Computers & Geosciences 36 (2010) 1123–1134

Contents lists available at ScienceDirect

Computers & Geosciences

journal homepage: www.elsevier.com/locate/cageo

A simplex-based approach to implement dimension independent spatial analyses

Farid Karimipour [a,b,*], Mahmoud R. Delavar [a], Andrew U. Frank [b,1]

[a] Department of Surveying and Geomatics Engineering, College of Engineering, University of Tehran, Tehran, Iran
[b] Institute for Geoinformation and Cartography, Vienna University of Technology, Gusshausstr. 27–29, A-1040 Vienna, Austria

ARTICLE INFO

Article history:
Received 10 June 2009
Received in revised form
17 January 2010
Accepted 4 March 2010

Keywords:
n-Dimensional spatial analyses
3D GIS
Simplexes
List data structure
Delaunay triangulation

ABSTRACT

Many applications in geosciences need to deal with 3D objects. For this, among other requirements, 2D spatial analyses must be extended to support 3D objects. This extension is an important research topic in GIS and computational geometry. Approaches that extend an existing algorithm for a 2D spatial analysis to work for 3D or higher dimensions lead to different algorithms and implementations for different dimensions. Following such approaches the code for a package that supports spatial analyses for both 2D and 3D cases is nearly two times the code size for 2D. While dimension independent algