extent of a set of points.

P1. ***Regular* polygons?** Are polygons allowed to have dangling parts (lines), such as the one in Figure 9.3b

P2. **All points part of the region?** Can outliers be ignored? Or do they have to be part of the region? In Figure 9.3a and Figure 9.3b they are all part of the region, in Figure 9.3c and Figure 9.3d one outlier is not.

P3. **Region is one connected component?** Or are more components allowed? In Figure 9.3a–Figure 9.3c there is one component, but Figure 9.3d has two.

P4. **Are holes allowed in a polygon?** Polygons in Figure 9.3a–**??** have only an exterior boundary, while in Figure 9.3d one polygon has



(a) **(b)** **(c)** **(d)**

Figure 9.3: Different properties for the spatial extent

an interior boundary too (a hole).

P5. Computational efficiency What is the time complexity of the algorithm, and does it require large and complex auxiliary data structures?

9.2 Convex hull

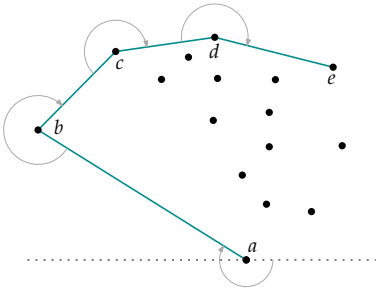
As explained in Section 3.2.1, given S, a set of points in $\mathbb{R}^2$, its convex hull, which we denote conv(S), is the minimal convex set containing S. Two examples of convex hulls are in Figures **??**b and **??**a.

For a given set of points, the convex hull is uniquely defined and does not require any parameters (unlike the other methods listed below). It is also relatively easy to compute: it can be extracted from the Delaunay triangulation, or compute directly using a specialised algorithm. For example, using the well-known *gift wrapping algorithm*, shown in Figure 9.4. It begins with a point that is guaranteed to be on conv(S) (we can take an 'extreme', such as a in Figure 9.4 because it is the point with the lowest y-coordinate), and then picks the point in S (omitting the ones already on conv(S)) for which the polar angle between the horizontal line and that point (at a) is the smallest (b in this case), and adds it to conv(S). Then for b, the polar angle is calculated from the line ab; and the algorithm continues this way until a is visited again.

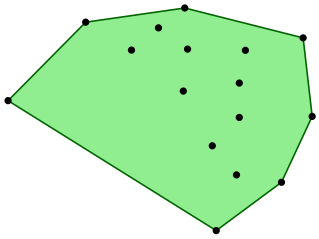
If S has n points and conv(S) is formed of h points, then the gift wrapping algorithm has a time complexity of $\mathcal{O}(n\,h)$; each of the h points are tested against all n points in S. However, there exist more efficient algorithm that have a time complexity of $\mathcal{O}(n \log n)$.

Properties convex hull:

P1.	The sole polygon is guaranteed to be regular (and convex).
P2.	All points are on or inside the region.
P3.	One component.
P4.	No holes in the region.
P5.	$\mathcal{O}(n \log n)$



(a)



(b)

Figure 9.4: (a) First four steps of the gift wrapping algorithm to compute the convex hull. **(b)** The resulting convex hull.

9.3 Moving arm

Arm of length l. The moving arm is a generalisation of the gift wrapping algorithm where the infinite line, used to calculate the polar angles, is replaced by a line segment of a given length l (the "moving arm"). This means that, unlike the original gift wrapping algorithm, only a subset of the points in S are considered at each step. This also means that potentially the result is a polygon that is non-convex. Figure 9.5 shows the first few steps for a given l, and it can be observed that 1 point is not part of the final region. Observe also that if l had been larger then conv(S) could have been obtained.

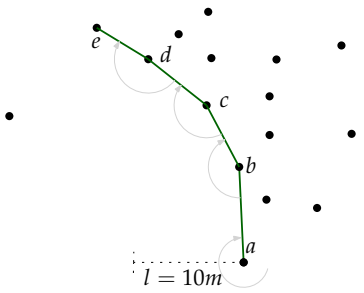


Figure 9.5: First four steps of the *moving arm algorithm* (with a lenght l) to compute the spatial extent.

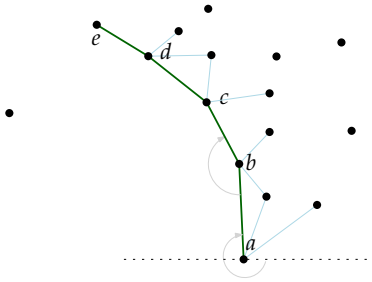


Figure 9.6: First four steps of the *moving arm algorithm* (with a *knn* where $k = 3$) to compute the spatial extent.

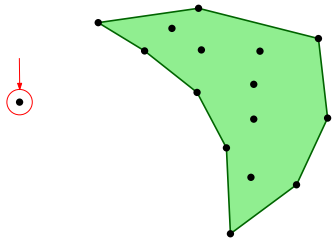
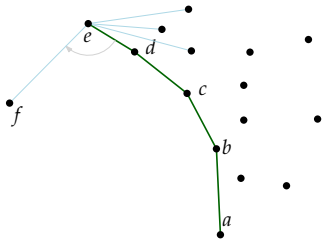
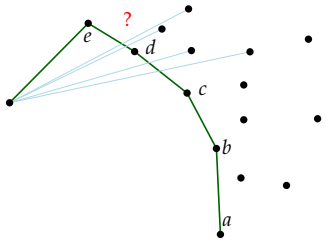


Figure 9.7: The resulting region for the moving arm, it is concave. Observe that 1 point from S (highlighted in red) is not part of the region.

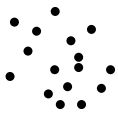


(a)



(b)

Figure 9.8: Moving arm with *kdd* when $k = 4$. **(a)** When the point e is being processed, f is the next one chosen. **(b)** From f, no other points can be chosen since the resulting region would be self-intersecting.



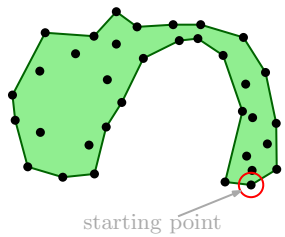


Figure 9.9: S has two distinct clusters. In green the typical output if S processed as a single cluster with a moving arm algorithm.

Adaptative arm with *knn*. There exists a variation of this algorithm where the length of the moving arm is adaptive at each step; the k nearest neighbours (knn) of a given point p are used to determine it (see Section 10.2). As can be seen in Figure 9.6, the largest polar angle, as used for gift wrapping algorithm, is used to select the point at each step.

No guarantee that it will work. Both versions of the algorithm will work in most cases, but there is no guarantee that they will for all inputs. Figure 9.8 shows a concrete example. In this case, a solution to this problem would be to either choose another k, or to rotate counter-clockwise instead of clockwise, which will in practice yield different results.

Different clusters? One drawback of the moving arm method is that only one polygon is obtained as a region. If S forms different clusters (see for instance Figure 9.9), then only the cluster that is the 'lowest' will be output for the region (since the lowest point is picked as a starting point). Notice that this can also be useful to discard unwanted outliers (unless the lowest point is an outlier). In practice, the problem of several clusters can be solved by preprocessing the input points with a clustering algorithm (in the case of Figure 9.9 two clusters should be detected) and then each cluster is processed separately. See Section 9.6 for an overview.

The worst case time complexity is the same as for the gift wrapping algorithm: $\mathcal{O}(n\,h)$. If a kd-tree is used, this stays the same but in practice will be sped up as the subset of S tested will be smaller. Each query in a kd-tree takes $\mathcal{O}(\log n)$ on average, but we need to store an auxiliary structure that takes $\mathcal{O}(n)$ storage.

Properties moving arm:

P1.	The sole polygon could be degenerate (self-intersection).
P2.	Outliers can be discarded (except if it is the lowest point)
P3.	One component.
P4.	No holes in the region.
P5.	$\mathcal{O}(n\,h)$

9.4 χ-shape

The χ-shape is based on first constructing the Delaunay triangulation (DT) of S, and then removing iteratively the longest edge forming the envelop (at first this envelop is conv(S)) until no edge is longer than a given threshold l. The idea is to construct one polygon that is potentially non-convex, and that contains all the points in S. Before removing an edge, we must verify that it will not introduce a topological issue in the envelop, that is that the envelop will not contain a self-intersection (see Figure 9.10c, the dangling edge is there twice, once in each direction).

A DT can be constructed in $\mathcal{O}(n \log n)$. The number of edges in a DT of n points is roughly $3n$ (thus $\mathcal{O}(n)$), and since verifying the topological constraint can be done locally (previous and next edge) the overall time complexity is $\mathcal{O}(n \log n)$

Properties χ-shape:

P1.	The sole polygon is guaranteed to be regular.
P2.	All points are part of the region
P3.	One component.
P4.	No holes in the region.
P5.	$\mathcal{O}(n \log n)$

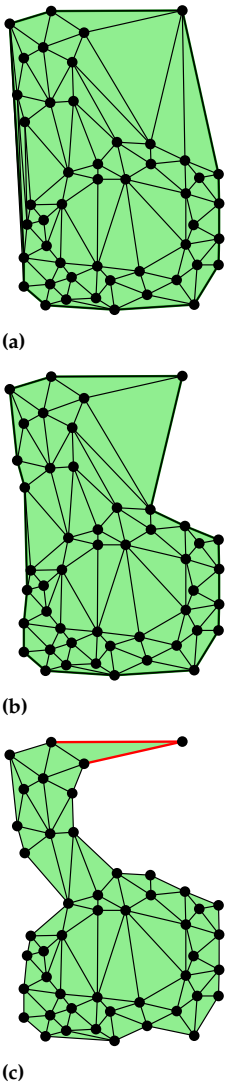


Figure 9.10: χ-shape examples. **(a)** S, its DT, and its envelope (cons(S)). **(b)** After some edges have been removed. **(c)** The final result for a given threshold. Observe that neither of the red edges can be removed because a self-intersection would be created

9.5 α-shape

The α-shape is conceptually a generalisation of the convex hull of a set S of points.

It is best understood with the following analogy. First imagine that $\mathbb{R}^2$ is filled with Styrofoam and that the points in S are made of hard material. Now imagine that you have a carving tool which is a circle of radius α, and that this tool can be used anywhere from any direction (it is 'omnipresent'), and that it is only stopped by the points. The result after carving, called the α-hull, is one or more pieces of Styrofoam. If we straighten the circular edges, then we obtain the α-shape. See Figure 9.11 for an example.

Now let α be a real number with $0 \leq \alpha \leq \infty$. If $\alpha = \infty$, then the α-shape is conv(S) because you will not be able to carve inside conv(S). As α decreases, the α-shape shrinks and cavities can appear, and different components can be created. If $\alpha = 0$ then the α-shape is S (which is a valid α-shape).

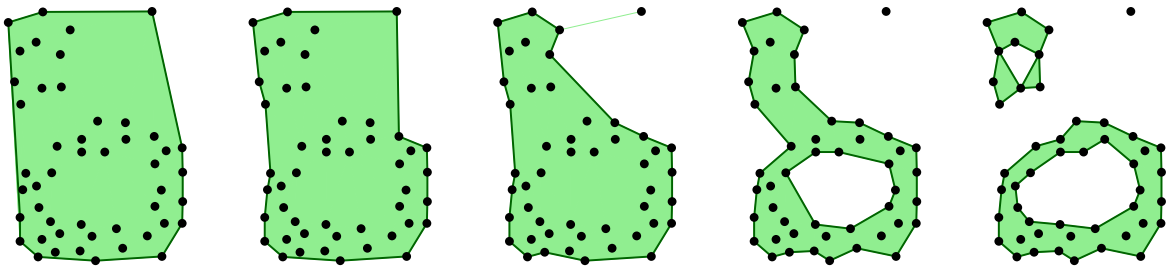


Figure 9.11: Five α-shape for the same set of points, with decreasing α values from left to right.

The α-shape is not a polygon or a region, but a complex formed of k-simplices, where $0 \leq k \leq 2$. Furthermore, it is a subcomplex of the Delaunay triangulation (DT) of S. That is, the easiest method to construct an α-shape is by first constructing DT(S), and then removing all edges that are shorter then 2α.

In practice, all the α-shapes of S (for different values of α) can be calculated and discretised since we know that α will range from the shortest edge to the longest. For each k-simplex, we can thus assign a range where it will be present. Implementations of the α-shape will often offer to compute automatically an α such that the complex obtained is for instance connected and contains only one polygon.

A DT can be constructed in $\mathcal{O}(n \log n)$ time, and the algorithm only requires to visit once each of the $\mathcal{O}(n)$ triangles, thus the time complexity is $\mathcal{O}(n \log n)$.

Properties α-shape:

P1.	A complex of k-simplices.
P2.	Some points can be omitted.
P3.	Several components possible.
P4.	Regions can contain holes.
P5.	$\mathcal{O}(n \log n)$

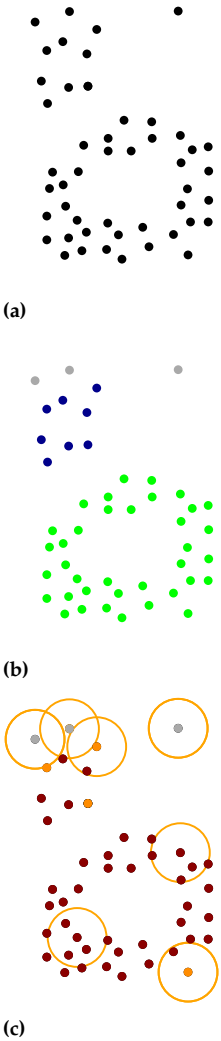


Figure 9.12: Clustering points. **(a)** A set of points S. **(b)** The result of a clustering algorithm (DBSCAN): points are assigned to a group (based on colours, here 2 groups) or labelled as outliers (grey points). **(c)** DBSCAN has 3 types of points: core points (dark red), border points (orange), and outliers (grey); the orange circle is the ϵ and the $n_{\min} = 2$.

9.6 Clustering algorithms

Clustering algorithms are used widely for statistical data analysis, they allow us to group points (often in higher dimensions) that are close to each others in one group. Different notions to create clusters can be used, eg distance between the points, density, intervals or particular statistical distributions. As Figure 9.12 shows, the result of a clustering algorithm is that each input point is assigned to a cluster (here a colour), and potentially some outliers are identified.

k-mean clustering. It is a centroid-based clustering, where a cluster is represented by its centroid. The parameter k is the number of clusters, usually given as input. A point belongs to a given cluster if its distance to the centroid is less than for any other cluster centroids. The algorithm can be seen as an optimisation problem, since we want the total distances from each point to its assigned centroid to be minimised. In practice, we often seek approximate solutions, for instance the location of the k centroids are first randomised, and thus the algorithm will yield different outputs. The algorithm is iterative: at each iteration the points are assigned to the closest centroid, and new centroid locations are updated. The algorithm stops when the centroids have converged and their locations do not change.

DBSCAN: density-based clustering. A density-based cluster is defined as an area of higher density then other points, the density being the number of points per area. The aim is to group points having many close neighbours.

The most used algorithm is DBSCAN (density-based spatial clustering of applications with noise), and works as follows. First the density is defined with 2 parameters: (1) ϵ is a distance defining the radius of the neighbourhood around a given point; (2) $n_{\min}$ is the minimum number of points that a neighbourhood can contain to be considered a cluster. Points are categorised either as: (1) *core points* if they have more than $n_{\min}$ neighbours within ϵ; (2) *border points* if they have less than $n_{\min}$ neighbours within ϵ, but are closer than ϵ to a core point; (3) *outliers*. In Figure 9.12c, if $n_{\min} = 2$ and the orange circle represents ϵ, notice that a few points are border points since they have only one point in their neighbourhood, and that 3 points are outliers (1 is clear, the other 2 are very close to being border points). A cluster is formed by recursively finding all the neighbouring points of a given core point, adding them to the cluster.

9.7 Notes & comments

The properties listed in Section 9.1 are taken, and slightly adapted, from Galton and Duckham (2006).

The *quickhull* algorithm is the most known and used convex hull algorithm, and it is valid in any dimensions. See Barber et al. (1996) for the details, and `http://www.qhull.org/` for implementations.

The gift wrapping algorithm to compute the convex hull of a set of points in $\mathbb{R}^2$ is from Jarvis (1973).

The moving arm with a length is presented and described in Galton and Duckham (2006), and the adaptative one in Moreira and Santos (2007). In both papers, the authors describe different strategies to make the algorithm work for all input, but these do not have any warranty to output a simple polygon.

The χ-shape was introduced in Duckham et al. (2008).

The explanation of the α-shape is taken from Edelsbrunner and Mücke (1994) and from the CGAL documentation[1].

1: `https://doc.cgal.org/latest/Alpha_shapes_2/index.html`

The DBSCAN algorithm was introduced in Ester et al. (1996).

9.8 Exercises

1. Given a DT(S), how to extract conv(S)?
2. What are the disadvantages of the χ-shape compared with the α-shape?
3. If the parameter l for the χ-shape is equal to the α parameter for an α-shape, will the resulting shapes be the same?
4. Given an α-shape of S, how to calculate how many components are part of it?
5. Draw what would happen if one of the 2 edges was removed in Figure 9.10c.
6. What is the influence of k for the moving arm algorithm (with a *kdd*)? Will a higher k create a larger or smaller region in general?

Handling massive terrains 10

In this chapter we discuss three methods to deal with massive terrains (or input datasets).

"Massive" is a vague and undefined term in GIS, and it is continuously changing: 10 years ago a point cloud dataset containing 5 million points was considered massive, while in 2018 it is not. There are some massive datasets even in small areas, eg a Lidar one of Dublin[1], containing around 1.4 billion points with a density of 300pts/m^2, which was collected with airborne laser scanners. The lidar dataset of the Netherlands, AHN[2], has about 10pts/m^2 covering the whole country, thus comprising more than 700 billion points which can be freely downloaded.

For the purposes of this course, we define as "massive" a dataset that does not fit into the main memory of a standard computer, which is usually around 16GB. This definition makes practical sense because working with data outside of the main memory of a computer is substantially slower (about 2 orders of magnitude for solid state drives and 5 for hard drives), causing many standard data processing algorithms to become impractical with massive datasets. Keep in mind that not only the (xyz) coordinates of the points of a point cloud need to be stored, but also often attributes for each point (LAS has several standard ones). Also, in the case of TINs, the geometry of the triangles—and potentially the topological relationships between them—need to be explicitly stored.

What is ironic is that while datasets like AHN3 are being collected in several countries, in practice they are seldom used since the tools that practitioners have, and are used to, usually cannot handle such massive datasets. Indeed, the traditional GISs and terrain modelling tools are limited by the main memory of computers: if a dataset is bigger then operations will be very slow, and will most likely not finish.

1: https://bit.ly/32GXiFq

2: *Actueel Hoogtebestand Nederland* (in Dutch): http://www.ahn.nl

10.1 Raster Pyramids

Raster pyramids are a well-known, standardised, and widely used mechanism to deal with large grids. They are also used for images (and called 'tiled pyramidal images' or 'overview images') and many software support them since they optimise visualisation and thus the speed of a software dealing with large images.

As shown in Figure 10.1, a pyramid means creating recursively copies at lower-resolutions of an original raster (having x columns and y rows), the first copy having a size $(\frac{x}{2}, \frac{y}{2})$, the second $(\frac{x}{4}, \frac{y}{4})$, and so on (the number of images is arbitrary and defined by the user). Notice that the extra storage will be maximum $\frac{1}{3}$ of the original raster: the first pyramid is $\frac{1}{4}$, the second $\frac{1}{16}$, the third $\frac{1}{64}$, etc.

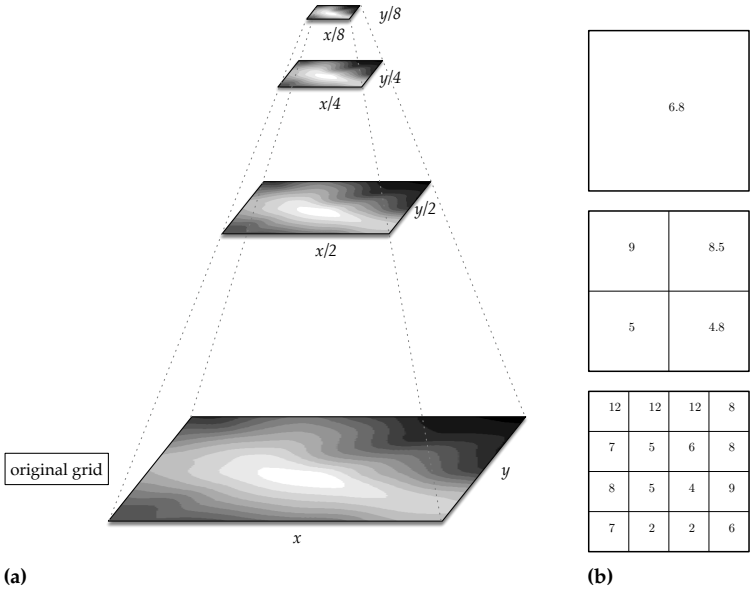


Figure 10.1: (a) The pyramid for a given raster file. **(b)** One 4×4 raster downsampled twice with average-method.

downsampling

Each lower-resolution copy of the raster is obtained with *downsampling*. The most common method is based on averaging the 4 pixels that are merged into one (as shown in Figure 10.1b), but other methods are possible such as nearest neighbour (interpolation method as seen in Chapter 4).

How does it work in practice?

gdaladdo. For certain formats, eg GeoTIFF, the lower-resolutions rasters can be stored directly in the same file as the original raster, and this is standardised. For other formats in GIS, eg the ASCII format '.asc', the pyramids are stored in an auxiliary file with the extension '.ovr', which is actually in TIFF format.

The GDAL utility `gdaladdo` (`https://www.gdal.org/gdaladdo.html`) allows us to create automatically the pyramids for a few formats. The downsampling method can be chosen.

10.2 Indexing points in 3D space with the kd-tree

A k-dimensional tree, kd-tree in short, is a data structure to organise points in a k-dimensional space; it also partitions the space into regions. In the context of terrains, k is in most cases either 2 or 3. Notice that in practice we would never say a "2d-tree" or a "3d-tree": we call them "kd-tree of dimension 2 (or 3)".

As shown in Figure 10.2, a kd-tree is a binary tree (thus each node has a maximum of 2 children, if any), and the main idea is that each level of the tree compares against one specific dimension. We 'cycle through' the dimensions as we walk down the levels of the tree.

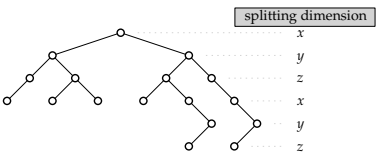


Figure 10.2: Example of kd-tree in 3D, with the dimensions used at each level.

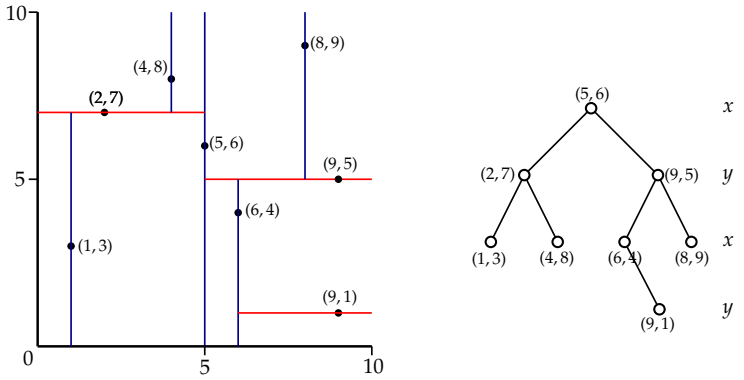


Figure 10.3: Example of kd-tree for 8 points in the $\mathbb{R}^2$.

Let S be a set of points in $\mathbb{R}^k$, and let κ be the kd-tree of dimension k of S. Each point p_i in S is a node of κ. A node implies a hyperplane that divides the space into 2 halfspaces according to one dimension; the hyperplane is perpendicular to the dimension of the node (which is linked to the level in the tree). Points with a lower coordinate value than the node along that dimension (corresponding to 'left' or 'under' the hyperplane) are put into the left subtree of the node, and the other ones into the right subtree.

Consider the kd-tree in 2D in Figure 10.3. The first dimension splits the data into 2 halfplanes along the line $x = 5$, then each of these halfplanes is independently split according to the y dimension (with the lines $y = 7$ and $y = 5$), then the 4 regions are split according to the x dimension, and so on recursively.

Construction of a kd-tree. In theory, any point could be used to divide the space according to each dimension, and that would yield a valid kd-tree. However, selecting the *median* point creates a *balanced* binary tree, which is desirable because it will improve searching and visiting the tree (see below). The tree in Figure 10.3 is balanced, but if for instance $(1, 3)$ had been selected as the root, then there would be no children on the left, and all of them would be on the right.

binary tree

The median point is the one whose value for the splitting dimension is the median of all the points involved in the operation. This implies that to construct the kd-tree of a set S of n points, as a first step n values need to be sorted, which is a rather slow operation. In practice, most software libraries will not sort n values, but rather sample randomly a subset of them (say 1%), and then use the median of this subset as the splitting node in the graph. While this does not guarantee a balanced tree, in practice the tree should be close to balanced.

The tree is built incrementally, ie points are added in the tree one after the other, and after each insertion the tree is updated. Each insertion is simple: traverse the tree starting from the root, go left or right depending on the splitting dimension value, and insert the new point as a new leaf in the tree. Figure 10.4 illustrates this for one point.

Observe that this insertion renders the tree unbalanced. Methods to balance a kd-tree exists but are out of scope for this course.

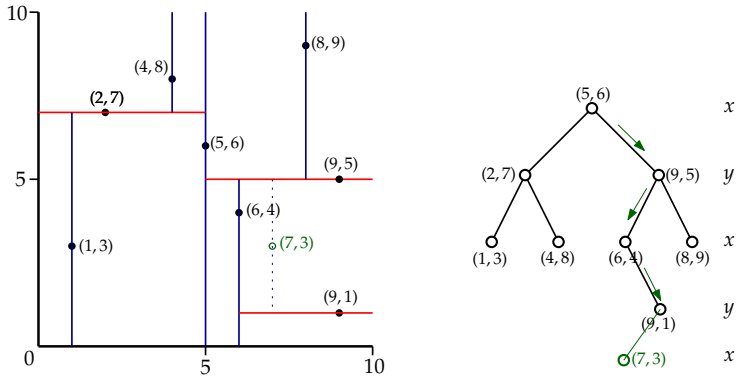


Figure 10.4: Insertion of a new point $(7, 3)$ in a kd-tree.

Nearest neighbour query in kd-trees. The nearest neighbour query aims to find the point c in a set S that is the nearest (according to the Euclidean distance) to a query point q. It can be performed brute-force (comparing distance to all points in S), but this is slow. An alternative is to construct the Voronoi diagram (actually the Delaunay triangulation), and navigate in the cells; this works but is in practice not as efficient as using a kd-tree.

First observe that the obvious method to find the cell in the kd-tree containing q does not work because q can be far away in the tree. Figure 10.5a illustrates this: c is $(6, 4)$ but is located in the right subtree of the root, while q is in the left subtree.

The idea of the algorithm we are presenting here is to traverse the whole tree (in depth-first order), but use the properties of the tree to quickly eliminate large portions of the tree. The eliminated subtrees are based on their bounding boxes. As we traverse the tree, we must keep track of the closest point c_{temp} so far visited.

The algorithm starts at the root, stores the current closest point c_{temp} as the root, and visits the nodes in the tree in the same order as for the insertion of a new point. This order is the one that is *most promising*, because we expect c to be close to the insertion location (albeit this is not always the case). At each node n_i it updates c_{temp} if it is closer. For this, the Euclidean distance is used. For the example in Figure 10.5b, point $(5, 6)$ is the first c_{temp}, and then although $(2, 7)$ and $(1, 3)$ are visited, neither is closer and thus after that step $c_{temp} = (5, 6)$.

The algorithm then recursively visits the other subtrees, and checks whether there could be any points, on the other side of the splitting hyperplane, that are closer to q than c_{temp}. The idea behind this step is that most of the subtrees can be eliminated by verifying whether the region of the bounding box of the subtree is closer than the current $dist(q, c_{temp})$, $dist()$ being the Euclidean distance between 2 points. If that distance is shorter, then it is possible that one point in the subtree is closer than c_{temp}, and thus that subtree must be visited. If not, then the whole subtree can be skipped, and the algorithm continues.

Figure 10.5c shows this idea after $(1, 3)$ has been visited. c_{temp} is $(5, 6)$, and we must decide whether the subtree right of $(2, 7)$ must be visited. In this case it must not be visited because the bounding box (light blue region) is 3.0unit from q, and $dist(q, c_{temp})$ is around 2.07; it is thus impossible that one point inside the subtree be closer than $(5, 6)$.

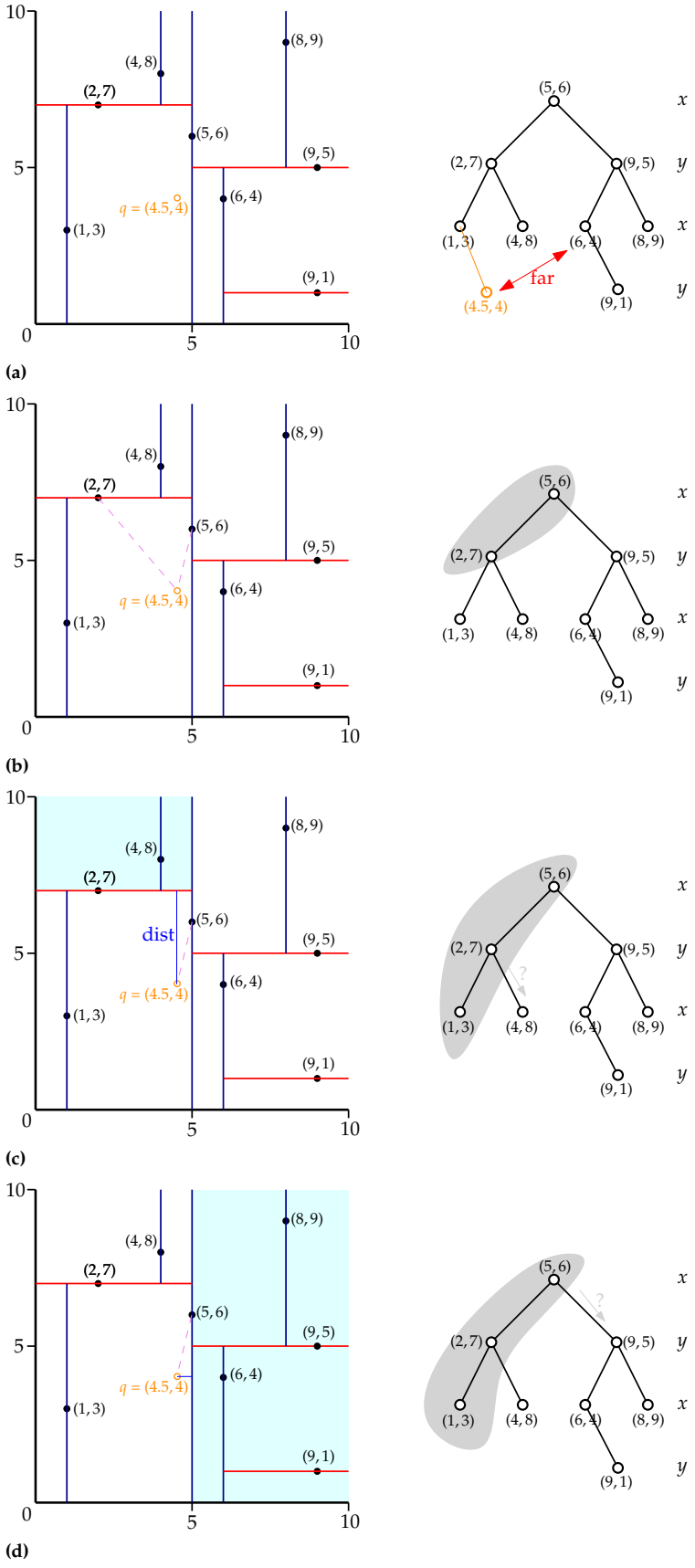


Figure 10.5: Several states for the nearest neighbour query based on a *k*d-tree.

The next step is verifying whether the subtree right of the root could contain a point closer than c_{temp}. In the Figure 10.5d, this is possible since the bounding box is only 0.5unit from q, and thus the subtree must be visited.

The algorithm continues until all subtrees have either been visited or eliminated. At the end, c is $(6, 4)$.

Time complexity. To insert a new point, and to search for a nearest neighbour, the time complexity on average is $\mathcal{O}(\log n)$. The tree stores one node per point, thus the space complexity is $\mathcal{O}(n)$.

***m*-closest neighbours.** The algorithm can be extended in several ways by simple modifications. It can provide the m nearest neighbours to a point by maintaining m current closest points instead of just one. A branch is only eliminated when m points have been found and the branch cannot have points closer than any of the m current bests.

10.3 Streaming paradigm to construct massive TINs and grids

The incremental construction algorithm for the Delaunay triangulation, presented in Chapter 3, will not work if the size of the input dataset is larger than the main memory. Or if it works, it will be very slow.

To deal with massive datasets, one can also design external memory algorithms. These basically use disks to store temporarily files that do not fit in memory, and instead of using the mechanism of the operating system, they design explicit rules for the swapping of data between the disk and the memory. The main drawbacks of this approach are that the design of such algorithms is rather complex, and that for different problems different solutions have to be designed.

spatial streaming

An alternative approach to dealing with massive datasets is *spatial streaming*, which mixes ideas from external memory algorithms with different ways to keep the memory footprint very low. The basic idea of this paradigm is that of a *streaming mesh*: a format for representing triangulations (or meshes) as a set of interleaved vertices, triangles and *vertex finalization tags* that indicate when a vertex will not be used anymore. Standard mesh formats do not use finalization and can therefore suffer if the mesh is larger than memory. These tags allows us to keep in memory only a small part of a large dataset.

spatial coherence

A streaming mesh basically documents the *spatial coherence* of a dataset, which Isenburg et al. (2006b) defines as: "a correlation between the proximity in space of geometric entities and the proximity of their representations in [the file]". They also demonstrate that real-world point cloud datasets often have natural spatial coherence and they exploit this coherence to compute Delaunay triangulations of massive datasets (instead of reordering the points, which is expensive); this coherence is expected since LiDAR samples are often stored in the order they were collected.

The ideas behind streaming are very useful for certain *local* problems (eg interpolation and creation of grids), but unfortunately they cannot be used directly (or it would be extremely challenging) for *global* problems such as visibility or flow modelling.

To read or to watch.

This video explains how the streaming concepts can be applied to constructing the Delaunay triangulation of massive datasets. You do not need to read the full paper, which is Isenburg et al. (2006b).

https://youtu.be/DRCGTF2y_tM

To read or to watch.

M. Isenburg et al. (2006a). Generating Raster DEM from Mass Points Via TIN Streaming. *Geographic Information Science—GIScience 2006*. Vol. 4197. Lecture Notes in Computer Science. Münster, Germany, pp. 186–198

PDF: http://dx.doi.org/10.1007/11863939_13

The article summarises Isenburg et al. (2006b) (you do not need to read it), and shows how large rasters can be constructed with spatial interpolation.

10.4 Notes & comments

The description of the kd-tree and the nearest neighbour query is adapted from Wikipedia (https://en.wikipedia.org/wiki/K-d_tree) and the lecture notes entitled "kd-Trees—CMSC 420" from Carl Kingsford (available at https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/kdtrees.pdf).

Vitter (2001) provides an overview of external algorithms..

10.5 Exercises

1. Isenburg et al. (2006a) argues that real-world point cloud datasets often have natural spatial coherence. Explain why that is for lidar datasets.
2. Given a simple point clouds stored in a CSV file (one x, y, z per line), how many passes over the file the triangulator of Isenburg et al. (2006a) make? What does each do?
3. How to construct a kd-tree that is as balanced as possible?
4. "The ideas behind streaming are very useful for certain *local* problems, but unfortunately they cannot be used directly for *global* problems such as visibility or flow modelling". Explain why that is with a concrete example.

11 Visibility queries on terrains

Several applications using terrains involve *visibility queries*, ie given a viewpoint, which part(s) of the surrounding terrain are visible. Examples of such applications are many: optimal position of telecommunication towers, path planning for hiking (to ensure the nicest views), estimation of the view for scenic drives, estimation of visual damage when trees in a forest are cut, etc. There are also several related problems. One example is the estimation of shadows (position of the sun continually varies, also with seasons). Shadows are important to accurately measure the photovoltaic potential, for determining solar envelopes, for assessing the value of real estate, and for estimating the thermal comfort, among other applications.

When referring to visibility problems, we address the following two fundamental problems:

line-of-sight (LoS): given a viewpoint v and another point q, does v sees q (and vice-versa)? Or, in other words, does the segment vq intersects the terrain? The result is either True or False.

viewshed: given a viewpoint v, which part(s) of the surrounding terrain are visible? The result is a polygon (potentially disconnected) showing the locations and extent of what is visible from v. Usually the extent is limited to a certain "horizon", or radius of visibility. If the terrain is formed of different objects (eg buildings), an object is either visible or not (simple case), or parts of objects can be visible (more complex).

Observe that for both problems, the viewpoint can either be directly on the terrain (at relative elevation 0m) or at a given height (2m for a human, or 30m for a telecommunication tower).

We discuss in this chapter the general problem of visibility as defined in computer graphics, and then discuss how terrains, being 2.5D surfaces, simplify the problem. We discuss how to solve these problems for both TINs and grids.

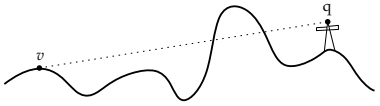


Figure 11.1: Line-of-sight between v and q; q is not visible.

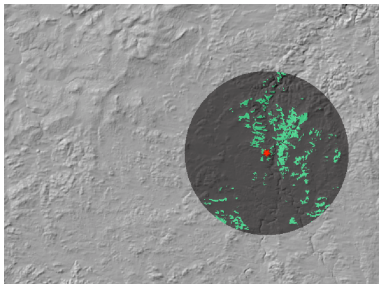


Figure 11.2: The viewshed at the location marked with a red star (green = visible; maximum view distance (dark grey) is set to 15km).

11.1 Rendering + ray casting

Rendering is the process of generating images from 2D or 3D scenes. Without going into details, as shown in Figure 11.3, it involves projecting the (3D) objects in a scene to an image (say 800x800 pixels) and assigning one colour to each pixel. The colour of a pixel is that of the closest object, but to obtain photorealistic images, lighting, shading, and other physics-based functions are often applied (this goes beyond the scope of this course!).

Ray casting is used for each pixel: a ray is defined between the viewpoint p and the centre of the pixel, and the closest object in the scene must be

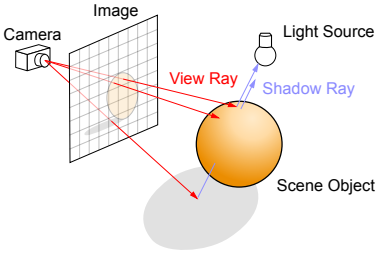


Figure 11.3: Ray tracing builds the image pixel by pixel by extending rays into the scene. (Figure from https://commons.wikimedia.org/wiki/File:Ray_trace_diagram.svg)

found. The main issue involves finding that closest object, and especially discard the other objects lying behind it.

To read or to watch.

Parts of the following chapter. It summarises different methods to determine which surfaces are visible, for the generic cases of objects in 3D space. Read only the following sections: 18.0, 18.1, 18.2, and 18.4.

D. Salomon (2011). Visible Surface Determination. *Texts in Computer Science*. Springer London, pp. 891–910

PDF: https://doi.org/10.1007/978-0-85729-886-7_18

11.2 For 2.5D terrains, the problem is simpler

The problem is simplified for terrains because a terrain is a 2.5D surface, and we can convert the problem to a 2D one. Furthermore, we can exploit the connectivity and adjacency between the 2D cells forming a terrain to minimise the number of objects to test (for intersections and for determining who is in front of who).

11.2.1 Visibility in TINs

To read or to watch.

Parts of the following chapter. Read from Section 1 to Section 3.1; it covers the simplest case for a LoS between 2 points on the surface.

M. de Berg (1997). Visualization of TINs. *Algorithmic Foundations of Geographic Information Systems*. Ed. by M. van Kreveld et al. Vol. 1340. Lecture Notes in Computer Science. Berlin: Springer-Verlag, pp. 79–97

PDF: https://doi.org/10.1007/3-540-63818-0_4

11.2.2 Visibility in grids

Solving visibility queries in grids is simpler than with triangles since the topology of the grid is implied (we have direct access to the neighbours of a given cell), and because grid cells are usually small we can assume that a grid cell is visible (or not) if its centre is visible (or not). The same assumption is tricky for triangles, since these can be large; in practice it is often assumed that a triangle is visible if its 3 vertices are visible, but this varies per implementation.

We describe here how both LoS and viewshed queries can be implemented for grids; the same principles could be applied to TINs with minor modifications.

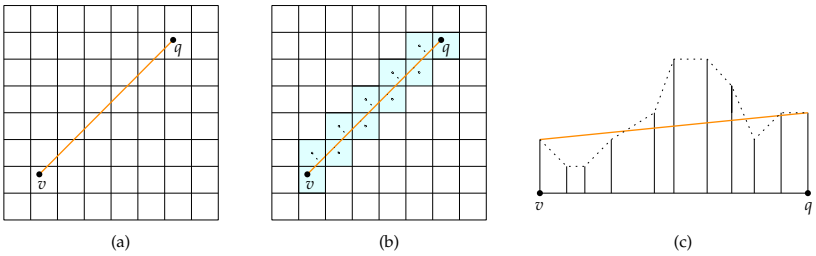


Figure 11.4: Line-of-sight for between v and q. Observe that along the profile, the points with elevation are not equally spaced.

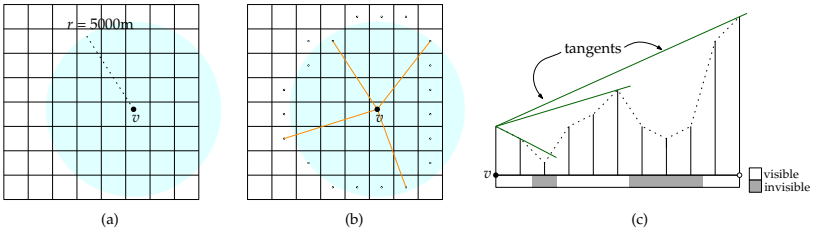


Figure 11.5: Viewshed for the point v; the blue circle is the radius of the horizon (5000km in this case).

Line-of-sight. A LoS query, between a viewpoint v and another point q, implies reconstructing the profile of the terrain along the vertical projection of vq (let us call it vq_{xy}). It then suffices to follow vq and verify whether the elevation at any (x, y) location along the profile is higher than that of vq. As shown in Figure 11.4, since the terrain is discretised into grid cells, there are 2 options to reconstruct the profile between v and q:

1. identify all the cells intersected by vq_{xy}, and assign the centre of each cell by projecting it to the terrain profile. This is what is done in Figure 11.4.
2. consider the edges of the cells, collect all the edges that are intersected by vq_{xy}, and linearly interpolate the elevations. This is far more expensive to compute, and therefore less used in practice.

The algorithm is thus as follows. Start at v, and for each pixel c encountered along vq_{xy}, verify whether the elevation value of vq at that location is higher than the elevation of c. If it is, then continue to the next pixel; if not, then there is an intersection and thus the visibility is False. If the pixel containing q is reached without detecting an intersection, then the visibility is True.

Viewshed. As shown in Figure 11.5, computing the viewshed from a single viewpoint v implies that the LoS between v and the centre of each pixel in a given radius is tested. The result of a viewshed is a binary grid; in Figure 11.2, True/visible pixels are green, and False/invisible ones are dark grey.

While this brute-force approach will work, several redundant computations will be made, since several of the rays from v will intersect the same grid cells. Furthermore, depending on the resolution, the number of cells in a 5km radius (a reasonable value where humans can see) can become *very* large. As an example, with the AHN3 gridded version (50cm resolution), this means roughly 400M queries ($(\frac{5000\times2}{0.5})^2$).

One alternative solution is shown in Figure 11.5b: it involves sampling the grid cells intersecting the border of the visible circle (obtaining several centres q_i), and computing the visibility of each of the cells along the line segment vq_i as we 'walk' from v. Observe that, along vq_i, it is possible

that a point far from v is visible, while several closer points are not; Figure 11.5c gives an example.

One solution involves using so-called *tangents*. The current tangent t_{cur} is first initialised as a vector pointing downwards. Then, starting at v, we walk along the ray vq_i, and for each cell intersected its elevation z is compared to the elevation of t_{cur} at that location. If z is lower, then the cell is invisible. If z is higher, then the cell is visible and t_{cur} is updated with a new tangent using the current elevation.

Viewsheds with several viewpoints v_i are also very useful, think for instance of obtaining the viewshed along a road. This can be computed by sampling the road at every 50m and computing the viewsheds from each of the points. Each viewshed yields a binary grid, and it suffices to use a map algebra operator to combine the results into one grid (if one cell is visible from any viewpoint, then it is visible).

11.3 Notes & comments

The 'tangent algorithm' to compute viewsheds was first described by Blelloch (1990).

The description here is inspired by that of De Floriani and Magillo (1999).

11.4 Exercises

1. Explain why the spacing in Figure 11.4c along the profile has points that are not equally spaced.
2. Your are given a 2.75D terrain of an area, it is composed of triangles, and your aim is to perform line-of-sight queries between some locations. Describe the algorithm that you will implement to perform the queries.

Point cloud processing 12

In this chapter we discuss algorithms for reading, transforming, processing, cleaning and extracting information from point clouds, and we also briefly present the main storage formats used in practice.

12.1 Point cloud file formats

A point cloud is essentially an array of 3D points, and often that is also how it is stored in a file. Regardless of the format, a point cloud file can often be seen as an array of *point records*, each of which contains the coordinates and attributes of one point.

A point record consists of several *fields*, each of which stores a single value, eg an integer, float, or boolean. A field can for instance represent the x-, y-, or z-coordinate of a point or one of its attributes, eg the lidar return number or colour information. The order and meaning of the fields in a record are fixed for all the point records in one file. How exactly the point records are structured and stored in the file, and what additional metadata are available, depends on the specific file format that is used.

Notice that, in additions to the widely used formats mentioned here, there are also many proprietary formats. These are often specific to one particular software and are therefore not very useful for data exchange.

12.1.1 ASCII formats

ASCII formats are plain text files. The point cloud information is thus stored as a sequence of ASCII characters, usually one point record per line. In most cases you can recognise such files by the *.xyz*, *.csv*, or *.txt* extensions; these are in most cases comma-separated values (CSV) files *. A benefit of ASCII files is that you can simply open and edit them in a text editor. The biggest downside is that they are not standardised, ie the type, order, and number of attributes varies and also the used coordinate reference system (CRS) are usually not documented in the file.

```
x y z
84499.948 446610.324 0.407
84499.890 446609.862 0.434
84499.832 446609.420 0.442
84499.777 446608.987 0.454
84499.715 446608.528 0.444
84499.839 446612.808 0.493

```

Figure 12.1: An example of a CSV file used to store the xyz coordinates of a point cloud.

* https://en.wikipedia.org/wiki/Comma-separated_values

```
ply
format ascii 1.0  ← encoding and ply version number
comment This is an example file for GEO1015!
element vertex 7  ← number of points, start of point record definition
property float x
property float y                          point record definition
property float z
property int custom_attribute
end_header
91443.89 438385.69 -0.80 11
91443.94 438386.10 -0.78 43
91444.00 438386.51 -0.79 44
91444.06 438386.94 -0.83 31   point records
91444.11 438387.36 -0.86 31
91443.88 438383.50 -0.83 22
91443.93 438383.91 -0.80 65
```

Figure 12.2: A simple PLY file with 1 additional user-defined attribute of type integer ("int"). It contains 7 points.

12.1.2 The PLY format

The PLY format can be considered a standardised ASCII format. A PLY file contains a header[†] that specifies the structure of the point records in the file, ie the number of attributes (called *properties*), their order, their names and their data types. This makes it a very flexible standard, since the user can decide on the composition of the point record. Figure 12.2 shows an example PLY file.

PLY files are readable by many software packages and can also be stored in a binary encoding[‡]. Compared to the ASCII encoding, the binary encoding results in a smaller file size and quicker reading and writing from and to the file. There is no standardised way to specify the CRS in a PLY file, although one could add a comment in the header stating the CRS.

12.1.3 The LAS format

The public LASER (LAS) file format is the most widely used standard for the dissemination of point cloud data. The LAS standard, currently at version 1.4, is maintained by the ASPRS organisation and, as the name implies, it was designed for datasets that originate from (airborne) lidar scanners. However, in practice it is also used for other types of point cloud, eg those derived from dense image matching. It is a binary-encoded standard and compared to the PLY format it is rather strict because it prescribes exactly what a point record should look like, ie what attributes are present and how many bits each attribute must use.

Table 12.1 shows the composition of the simplest record type that is available for LAS files. Other record types are available that also include fields to store for instance RGB colour information or the GPS time (the time a point was measured by the scanner), but all records types include at least the fields shown in Table 12.1. While the LAS standard clearly specifies that all these fields are required, some of the fields are very specific to lidar acquisition and they are sometimes ignored in practice, eg if a point cloud originating from dense matching is stored in the LAS

[†] A header is supplemental information placed at the beginning of a file, eg to store metadata about the file.

[‡] https://en.wikipedia.org/wiki/PLY_(file_format)#ASCII_or_binary_format

Table 12.1: LAS Point Data Record Format 0

Field	Format	Length (bits)	Description
X	int	32	X coordinate.
Y	int	32	Y coordinate.
Z	int	32	Z coordinate.
Intensity	unsigned int	16	The pulse return amplitude.
Return number	unsigned int	3	The total pulse return number for a given output pulse.
Number of returns	unsigned int	3	Total number of returns for a given pulse
Scan Direction Flag	boolean	1	Denotes the direction at which the scanner mirror was traveling at the time of the output pulse. A bit value of 1 is a positive scan direction, and a bit value of 0 is a negative scan direction (where positive scan direction is a scan moving from the left side of the in-track direction to the right side and negative the opposite).
Edge of Flight Line	boolean	1	Has a value of 1 only when the point is at the end of a scan. It is the last point on a given scan line before it changes direction.
Classification	unsigned int	5	Classification code
Scan Angle Rank	int	4	The angle at which the laser pulse was output from the scanner including the roll of the aircraft.
User Data	unsigned int	4	May be used at the user's discretion.
Point Source ID	unsigned int	8	Indicates the file from which this point originated. Non-zero if this point was copied from another file.

format. Notice that unused fields will still take up storage space in each record.

The CRS of the point cloud can be stored in the header of a LAS file, together with some other general information such as the total number of points and the bounding box of the point cloud. The X, Y, and Z fields are stored as 32-bit integers. To convert these values to the actual coordinates on the ground, they need to be multiplied by a scaling factor and added to an offset value, ie:

$$\begin{aligned} X_{coordinate} &= (X_{record} * X_{scale}) + X_{offset} \\ Y_{coordinate} &= (Y_{record} * Y_{scale}) + Y_{offset} \\ Z_{coordinate} &= (Z_{record} * Z_{scale}) + Z_{offset}. \end{aligned}$$

The scaling factors X_{scale}, Y_{scale}, Z_{scale} and the offsets X_{offset}, Y_{offset}, Z_{offset} are also given in the header. Notice that the scaling factor determines the number of decimals that can be stored, eg the factors 10, 100, and 1000 would give us 1, 2, and 3 decimals respectively.

The LAS standard defines several classification codes, as listed in Table 12.2. These codes are to be used as values for the classification field of a point record, and are intended to indicate the type of object a point belongs to. Which classes are used strongly depends on the dataset at hand. The codes 0 and 1 may appear ambiguous, but there is a clear distinction. To be exact, the code 0 is used for points that never subject to a classification algorithm, whereas the code 1 is used for points that have been processed by a classification algorithm, but could not successfully be assigned to a class. It is possible to define your own classes using code ranges that are reserved for that purpose.

Table 12.2: The first 10 LAS classification code numbers. More codes exist, but they are not listed here.

Code	Meaning
0	never classified
1	unclassified
2	ground
3	low vegetation
4	medium vegetation
5	high vegetation
6	building
7	low point (noise)
8	*reserved*
9	water

Figure 12.3: Classification codes used in the AHN3 dataset.

AHN3 classification The national Dutch AHN3 lidar dataset is disseminated in the LAZ format and uses the LAS classification codes. Figure 12.3 shows all the codes that are used in AHN3. Notice that apart from the pre-defined codes from Table 12.2, it also uses the custom code 26 for an 'artefact' (Dutch: *kunstwerk*) class that includes special infrastructural works such as bridges and viaducts. The points in the unclassified class (1) typically belong to vegetation, street furniture or cars.

The LAZ format Finally, a compressed variant of the LAS format, dubbed the LAZ format, exists. While it is not maintained by an 'official' organisation like the LAS standard, it is an open standard and it is widely used, especially for very big dataset. Through the use of lossless compression algorithms that are specialised for point cloud data, a LAZ file can be packed into a fraction of the storage space required for the equivalent LAS file without any loss of information. This makes it more effective than simply using ZIP compression on a LAS file. In addition support for LAZ is typically built into point cloud reading and writing software, so to the user it is no different than opening a LAS file (although the compression does take some time).

The LAZ format closely resembles the LAS format, ie the header and the structure of the point records are virtually identical. In a LAZ file the point records are grouped in blocks of 50,000 records each. Each block is individually compressed, which makes it possible to partially decompress only the needed blocks from a file (instead of always needing to compress the whole file). This can save a lot of decompression computations if only a few points from a huge point cloud are needed. Also notice that the effectiveness of the compression algorithms depends on the similarity in information between subsequent point records. Typically information is quite similar for points that are close to each other in space. Therefore, a greater compression factor can often be achieved after spatially sorting the points.

In practice, for the AHN3 dataset, the LAZ file of the same area will about 10X more compacter then its LAS counterpart.

12.2 Thinning

A point cloud with fewer points is easier to manage and quicker to visualise and process. Therefore a point cloud is sometimes *thinned*, which simply means that a portion of the points is discarded and not used for processing. Commonly encountered thinning methods in practice are:

random randomly remove a given percentage of the points.

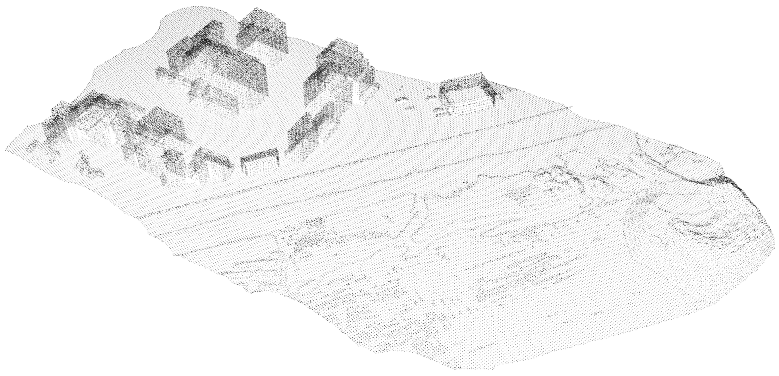
nth-point iterate through the points and keep only the first point for every n points. For example a dataset with 1000 points is reduced to 100 points if $n = 10$. This is the quickest thinning method.

grid Overlay a 2D or 3D regular grid over the point cloud and keep one point per grid cell. That can be one of the original points, an average of those, or the exact centre of the cell. The thinning factor depends on the chosen cell-size. Notice that the result is often a point cloud with a homogeneous point density on all surfaces (only on the horizontal surfaces if a 2D grid is used).

See Figure 12.4 for a comparison between random thinning and grid thinning.

From Section **??** you undoubtedly remember that TIN simplification has a somewhat similar objective: data reduction. However, for a given number of resulting points, TIN simplification yields a higher quality end result because it only removes points that are deemed unimportant. Thinning methods on the other hand do not consider the 'importance' of a point in any way, and might discard a lot of potentially meaningful details. So why bother with thinning? The answer is that thinning methods are a lot faster since they do not require something like a computationally expensive triangulation. Especially in scenarios where the point density is very high and the available time is limited, thinning can be useful.

(a) random thinning



(b) 3D grid thinning

Figure 12.4: Comparison of two thinning methods. The thresholds were chosen such that the number of remaining points is approximately the same.

12.3 Outlier detection

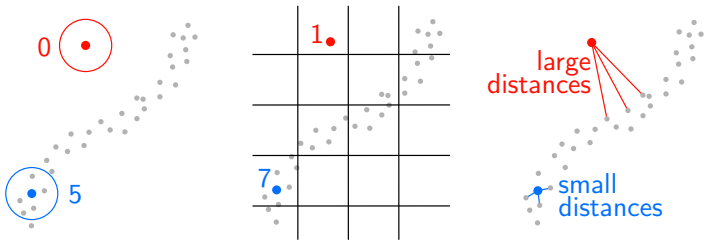
Recall from Chapter 2 that outliers are points that have a large error in their coordinates. Outliers are typically located far away from the terrain surface and often occur in relatively low densities. Outlier detection aims to detect and remove outliers and is a common processing step for point clouds.

Most outlier detection methods revolve around analysing the local neighbourhood of a point. The neighbourhood can be defined using a k-nearest neighbour (knn) search (see Section 10.2), a fixed radius search, or by superimposing a regular grid on the point cloud and finding the points that are in the same grid-cell. The points that are determined to be in the neighbourhood of a point of interest p are used to determine whether p is an outlier or not.

The underlying assumption of most outlier detection methods is that an outlier is often an isolated point, ie there are not many points in its neighbourhood. We distinguish the following outlier detection methods (see also Figure 12.5):

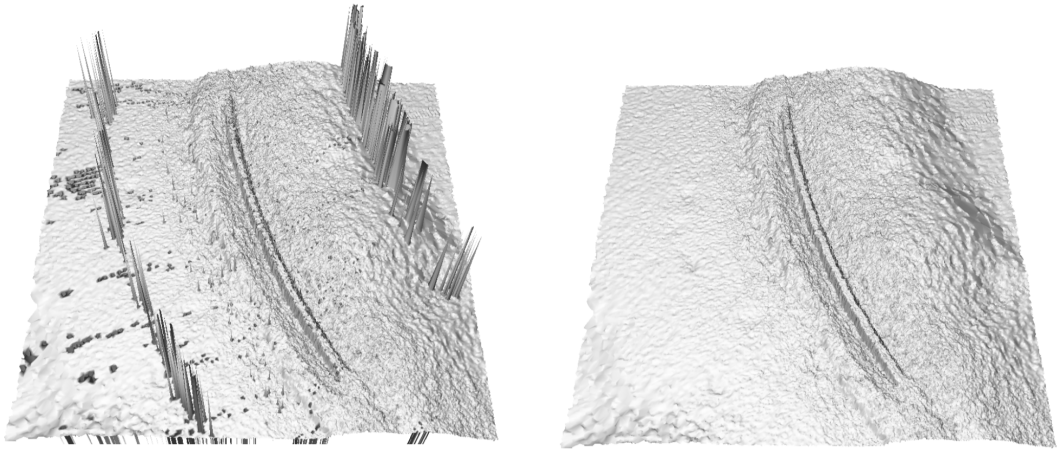
radius count count the number of points that are within a fixed radius from p. If the count is lower than a given threshold, p is marked as an outlier.

grid count Superimpose a grid on the point cloud and count for each grid-cell the number of points. If the count is lower than a given threshold, the points inside the corresponding grid cell are marked as outliers. Sometimes the neighbourhood is extended with adjacent



(a) radius count **(b)** grid count **(c)** knn distance ($k = 3$)

Figure 12.5: Three outlier detection methods based on local point density. The red point is an outlier, whereas the blue point is an inlier.



(a) before outlier detection **(b)** after outlier detection

Figure 12.6: Outlier detection in a multi-beam echo sounding dataset using a TIN (Arge et al., 2010).

grid cells. The grid method has the advantage that it can be used with the spatial streaming paradigm (see Section 10.3).

knn distance Find the k nearest neighbours of p, eg using a kd-tree, and compute the mean or median of the distances between p and its neighbours. If this value is above a given threshold, p is marked as an outlier.

These methods generally work well if the outliers are isolated. However, in some cases this assumption does not hold. For example in case of a point cloud derived from multi-beam echo sounding[§] a common issue is the occurrence of (shoals of) fish. These fish cause large groups of points that are clustered closely together above the seafloor. These are not isolated points since each outlier will have plenty of other points nearby. A possible solution is to construct a TIN of all points and to 'cut' the relatively long edges that connect the outlier clusters to the seafloor. This splits the TIN into several smaller TINs, and the largest of those should then be the seafloor surface without the outliers. Figure 12.6 gives an example.

12.4 Ground filtering

Ground filtering involves classifying the points of a point cloud into ground points and non-ground points. Ground points are those points that are part of the bare-earth surface of the earth, thus excluding

[§] See Section **??**.

Figure 12.7: Cross-section of a terrain with lamp posts and trees before (top) and after (bottom) ground filtering (Axelsson, 2000).

vegetation and man-made structures such as buildings and cars. The ground points can then be used to generate a DTM, usually as a TIN or a raster. Or, the non-ground points can be used as input for another classifier, eg to classify buildings and vegetation possibly using a region growing algorithm (see Section 12.5.2).

Ground filtering methods are typically based on the assumptions that

1. the ground is a continuous surface without sudden elevation jumps,
2. for a given 2D neighbourhood, the ground points are the ones with the lowest elevation.

This is reasonable because non-ground points are typically measurements from objects above the ground such as trees, street furniture and buildings (see eg Figure 12.7).

Notice that the resulting bare-earth model may thus have holes where these non-ground objects used to be. If needed, these holes can be filled in a subsequent processing step involving spatial interpolation.

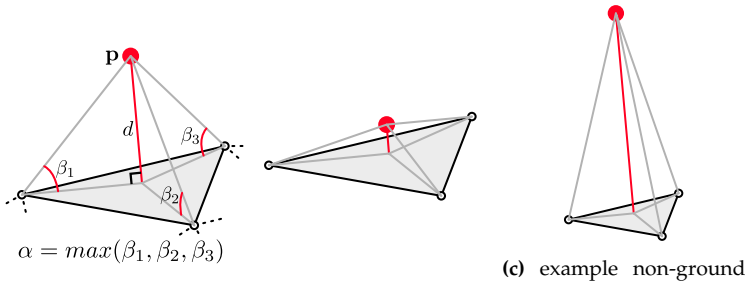
12.4.1 Ground filtering with TIN refinement

We will now discuss an effective ground filtering method that is based on the *greedy* insertion of ground points into a TIN. Indeed, the same algorithmic paradigm of iterative TIN refinement that we saw earlier in Section **??** is used. The algorithm consists of three main steps:

1. construction of a rudimentary initial TIN (usually a Delaunay TIN);
2. computation of two geometric properties for each point that is not already labelled as ground;
3. incremental insertion of points that pass a simple and local 'ground test' based on the computed geometric properties.

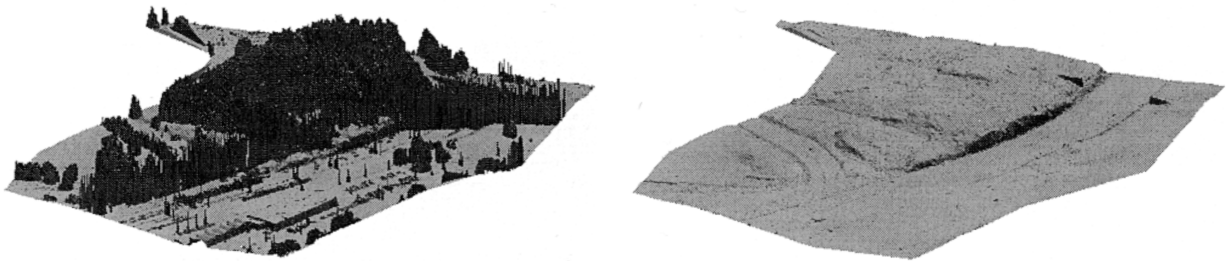
The latter two steps are repeated until all remaining points fail the ground test.

In the first step a rudimentary initial TIN is constructed from a number of points that have locally the lowest elevation and are spread somewhat evenly over the data extent. These points are found by superimposing a 2D grid over the data extent and by selecting the lowest point for each grid cell (similar to grid thinning). The cell-size of the grid should be chosen such that it is larger than the largest non-ground object (usually a building). Thus, if the largest building has a footprint of 100x100m, the cellsize should be a bit larger, eg 110m, so that it is guaranteed that each grid-cell has at least a few ground points. Each point that is inserted into the TIN is considered to be a ground point.



(a) **(b)** example ground point **(c)** example non-ground point

Figure 12.8: Geometric properties for a point p in the method for ground filtering based on TIN refinement.



(a) Before **(b)** After

Figure 12.9: Ground filtering (Axelsson, 2000)

In the second step two geometric properties are computed for each unclassified point. These properties are based on the relation between the point p and the triangle in the current TIN that intersects its vertical projection. The two properties are illustrated in Figure 12.8a. The first property, denoted d, is the perpendicular distance between the p and the plane spanned by the triangle. The second property, denoted α, is the largest angle of the angles between the triangle and the three vectors that connect each vertex with p.

In the ground test of the final step, it is simply checked for each point if its d is below a given threshold d_{max} and if its α is below a given threshold α_{max}. If this is indeed the case, the point is labelled as a ground point and inserted into the TIN. Compare Figures 12.8b and 12.8c.

Of course, if the triangles in the TIN change, the properties of the overlapping unclassified points need to be recomputed. However, the algorithm is greedy, which means that it never "goes back" on operations that were previously performed, and thus when a point p is inserted as the ground, it is never removed. When all remaining points fail the ground test, the algorithm terminates. Figure 12.9 gives an example result.

Time complexity. The worst case time complexity is $\mathcal{O}(n^2)$, since the most expensive term is related to the last two steps of the algorithm; for each of the n points we need to do the ground test, and after each test we need to recompute the geometric properties of at most n points.

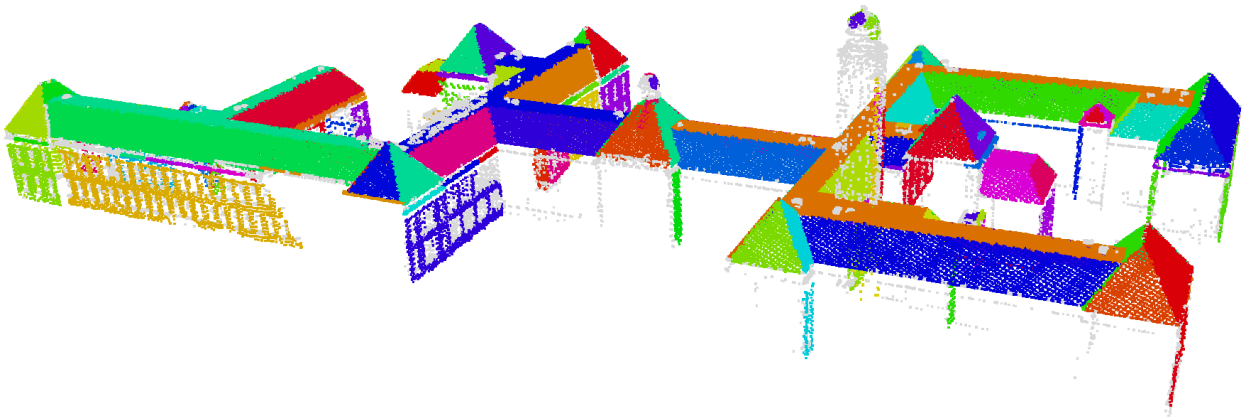


Figure 12.10: Planar regions in the AHN3 point cloud. Each region was assigned a random colour.

12.5 Shape detection

Using shape detection we are able to automatically detect simple shapes such as planes in a point cloud. See for example Figure 12.10, where the points are randomly coloured according to the corresponding planar surfaces. Shape detection is an important step in the extraction and reconstruction of more complex objects, eg man-made structures such as buildings are often composed of planar surfaces.

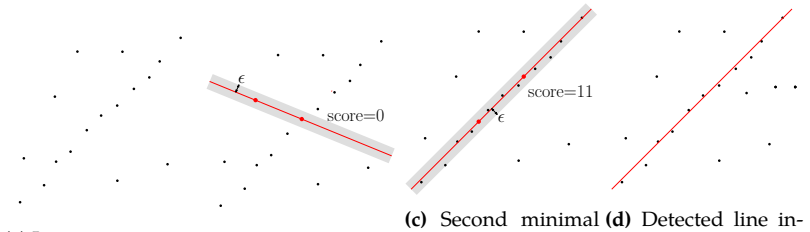
In this section, three shape detection methods will be introduced: RANSAC, region growing, and Hough transform.

But, first we will declare some common terminology. Let P denote a point cloud. If we perform shape detection on P we aim to find a subset of points $S \subset P$ that fit with a particular shape. Most shape detection methods focus on shapes that can be easily parametrised, such as a line, a plane, or a sphere. If we specify values for the parameters of such a parametrised shape, we define an *instance* of that shape. A line for example can be parametrised using the equation $y = mx + b$, in this case m and b are the parameters. We can create an instance of a line by specifying values for its parameters m and b, respectively fixing the slope and the position of the line.

In the following, the methods are described in a general way, ie without specialisations for one particular shape. Only for illustrative purposes specific shapes such as a line or a plane are used to (visually) explain the basic concept of each shape detection method, but the same could be done with spheres, cones, or others.

12.5.1 RANSAC

RANSAC is short for RANdom SAmpling Consensus and as its name implies it works by randomly sampling the input points. In fact it starts by picking a random set of points $M \subset P$. This set M is called the *minimal set* and contains exactly the minimum number of points that is needed to uniquely construct the shape that we are looking for, eg 2 for a line and 3 for a plane. From the minimal set M the (unique) shape instance $\mathcal{I}$ is constructed (see Figures 12.11b and 12.11c). The algorithm then checks for each point $p \in \{P \setminus M\}$ if it fits with $\mathcal{I}$. This is usually



(a) Input points **(b)** First minimal set **(c)** Second minimal set **(d)** Detected line instance

Figure 12.11: RANSAC for line detection ($k = 2$ iterations)

done by computing the distance d from p to $\mathcal{I}$ and comparing d against a user-defined threshold ϵ. If $d < \epsilon$ we say that p is an *inlier*, otherwise p is an outlier. The complete set of inliers is called the *consensus set*, and its size is referred to as the *score*. The whole process from picking a minimal set to computing the consensus set and its score, as shown in Algorithm 6, is repeated a fixed number of times, after which the shape instance with the highest score is outputted (Figure 12.11d).

Algorithm 6: The RANSAC algorithm

Input: An input point cloud P, the error threshold ϵ, the minimal number of points needed to uniquely construct the shape of interest n, and the number of iterations k

Output: the detected shape instance $\mathcal{I}_{best}$

```
1  s_best ← 0;
2  I_best ← nil;
3  for i ← 0...k do
4      M ← n randomly selected points from P;
5      I ← shape instance constructed from M;
6      C ← ∅ ;
7      for all p ∈ P \ M do
8          d ← distance(p, I);
9          if d < ϵ then
10             add p to C;
11     s ← score(C);
12     if s > s_best then
13         s_best ← s ;
14         I_best ← I;
```

The most touted benefit of RANSAC is its robustness, ie its performance in the presence of many outliers (up to 50%). Other algorithms to identify planes, eg fitting a plane with least-square adjustment, are usually more sensitive to the presence of noise and outliers (which are always present in real-world datasets). The probability that a shape instance is detected with RANSAC depends mainly on two criteria:

1. the number of inliers in P, and
2. the number of iterations k.

Naturally, it will be easier to detect a shape instance in a dataset with a relatively low number of outliers. And it is more likely that a shape instance is found if more minimal sets are evaluated. Picking a sufficiently high k is therefore important for the success of the algorithm (although a higher k also increases the computation time).

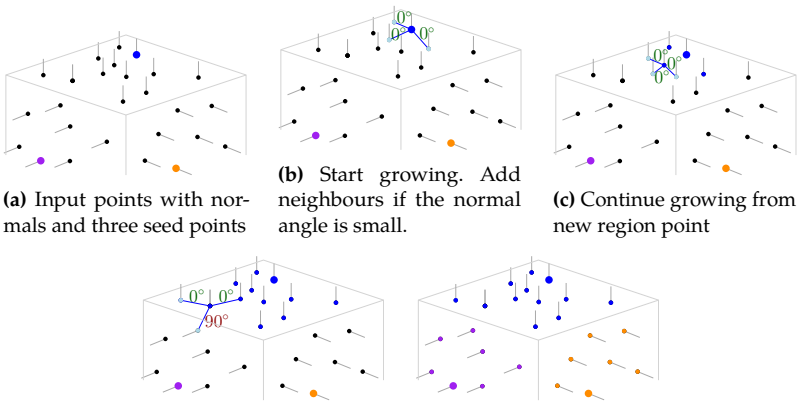
Because of the random nature of RANSAC, the minimal sets that it will

evaluate will be different every time you run the algorithm, even if the input data is the same. The detected shape instance can therefore also be different every time you run the algorithm; RANSAC is therefore said to be a *non-deterministic* algorithm. This could be a disadvantage.

Time complexity. The time complexity of RANSAC is $\mathcal{O}(kn)$, where n is the size of P.

12.5.2 Region growing

Region growing works by gradually growing sets of points called *regions* that fit a particular shape instance. A region R starts from a *seed point*, ie a point that is suspected to fit a shape instance. More points are added to R by inspecting *candidate points*, ie points in the neighbourhood of the members of R. To check if a candidate point c should be added to R, a test is performed. In the case of region growing for plane detection (see Figure 12.12) this test entails computing the angle between the normal vector of c and the normal vector[¶] of its neighbour in R. If this angle is small it is assumed that c lies in the plane instance that corresponds to R, and that it can therefore be added to R. Otherwise c is ignored (Figure 12.12d). This process of growing R continues until no more



(a) Input points with normals and three seed points

(b) Start growing. Add neighbours if the normal angle is small.

(c) Continue growing from new region point

(d) Stop growing where the normal angle is too great

(e) Final regions from all three seed points

Figure 12.12: Region growing for plane detection based on the angle between neighbouring point normals

candidates can be found that are compatible with R. When this happens, the algorithm proceeds to the next seed point to grow a new region.

Algorithm 7 gives the pseudo-code for the region growing algorithm. Notice that the set S is used to keep track of the points in the current region whose neighbours still need to be checked. Also notice that candidate points that are already assigned to a region are skipped.

The seed points can be generated by assessing the local neighbourhood of each input point. For example in case of plane detection one could fit a plane through each point neighbourhood and subsequently sort all points on the fitting error. Naturally, points with a low plane fitting error are probably part of a planar region so we can expect them to be good seeds.

[¶] The normal vector for a point $p \in P$ can be found by fitting a plane to the points in the local neighbourhood of p. The vector orthogonal to this plane is the normal vector.

Algorithm 7: The Region growing algorithm

Input: An input point cloud P, a list of seed points L_S, a function to find the neighbours of a point $neighbours()$
Output: A list with detected regions L_R

```
1  L_R ← [];
2  for each s in L_S do
3      S ← {s};
4      R ← ∅;
5      while S is not empty do
6          p ← pop(S);
7          for each candidate point c ∈ neighbours(p) do
8              if c was not previously assigned to any region then
9                  if c fits with R then
10                     add c to S;
11                     add c to R;
12     append R to L_R;
```

To compute the point neighbourhoods a k-nearest neighbour search or a fixed radius search can be used, which can both be implemented efficiently using a kd-tree (see Section 10.2). Notice that region growing is based on the idea that we can always find a path of neighbouring points between any pair of points within the same region. This does mean that two groups of points that fit the same shape instance but are not connected through point neighbourhoods will end up in different regions. Other shape detection methods described in this chapter do not need point neighbourhood information.

Time complexity. If we assume that

1. the number of seeds in L_S is linear with n, ie the size of P,
2. the size of S is at most n, and that
3. a $k$nn search takes $\mathcal{O}(k \log n)$, where k is the number of neighbours,

we come to a worst-case time complexity of $\mathcal{O}(n^2 k \log n)$. In practice it should be better since S is not likely to be n large, and it will get smaller the more regions have been found.

12.5.3 Hough transform

The Hough transform uses a voting mechanism to detect shapes. It lets every point $p \in P$ vote on each shape instance that could possible contain p. Possible shape instances thus accumulate votes from the input points. The detected shape instances are the ones that receive the highest number of votes. To find the possible shape instances for p, the algorithm simply checks all possible parameter combinations that give a shape instance that fits with p.

It is thus important to choose a good parametrisation of the shape that is to be detected. For instance when detecting lines one could use the slope-intercept form, ie $y = mx + b$. However, this particular parametrisation can not easily represent vertical lines, because m would need become infinite which is computationally difficult to manage. A

better line parametrisation is the Hesse normal form which is defined as

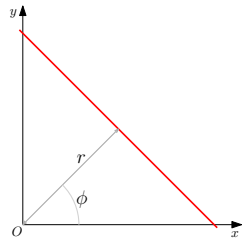
$$r = x \cos\phi + y \sin\phi.$$

As illustrated in Figure 12.13a (r, ϕ) are the polar coordinates of the point on the line that is closest to the origin, ie r is the distance from the origin to the closest point on the line, and $\phi \in [0°, 180°]$ is the angle between the positive x-axis and the line from the origin to that closest point on the line. This parametrisation has no problems with vertical lines (ie $\phi = 90°$). Similarly, for plane detection we can use the parametrisation

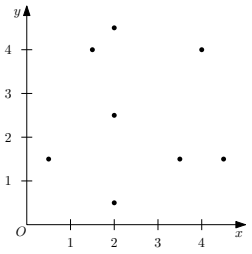
$$r = x \cos\theta \sin\phi + y \sin\phi \sin\theta + z \cos\phi.$$

Where (r, θ, ϕ) are the spherical coordinates of the point on the plane that is closest to the origin.

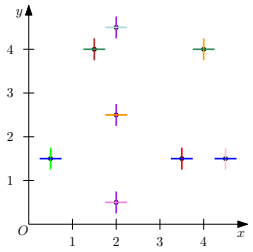
Figure 12.13 shows an example for line detection with the Hough transform and Algorithm 8 gives the full pseudo-code. The votes are saved



(a) Line parametrisation



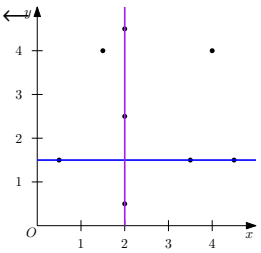
(b) Input points



(c) Line instances for each point

r \ ϕ	0°	90°
0.0	0	0
0.5	0	0
1.5	3 ←	1
2.0	0	3 ←
2.5	1	0
3.0	0	0
3.5	0	1
4.0	2	1
4.5	1	1

(d) Accumulator. Most occurring line instances are highlighted.



(e) Detected line instances

Figure 12.13: Hough transform for line detection with a 9×2 accumulator. The (ϕ, r) line parametrisation is chosen because this form can represent vertical lines (unlike the $y = mx + b$ form for example).

Algorithm 8: The Hough transform algorithm

Input: An input point cloud P, an accumulator matrix A, a detection threshold α

Output: A list with detected shape instances L_I

```
for each p in P do
    for each instance i from A that fits with p do
        increment A[i];
L_I ← all shape instances from A with a more than α votes;
```

in an *accumulator* which is essentially a matrix with an axis for each parameter of the shape, eg for detecting lines we would need two axes (See

Figure 12.13d). Notice that each element in the accumulator represents one possible shape instance. Because each axis only has a limited number of elements, each parameter is *quantised*. This means that each parameter is restricted in the possible values it can have. The chosen quantisation determines the sensitivity of the accumulator. The accumulator of Figure 12.13d for example, can only detect horizontal and vertical lines, because the ϕ parameter is quantised in only two possible values. Notice that the accumulator can be made more sensitive by choosing a finer quantisation, effectively increasing the size of the accumulator (although that will also make the algorithm run slower).

Time complexity. The time complexity of the Hough transform algorithm as discussed here is $\mathcal{O}(nm)$, where m is the number of elements in the accumulator.

12.6 Notes & comments

More information on the PLY format can be found online: `http://paulbourke.net/dataformats/ply/`. Notice that the PLY format can also be used to store a 3D mesh.

The full LAS specification, currently at version 1.4, is described in ASPRS (2013), and Isenburg (2013) describes the details of the compressed LAZ format.

Arge et al. (2010) introduced the outlier detection method for echo-sounding datasets by cutting long edges in a TIN.

Axelsson (2000) originally proposed the greedy TIN insertion algorithm for ground filtering. He also describes how to handle discontinuities in the terrain such as cliffs. It should be said that his paper is a bit scarce on details, and if you are interested in those you are better off reading some excerpts of the work of Lin and Zhang (2014).

A comparison with several other ground filtering methods can be found in the work of Meng et al. (2010).

Fischler and Bolles (1981) originally introduced the RANSAC algorithm and applied to cartography in that same paper. On wikipedia you can read how you can compute the required number of RANSAC iterations to achieve a certain probability of success given that you know how many outliers there are in your dataset[‖].

Limberger and Oliveira (2015) describes how to efficiently do plane detection in large point clouds using a variant of the Hough transform.

12.7 Exercises

1. The LAS standard gives a global point offset in the header. What is the benefit of using such a global offset?
2. What is the difference between thinning an point cloud prior to triangulation and TIN simplification?

[‖] `https://en.wikipedia.org/wiki/Random_sample_consensus#Parameters`

3. What is the probability that the line instance in Figure 12.11d is detected with $k = 2$ iterations?
4. In Chapter 2 it is described how point density can vary based on the acquisition conditions. How could a (strongly) varying point density affect the effectiveness of the region growing algorithm?
5. How many axes would an accumulator need for plane detection with the Hough transform?

Processing bathymetric data to produce hydrographic charts

13

An hydrographic chart is a map of the underwater world specifically intended for the safe navigation of ships, see below for an example. In its digital form, it is often called an electronic navigational chart, or ENC. The information appearing on an ENC are standardised, and there are open formats.

We focus in this chapter on one element of these charts: depth-contours. These are contour lines that, instead of elevation, show the depth with respect to a given level of water. The creation of these depth-contours from an input point cloud of depth measurements requires many tools that were introduced in this book; Delaunay triangulation and the Voronoi diagram (Chapter 3), interpolation (Chapter 4), and contour generation from a TIN (Chapter 8).

13.1 How are depth-contours produced in practice?

Traditionally, depth-contours were drawn by hand by skilled hydrographers. They used a sparse set of scattered surveyed depth measurements to deduct and depict the morphology of the seafloor with smooth-looking curves.

multibeam echosounder (MBES)

Nowadays, with technologies such as multibeam echosounders (MBES) offering an almost full coverage of the seafloor (see Section 2.2.3), one would expect the contouring process to be fully automatic. It is however in practice still a (semi-)manual process since the new technologies have ironically brought new problems: computers have problems processing the massive amount of data, especially in choosing which data is relevant and which is not.

The raw contours constructed directly from MBES datasets are often not satisfactory for navigational purposes since, as Figure 13.2a shows, they are zigzagging (the representation of the seafloor thus contains "waves", ie the slope changes abruptly) and they contain many "island" contours (seafloor has several local minima and maxima). These artefacts are the result of measurement noise that is present in MBES datasets, ie the variation in depth between two close samples can be larger than in reality, even after the dataset has been (statistically) cleaned. Figure 13.2b illustrates what is expected by hydrographers.

13.1.1 Generalisation is required to obtain good depth contours

Creating good depth-contours requires *generalisation*, ie the process of meaningfully reducing information.

Figure 13.1: An example of an ENC (electronic navigational chart) in the Netherlands. [photo of a paper map from the *Hydrografische Dienst*]

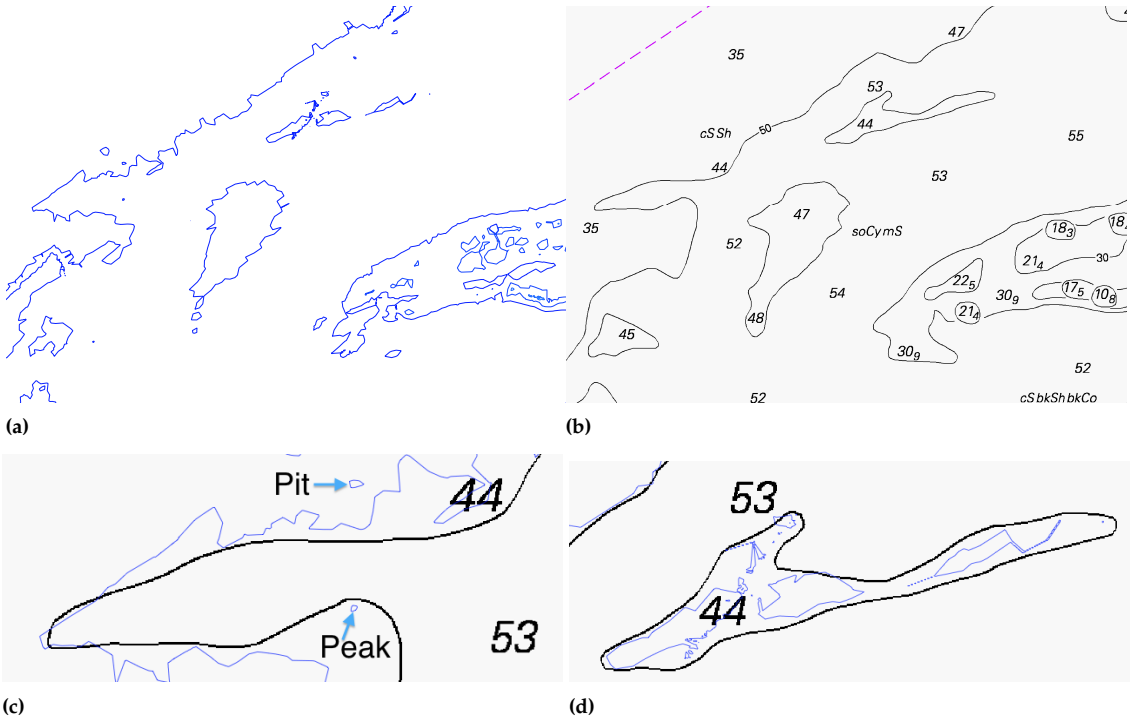


Figure 13.2: Comparison of **(a)** depth-contours obtained automatically from the raw MBES data and **(b)** the hydrographic chart from the Royal Australian Navy for the Torres Strait north of Australia. Raw depth contours are blue, generalized depth contours are black. **(c)** Pits are removed, while peaks are preserved or integrated with another contour. **(d)** Groups of nearby contour lines are aggregated

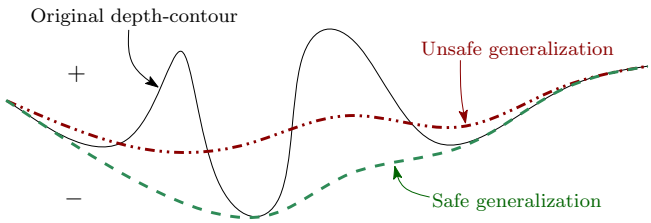


Figure 13.3: During generalisation, depth-contours can only be moved towards greater depth (indicated by a "–" in the figure).

The process of generalisation is guided by constraints that essentially define when a depth-contour is "good". A good depth contour satisfies all of the following four generalisation constraints.

1. The *safety constraint*. At every location, the indicated depth must not be deeper than the depth that was originally measured at that location; this is to guarantee that a ship never runs aground because of a faulty map. This constraint is a so-called hard constraint, ie it can never be broken.
2. The *legibility constraint*. An overdose of information slows down the map reading process for the mariner, thus only the essential information should be depicted on the map in a form that is clearly and efficiently apprehensible.
3. The *topology constraint*. The topology of the depicted map elements must be correct, ie isocontours may not touch or intersect (also a hard constraint).
4. The *morphology constraint*. The map should be as realistic and accurate as possible, ie the overall shape of the morphology of the underwater surface should be clearly perceivable and defined features should be preserved.

It should be noted that these four constraints are sometimes incompatible with each other. For instance, the morphology constraint tells us to stay close to the measured shape of the seafloor, while the legibility constraint forces us to deviate from that exact shape by disregarding details.

Also, because of the safety constraint, depth-contours can only be modified such that the safety is respected at all times: contours can only be pushed towards the deeper side during generalisation, as illustrated in Fig 13.3. It is therefore obvious that the end result must be a reasonable compromise between the four constraints, although the hard constraints must not be broken.

13.2 Common methods used in practice are not satisfactory

The generation of depth contours, and their generalisation, can be done by several methods. We present here the most frequently used methodologies to generate depth-contours from an MBES point cloud.

13.2.1 Displacement and generalisation of the lines

It is tempting to start with the raw contours lines and use a generalisation operator to simplify them, eg the Douglas-Peucker method. It should

however be noticed that this method does *not* guarantee that the safety constraint will be respected, that is the generalised line will be "pushed" both to the deeper and the shallower part (there is no control for this with the Douglas-Peucker method).

There exist a few algorithms to control the direction in which a line can be moved (these methods are outside the scope of this book), but these methods work only on lines individually and thus the resulting set of lines can contain intersecting lines. Furthermore, this solution does not solve the presence of many island contours, or can at best delete the small ones (and not aggregate them as in Figure 13.2d).

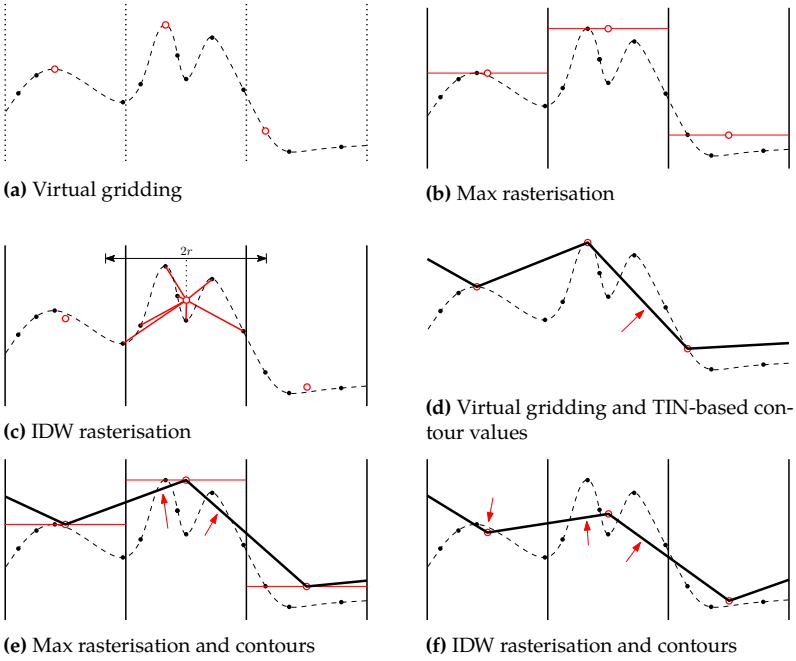
13.2.2 Creation of a simplified raster

Practitioners usually first interpolate the original MBES samples to create a (coarse) grid and then directly extract the contours from the grid. If the number of samples is too high to be processed by a computer, they often use a subset, which has the added benefits of creating smoother and simpler depth-contours.

The following are methods that use a raster data structure either to select a subset of the input samples or to construct a raster surface.

Selection with virtual gridding. This is a point filtering method that aims at reducing the volume of data, in order to create generalised contours and to speed up the computation time, or simply to make the computation possible, in the case the input dataset is several orders of magnitude bigger than the main memory of a computer. The idea is to overlay a virtual grid on the input points and to keep one point for every grid cell. The selected points can either be used to construct a raster or TIN surface, see below. While different functions can be used to select the point (eg deepest, shallowest, average, or median), because of the safety constraint the shallowest point is often chosen by practitioners, see Figure 13.4a for a one-dimensional equivalent. It should however be stressed that choosing the shallowest point does not guarantee safe contours. The problem is that contour extraction algorithms perform a linear interpolation on the raster cells. As can be observed from Figure 13.4d, this easily results in safety violations at 'secondary' local maxima in a grid cell. The number and severity of these violation is related to the cellsize of the virtual grid: a bigger cellsize will result in more and more severely violated points. Notice that it is not possible to reduce the cellsize such that the safety issue can be guaranteed.

Max rasterisation. As Figure 13.4b shows, it is similar to virtual gridding, the main difference is that a raster (a surface) is created where every cell in the virtual grid becomes a raster cell whose depth is the shallowest of all the samples. This disregards the exact location of the original sample points, and moves the shallowest point in the grid cell to the centre of the pixel. That means that the morphology constraint is not respected. Moreover, as Figure 13.4e shows, the safety constraint is not guaranteed, for the same reasons as with virtual gridding. Again, the severity of these problems depends on the chosen cellsize.



(a) Virtual gridding

(b) Max rasterisation

(c) IDW rasterisation

(d) Virtual gridding and TIN-based contour values

(e) Max rasterisation and contours

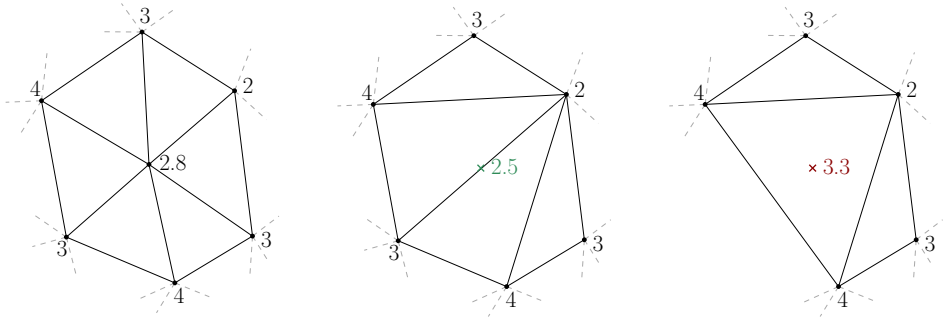
(f) IDW rasterisation and contours

Figure 13.4: On the top: profile views of different filtering and rasterisation methods. On the bottom: the corresponding contours. The arrows indicate where the safety constraint is violated with respect to the original points. Also note that in case a grid cell contains no data, no contours can be derived.

Interpolation to a raster. For hydrographic charts, the raster surface is often constructed with spatial interpolation, particularly with the method of inverse distance weighting (IDW). Figures 13.4c and 13.4f illustrate the process of IDW interpolation, notice that as a result of the averaging that takes place, extrema are disregarded and subsequently the safety constraint is also violated.

13.2.3 TIN simplification

One could use TIN simplification as explained in Chapter 8 to simplify the riverbed. This would also simplify the depth-contours that are generated from the TIN. However, as Figure 13.5 shows, the safety constraint is not guaranteed to be respected when vertices are removed from a TIN. This is due to the fact that the triangulation must be updated (with flips, see Chapter 3) and the shapes of the triangles are not controlled by the depth of the vertices, but rather by the Delaunay criterion.



(a) Initial configuration **(b)** 1st vertex removal **(c)** 2nd vertex removal

Figure 13.5: Due to the re-triangulation after a removal, violations of the safety constraint may occur after a series of points are removed. The first vertex is removed (locally the resulting surface will be shallower). However, the second removal changes the configuration of triangles and at that location the surface is now deeper. A lower number means a shallower point.

13.3 A Voronoi-based surface approach

Part of the problems with existing approaches to generate depth-contours is the fact that the different processes, such as spatial interpolation, generalisation and contouring, are treated as independent processes. These are in fact interrelated, and we introduce in the following a method where the different processes are integrated. This method uses several of the algorithms and data structures studied in this book, and with small extensions and modifications we can obtain depth-contours that are safe and legible.

The key idea behind the method, called the Voronoi-based surface approach, is to have one single consistent representation of the seafloor from which contours can be generated on-the-fly (potentially for different map scales, or with varying degrees of generalisation). Instead of performing generalisation by moving lines or using a subset of the original samples, we include all MBES points in a triangulation (the surface) and manipulate this triangulation directly with generalisation operators that fulfil the constraints listed in Section 13.1.1.

Figure 13.6 gives a schematic overview of the different components of our Voronoi-based surface concept.

Firstly, all the input points of a given area are used to construct a Delaunay TIN. Secondly, a number of generalisation operators are used that alter

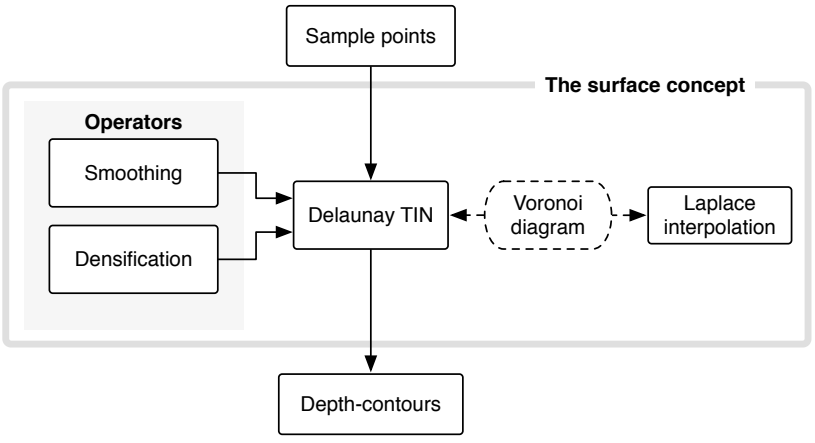


Figure 13.6: Overview of the Voronoi- and surface-based approach.

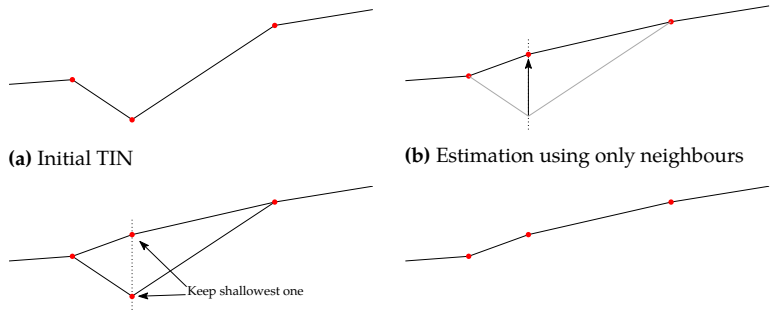
the TIN using Laplace interpolation, which is based on the Voronoi diagram. These operators aim at improving the slope of the surface, and permit us to generalise the surface. Finally, contour lines are derived from the altered TIN using linear interpolation.

Representing a field in a computer is problematic since computers are discrete machines. We therefore need to *discretise* the field, ie partition it into several pieces that cover the whole area (usually either grid cells or triangles). Contours in Figure 13.2a are not smooth basically because the seabed is represented simply with a TIN of the original samples, which is a C^0 interpolant. However, as we demonstrate below, we can obtain a smooth looking approximation of the field by densifying the TIN using the Laplace interpolant (see Section 4.3.5), which is C^1.

Two generalisation operators allow us to obtain a smoother surface from which depth-contours can be extracted: (1) smoothing; (2) densification.

13.3.0.1 The smoothing operator

The smoothing operator basically estimates, with the Laplace interpolant (see Section 4.3.5 on page 44), the depth of each vertex in a dataset by considering its natural neighbour (see Figure 13.7). If this depth is shallower, then the vertex is assigned this value; if it is deeper then nothing is done. Thus, the smoothing operator does not change the planimetric coordinates of vertices, but only lifts the vertices' depths.



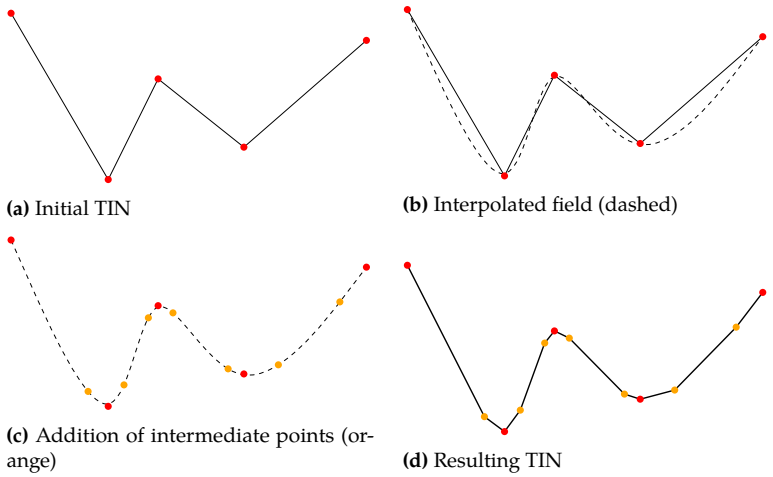
(a) Initial TIN **(b)** Estimation using only neighbours **(c)** Comparison of depths **(d)** Resulting TIN

Figure 13.7: Cross-section view of the smoothing of a single vertex in a TIN.

To perform the Laplace interpolation for each vertex v in the Voronoi diagram $\mathcal{D}$, it suffices to obtain the natural neighbours p_i of v, and for each calculate the lengths of the Delaunay and the dual Voronoi edge (as explained in Section 4.3.5). There is no need to insert/remove v in the dataset, since we are only interested in estimating its depth (without considering the depth it is already assigned).

The primary objective of smoothing is to generalise the surface by removing high frequency detail while preserving the overall feature shape. Applying it reduces the angle between adjacent triangles which gives the surface a smoother look.

It performs two linear loops over the n vertices of the dataset (the depths are only updated after all the depths have been estimated), and since the smoothing of one vertex is performed in expected constant time, the expected time complexity of the algorithm is $\mathcal{O}(n)$.



(a) Initial TIN

(b) Interpolated field (dashed)

(c) Addition of intermediate points (orange)

(d) Resulting TIN

Figure 13.8: Cross-section view of the densification operator in a TIN.

Observe that the operator can be performed either on a portion of a dataset, or on the whole dataset. Furthermore this operator can be applied any number of times, delivering more generalisation with each pass.

13.3.0.2 The densification operator

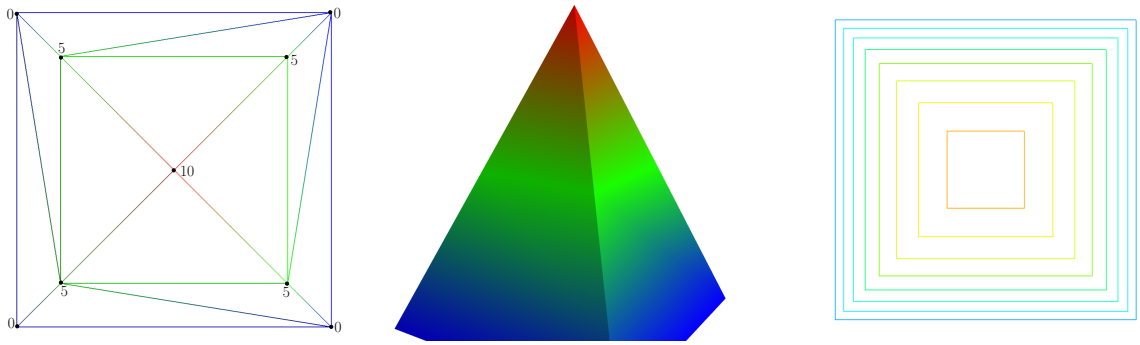
Its objective is primarily to minimise the discretisation error between the Laplace interpolated field and the contours that are extracted from the DT, this is illustrated in Figure 13.8.

By inserting extra vertices in large triangles (to break them into three triangles), the resolution of the DT is improved. As a result also the extracted contour lines have a smoother appearance because they now have shorter line-segments; see Section 8.2.4 for an explanation. We insert a new vertex at the centre of the circumscribed circle of any triangle that has an area greater than a preset threshold; its depth is assigned with the Laplace interpolant. The circumcentre is chosen here because that location is equidistant to its three closest points, and subsequently results in a very natural point distribution.

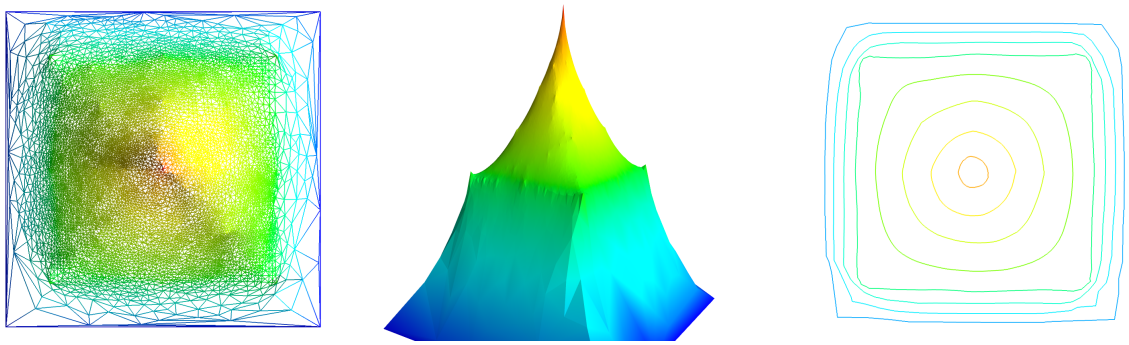
Figure 13.9 shows an example of these ideas. Figure 13.9a and Figure 13.9b show the original dataset, which is a very simple pyramid having its base at elevation 0, and its summit at 10. Figure 13.9d–f shows the results when a densification operator based on the Laplace interpolant is used. It should be noticed that the "top" of the pyramid was densified, and not so much the bottom, therefore the contour lines near the bottom should be ignored (the fact that they are close to the border of the dataset also creates artefacts).

Densification aims to reduce the difference between the linear TIN and the Laplace interpolated field of its vertices—effectively improving the resolution of the extracted contours. Therefore, densification is to be applied just before the extraction of the depth-contours. If applied *before* the smoothing operator, it would limit the effectiveness of that operator, since a denser triangulation smoothes more slowly.

The densification operator uses an area-threshold that determines which triangles should be densified. This way triangles that are already sufficiently small are not densified. It performs a single pass on the input



(a) Original data **(b)** Perspective view of original data **(c)** Contour lines from original data



(d) Densified with Laplace interpolant **(e)** Perspective view after densification **(f)** Contour lines from densified surface

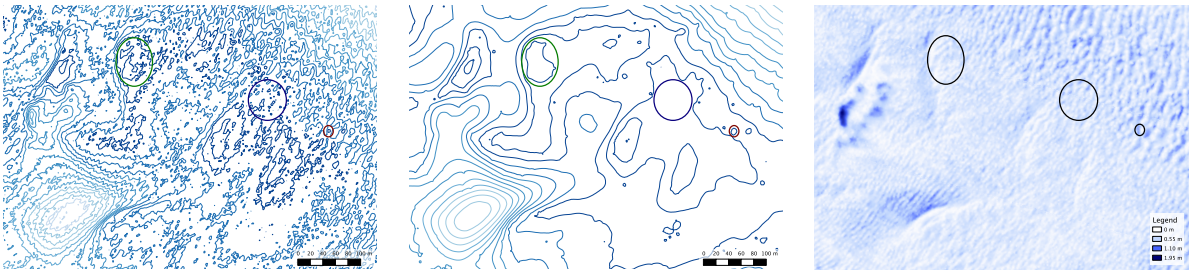
Figure 13.9: Original data are shown in **(a)** and **(b)**, and the resulting contour lines in **(c)**. The three figures below represent the same area densified with the Laplace interpolant.

triangles, thus with every call the resolution of the DT is increased, until all triangles have reached a certain area.

If the maximum area threshold is ignored, a single call costs $\mathcal{O}(n)$ time, as it only requires a single pass over the n triangles of the TIN. However, when a number of t densification passes is sequentially performed, it only scales to $\mathcal{O}(3^t\ n)$ time, since every point insertion creates two new triangles. However, because of the maximum area threshold, that worst case scenario will never be reached in practice with large t.

13.4 Some examples of results with real-world datasets

Figure 13.10 shows the results obtained with the implementation of the method described in this chapter. This was tested with an MBES dataset from Zeeland, in the Netherlands. As can be observed from Figure 13.10a, the raw and ungeneralised contours in the dataset have a very irregular and cluttered appearance. However, the smoothed contours (100 smoothing passes) from Figure 13.10b have a much cleaner and less cluttered appearance. Clearly, the number of contour lines has diminished. This is both because pits (local minima) have been lifted upwards by the smoothing operator, and nearby peaks (local maxima) have been aggregated (because the region in-between has been lifted upwards). Notice also that a third effect of the smoothing operator is the



(a) **(b)** **(c)**

Figure 13.10: The effect of the smoothing operator in the Zeeland dataset. **(a)** Raw contours extracted at a 50cm depth interval. **(b)** Smoothed contours (100 smoothing passes). The ellipses mark areas where aggregation (left), omission (middle) and enlargement (right) take place. **(c)** Difference map between the initial and 100X smoothed interpolated and rasterised fields (pixelsize 50 cm).

enlargement of certain features as a result of the uplifting of the points surrounding a local maximum.

The effects of the densification operator are also visible. The sharp edges of the undensified lines are caused by the large triangles in the initial TIN, however after densification these large triangles are subdivided into much smaller ones. The result is a much smoother contour line that still respects the sample points.

Naturally, the smoothing operator also smoothes and simplifies the resulting contour lines. This is demonstrated in Figure 13.11 illustrates the effect of the smoothing operator on a single contour over 30 smoothing passes. It is clear that the contour line moves towards the inner region, which is the deeper side of the contour, which is to be expected since the smoothing operator is safe per definition (and only lifts the surface upwards). What can also be seen is that the line is simplified (the details on the outer rim disappear, note however that the point count stays the same) and smoothed.

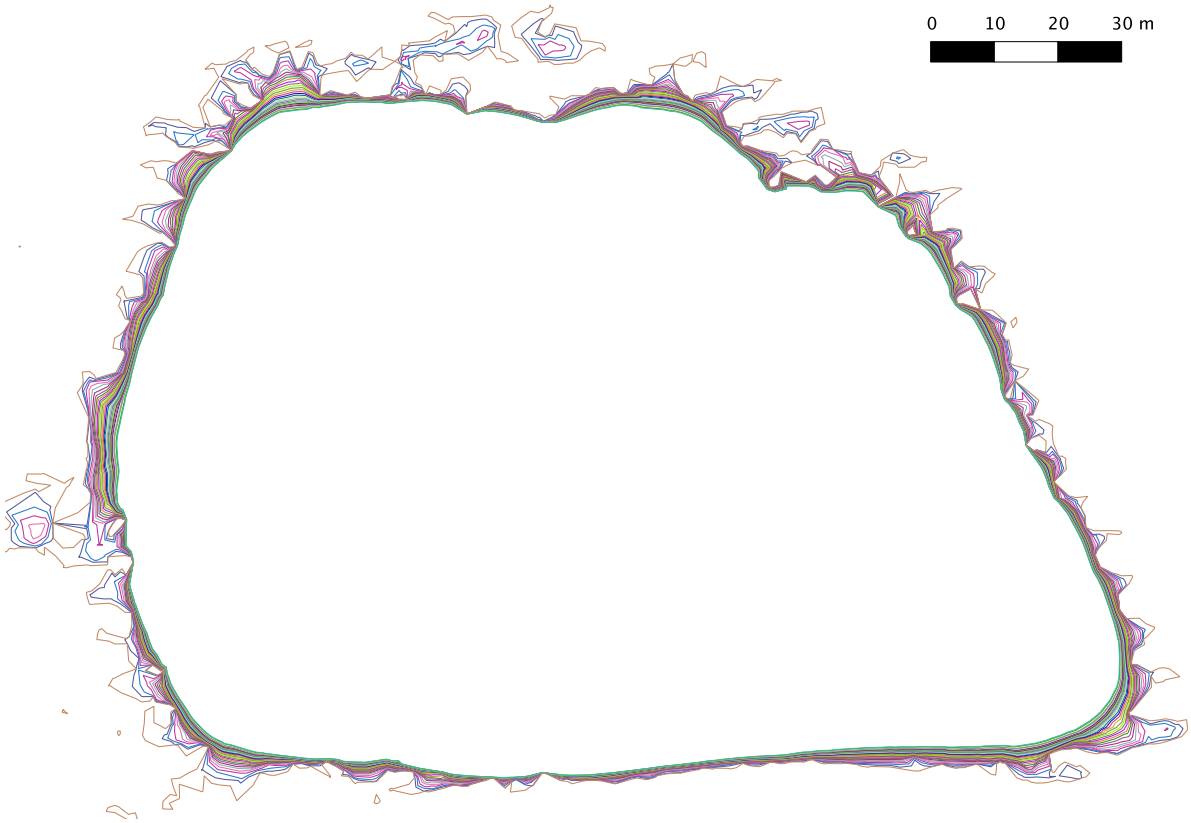


Figure 13.11: From 0X smoothing (outer) to 30X smoothing (inner) for a given dataset.

13.5 Notes & comments

The algorithm and the methodology of this chapter are (mostly) taken from Peters et al. (2014).

Zhang and Guilbert (2011) explains in detail how the generalisation of the content of a nautical chart is hindered by the four constraints.

13.6 Exercises

1. Explain why it is easier to respect the safety constraint using a TIN that has all the original MBES points as opposed to using a set of raw contours generated directly from that MBES dataset.
2. Explain why simplification with Douglas-Peucker is not applicable in a bathymetric context. And give a concrete example.
3. For a terrain "on the land", if Douglas-Peucker is used to simplify isocontours, what problems can be expected?
4. The Laplace interpolation is used in the methodology presented, but would the natural neighbour interpolation method also be suitable?

Bibliography

Akima, H. (1978). A method of bivariate interpolation and smooth surface fitting for irregularly distributed data points. *ACM Transactions on Mathematical Software* 4.2, pp. 148–159.

Arge, L., K. G. Larsen, T. Mølhave, and F. van Walderveen (2010). Cleaning massive sonar point clouds. *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*. GIS '10. San Jose, California: ACM, pp. 152–161.

ASPRS (2013). *LAS specification version 1.4 – R13*. The American Society for Photogrammetry & Remote Sensing.

Axelsson, P. (2000). DEM generation from laser scanner data using adaptive TIN models. *International Archives of Photogrammetry and Remote Sensing* 33.4, pp. 111–117.

Bailly, J.-S., P. Kinzel, T. Allouis, D. Feurer, and D. Le Coarer (2012). Airborne LiDAR Methods Applied to Riverine Environments. *Fluvial Remote Sensing for Science and Management*. GIScience and Remote Sensing. Wiley Blackwell, pp. 141–161.

Barber, C. B., D. P. Dobkin, and H. T. Huhdanpaa (1996). The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software* 22.4, pp. 469–483.

Barnes, R., C. Lehman, and D. Mulla (2014a). An efficient assignment of drainage direction over flat surfaces in raster digital elevation models. *Computers & Geosciences* 62, pp. 128–135.

Barnes, R., C. Lehman, and D. Mulla (2014b). Priority-flood: An optimal depression-filling and watershed-labeling algorithm for digital elevation models. *Computers & Geosciences* 62, pp. 117–127.

Berg, M. de (1997). Visualization of TINs. *Algorithmic Foundations of Geographic Information Systems*. Ed. by M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer. Vol. 1340. Lecture Notes in Computer Science. Berlin: Springer-Verlag, pp. 79–97.

Beven, K. (2012). *Rainfall-Runoff Modelling: The Primer*. Wiley-Blackwell.

Blelloch, G. E. (1990). *Vector models for data-parallel computing*. MIT Press.

Brown, K. Q. (1979). Voronoi diagrams from convex hulls. *Information Processing Letters* 9.5, pp. 223–228.

Burrough, P. A. and R. A. McDonnell (1998). *Principles of Geographical Information Systems*. Oxford University Press.

Chazette, P., J. Totems, L. Hespel, and J.-S. Bailly (2016). 5 - Principle and Physics of the LiDAR Measurement. *Optical Remote Sensing of Land Surface*. Ed. by N. Baghdadi and M. Zribi. Elsevier, pp. 201 –247.

Clough, R. and J. Tocher (1965). Finite element stiffness matrices for analysis of plates in bending. *Proceedings of Conference on Matrix Methods in Structural Analysis*.

Cressie, N. A. C. (1993). *Statistics for Spatial Data*. Wiley Series in Probability and Statistics. John Wiley & Sons.

Dakowicz, M. and C. M. Gold (2003). Extracting meaningful slopes from terrain contours. *International Journal of Computational Geometry and Applications* 13.4, pp. 339–357.

De Floriani, L. and P. Magillo (1999). intervisibility on terrains. *Geographical Information Systems*. Ed. by P. A. Longley, M. F. Goodchild, D. J. Maguire, and D. W. Rhind. Vol. 1. John Wiley & Sons. Chap. 38, pp. 543–556.

De Floriani, L. and P. Magillo (2002). Regular and irregular multi-resolution terrain models: A comparison. *Proceedings 10th ACM International Symposium on Advances in GIS*. McLean, Virginia, USA, pp. 143–148.

Delaunay, B. N. (1934). Sur la sphère vide. *Izvestia Akademia Nauk SSSR, Otdelenie Matematicheskii i Estestvennyka Nauk* 7, pp. 793–800.

Devillers, O. (2009). *Vertex removal in two dimensional Delaunay triangulation: Asymptotic complexity is pointless*. Tech. rep. RR-7104. INRIA.

Devillers, O., S. Pion, and M. Teillaud (2002). Walking in a triangulation. *International Journal of Foundations of Computer Science* 13.2, pp. 181–199.

Dirichlet, G. L. (1850). Über die reduktion der positiven quadratischen formen mit drei unbestimmten ganzen zahlen. *Journal für die Reine und Angewandte Mathematik* 40, pp. 209–227.

Dowman, I. (Jan. 2004). Integration of LiDAR and IFSAR for mapping. *International Archives of Photogrammetry and Remote Sensing* 35.

Duckham, M., L. Kulik, M. Worboys, and A. Galton (2008). Efficient generation of simple polygons for characterizing the shape of a set of points in the plane. *Pattern Recognition* 41.10, pp. 3224–3236.

Dyn, N., D. Levin, and S. Rippa (1990). Data dependent triangulations for piecewise linear interpolation. *IMA Journal of Numerical Analysis* 10.1, pp. 137–154.

Eberly, D. (2018). C^1 quadratic interpolation of meshes. https://www.geometrictools.com/Documentation/C1QuadraticInterpolation.pdf.

Edelsbrunner, H. and E. P. Mücke (1994). Three-dimensional alpha shapes. *ACM Transactions on Graphics* 13.1, pp. 43–72.

Edelsbrunner, H. and R. Seidel (1986). Voronoi diagrams and arrangements. *Discrete & Computational Geometry* 1, pp. 25–44.

Edelsbrunner, H., J. Harer, and A. Zomorodian (2001). Hierarchical morse complexes for piecewise linear 2-manifolds. *Proceedings 17th Annual Symposium on Computational Geometry*. Medford, USA: ACM Press, pp. 70–79.

Ester, M., H.-P. Kriegel, J. Sander, and X. Xu (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. *Proceedings 2nd International Conference on Knowledge Discovery and Data Mining (KDD-96)*. Ed. by E. Simoudis, J. Han, and U. M. Fayyad, pp. 226–231.

Evan, I. S. (1980). An integrated system of terrain analysis and slope mapping. *Zeitschrift für Geomorphologie, Supplementband* 36, pp. 274–295.

Fairfield, J. and P. Leymarie (1991). Drainage networks from grid elevation models. *Water Resources Research* 27.5, pp. 709–717.

Farin, G. (1985). A modified Clough-Tocher interpolant. *Computer Aided Geometric Design* 2.1–3, pp. 19–27.

Ferretti, A., A. Monti-Guarnieri, C. Prati, F. Rocca, and D. Massonnet (2007). *InSAR Principles: Guidelines for SAR Interferometry Processing and Interpretation*. ESA Publications.

Fewster, R. (2014). Lecture notes for Stats 325. Available at https://www.stat.auckland.ac.nz/~fewster/325/notes.php.

Fischler, M. A. and R. C. Bolles (June 1981). Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography. *Commun. ACM* 24.6, pp. 381–395.

Fisher, P. F. (1997). The pixel: A snare and a delusion. *International Journal of Remote Sensing* 18.3, pp. 679–685.

Flötotto, J. (2003). A coordinate system associated to a point cloud issued from a manifold: definition, properties and applications. PhD thesis. Université de Nice-Sophia Antipolis.

Fortune, S. (1987). A sweepline algorithm for Voronoi diagrams. *Algorithmica* 2, pp. 153–174.

Frank, A. U. (1992). Spatial concepts, geometric data models, and geometric data structures. *Computers & Geosciences* 18.4, pp. 409–417.

Galton, A. and M. Duckham (2006). What Is the Region Occupied by a Set of Points? *Geographic Information Science*. Springer, pp. 81–98.

Garland, M. and P. S. Heckbert (1995). *Fast polygonal approximation of terrain and height fields*. Tech. rep. CMU-CS-95-181. Pittsburgh, PA, USA: School of Computer Science, Carnegie Mellon University.

Goodchild, M. F. (1992). Geographical data modeling. *Computers & Geosciences* 18.4, pp. 401–408.

Gröger, G. and L. Plümer (2005). How to Get 3-D for the Price of 2-D—Topology and Consistency of 3-D Urban GIS. *GeoInformatica* 9.2, pp. 139–158.

Gudmundsson, J., M. Hammar, and M. van Kreveld (2002). Higher order Delaunay triangulations. *Computational Geometry—Theory and Applications* 23, pp. 85–98.

Guibas, L. J. and J. Stolfi (1985). Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics* 4.2, pp. 74–123.

Horn, B. K. P. (1981). Hill shading and the reflectance map. *Proceedings IEEE*. Vol. 69. 1, pp. 14–47.

Isenburg, M. (2013). LASzip: lossless compression of LiDAR data. *Photogrammetric Engineering and Remote Sensing* 79.2, pp. 209–217.

Isenburg, M., Y. Liu, J. R. Shewchuk, J. Snoeyink, and T. Thirion (2006a). Generating Raster DEM from Mass Points Via TIN Streaming. *Geographic Information Science—GIScience 2006*. Vol. 4197. Lecture Notes in Computer Science. Münster, Germany, pp. 186–198.

Isenburg, M., Y. Liu, J. R. Shewchuk, and J. Snoeyink (2006b). Streaming computation of Delaunay triangulations. *ACM Transactions on Graphics* 25.3, pp. 1049–1056.

Jarvis, R. (1973). On the identification of the convex hull of a finite set of points in the plane. *Information Processing Letters* 2.1, pp. 18–21.

Kjellstrom, B. and C. Kjellstrom Elgin (2009). *Be expert with map and compass*. 3rd. Wiley.

Kornus, W and A Ruiz (2003). Strip adjustment of LIDAR data. *V Semana Geomática de Barcelona* 11.03.

Kreveld, M. van (1996). Efficient methods for isoline extraction from a TIN. *International Journal of Geographical Information Science* 10.5, pp. 523–540.

Kreveld, M. van (1997). Digital elevation models and TIN algorithms. *Algorithmic Foundations of Geographic Information Systems*. Ed. by M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer. Vol. 1340. Lecture Notes in Computer Science. Berlin: Springer-Verlag, pp. 37–78.

Kreveld, M. van, R. van Oostrum, C. Bajaj, V. Pascucci, and D. Schikore (1997). Contour trees and small seed sets for isosurface traversal. *Proceedings 13th Annual Symposium on Computational Geometry*. Nice, France: ACM Press, pp. 212–219.

Krige, D. G. (1951). A statistical approach to some basic mine problems on the Witwatersrand. *Journal of the Chemical and Metallurgical Society of South Africa* 52, pp. 119–139.

Kumler, M. P. (1994). An intensive comparison of triangulated irregular networks (TINs) and digital elevation models (DEMs). *Cartographica* 31.2.

Kweon, I. S. and T. Kanade (1994). Extracting Topographic Terrain Features from Elevation Maps. *Journal of Computer Vision, Graphics and Image Processing: Image Understanding* 59.2, pp. 171–182.

Lawson, C. L. (1972). Transforming triangulations. *Discrete Applied Mathematics* 3, pp. 365–372.

Lichtenstern, A. (2013). Kriging methods in spatial statistics. MA thesis. Technische Universität München.

Limberger, F. A. and M. M. Oliveira (2015). Real-Time Detection of Planar Regions in Unorganized Point Clouds. *Pattern Recognition* 48.6, pp. 2043–2053.

Lin, X. and J. Zhang (2014). Segmentation-based filtering of airborne LiDAR point clouds by progressive densification of terrain segments. *Remote Sensing* 6.2, pp. 1294–1326.

Liu, Y. and J. Snoeyink (2005). The "far away point" for Delaunay diagram computation in $\mathbb{E}^d$. *Proceedings 2nd International Symposium on Voronoi Diagrams in Science and Engineering*. Seoul, Korea, pp. 236–243.

Magillo, P., E. Danovaro, L. De Floriani, L. Papaleo, and M Vitali (2009). A discrete approach to compute terrain morphology. *Computer Vision and Computer Graphics. Theory and Applications*. Vol. 21. Springer Berlin Heidelberg, pp. 13–26.

Matheron, G. (1962). *Traité de géostatistique appliquée*. Editions Technip.

Matheron, G. (1965). *Les variables régionalisées et leur estimation: une application de la théorie des fonctions aléatoires aux sciences de la nature*. Masson.

Meng, X., N. Currit, and K. Zhao (2010). Ground Filtering Algorithms for Airborne LiDAR Data: A Review of Critical Issues. *Remote Sensing* 2.3, pp. 833–860.

Metz, M., H. Mitasova, and R. Harmon (2011). Efficient extraction of drainage networks from massive, radar-based elevation models with least cost path search. *Hydrology and Earth System Sciences* 15, pp. 667–678.

Mitasova, H. and L. Mitas (1993). Interpolation by regularized spline with tension: I. Theory and implementation. *Mathematical Geology* 25, pp. 641–655.

Moreira, A. and M. Y. Santos (2007). Concave hull: a k-nearest neighbours approach for the computation of the region occupied by a set of points. *Proceedings GRAPP 2007, 2nd International Conference on Computer Graphics Theory and Applications*. Barcelona, Spain, pp. 61–68.

Mostafavi, M. A., C. M. Gold, and M. Dakowicz (2003). Delete and insert operations in Voronoi/Delaunay methods and applications. *Computers & Geosciences* 29.4, pp. 523–530.

Mücke, E. P., I. Saias, and B. Zhu (1999). Fast randomized point location without preprocessing in two- and three-dimensional Delaunay triangulations. *Computational Geometry—Theory and Applications* 12, pp. 63–83.

O'Callaghan, J. F. and D. M. Mark (1984). The extraction of drainage networks from digital elevation data. *Computer Vision, Graphics, and Image Processing* 28, pp. 323–344.

Okabe, A., B. Boots, K. Sugihara, and S. N. Chiu (2000). *Spatial tessellations: Concepts and applications of Voronoi diagrams*. Second. John Wiley and Sons.

Oliver, M. A. and R. Webster (2015). *Basic Steps in Geostatistics: The Variogram and Kriging*. Springer Briefs in Agriculture. Springer.

Peters, R., H. Ledoux, and M. Meijers (2014). A Voronoi-based approach to generating depth-contours for hydrographic charts. *Marine Geodesy* 37.2, pp. 145–166.

Quinn, P., K. Beven, P. Chevallier, and O. Planchon (1991). The prediction of hillslope flow paths for distributed hydrological modelling using digital terrain models. *Hydrological Processes* 5, pp. 59–79.

Ressl, C., H. Brockmann, G. Mandlburger, and N. Pfeifer (2016). Dense Image Matching vs. Airborne Laser Scanning – Comparison of two methods for deriving terrain models. *Photogrammetrie - Fernerkundung - Geoinformation* 2016.2, pp. 57–73.

Reuter, H., T. Hengl, P. Gessler, and P. Soille (2009). Chapter 4 Preparation of DEMs for Geomorphometric Analysis. *Geomorphometry*. Ed. by T. Hengl and H. I. Reuter. Vol. 33. Developments in Soil Science. Elsevier, pp. 87 –120.

Rippa, S. (1990). Minimal roughness property of the Delaunay triangulation. *Computer Aided Geometric Design* 7, pp. 489–497.

Salomon, D. (2011). Visible Surface Determination. *Texts in Computer Science*. Springer London, pp. 891–910.

Schneider, B. (2005). Extraction of hierarchical surface networks from bilinear surface patches. *Geographical Analysis* 37, pp. 244–263.

Seidel, R. (1982). Voronoi diagrams in higher dimensions. PhD thesis. Austria: Diplomarbeit, Institut für Informationsverarbeitung, Technische Universität Graz.

Shewchuk, J. R. (1997). Delaunay Refinement Mesh Generation. PhD thesis. Pittsburg, USA: School of Computer Science, Carnegie Mellon University.

Sibson, R. (1978). Locally equiangular triangulations. *The Computer Journal* 21, pp. 243–245.

Sibson, R. (1981). A brief description of natural neighbour interpolation. *Interpreting Multivariate Data*. Ed. by V. Barnett. New York, USA: Wiley, pp. 21–36.

Sibson, R. (1997). Contour Mapping: Principles and Practice. Available at `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.51.63`.

Skidmore, A. K. (1989). A comparison of techniques for calculating gradient and aspect from a gridded digital elevation model. *International journal of geographical information systems* 3.4, pp. 323–334.

Tarborton, D. G. (1997). A new method for the determination of flow directions and upslope areas in grid digital elevation models. *Water Resources Research* 33.2, pp. 309–319.

Thiessen, A. H. (1911). Precipitation average for large areas. *Monthly Weather Review* 39, pp. 1082–1084.

Tse, R. O. C. and C. M. Gold (2004). TIN meets CAD—extending the TIN concept in GIS. *Future Generation Computer Systems* 20.7, pp. 1171–1184.

Vitter, J. S. (2001). External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys* 33.2, pp. 209–271.

Voronoi, G. M. (1908). Nouvelles applications des paramètres continus à la théorie des formes quadratiques. *Journal für die Reine und Angewandte Mathematik* 134, pp. 198–287.

Vosselman, G. (2002). Strip offset estimation using linear features. *3rd International Workshop on Mapping Geo-Surfical Processes using Laser Altimetry, Columbus, Ohio, USA*.

Wackernagel, H. (2003). *Multivariate Geostatistics: An Introduction with Applications*. Third. Springer.

Wang, K., C.-p. Lo, G. A. Brook, and H. R. Arabnia (2001). Comparison of existing triangulation methods for regularly and irregularly spaced height fields. *International Journal of Geographical Information Science* 15.8, pp. 743–762.

Watson, D. F. (1992). *Contouring: A guide to the analysis and display of spatial data*. Oxford, UK: Pergamon Press.

Wood, J. D. (1996). The Geomorphological Characterisation of Digital Elevation Models. PhD thesis. UK: University of Leicester.

Zevenbergen, L. W. and C. R. Thorne (1987). Quantitative analysis of land surface topography. *Earth Surface Processes and Landforms* 12.1, pp. 47–56.

Zhang, X. and E. Guilbert (2011). A multi-agent System Approach for Feature-driven Generalization of isobathymetric Line. *Advances in Cartography and GIScience. Volume 1*. Ed. by A. Ruas. Springer, pp. 477–495.

Alphabetical Index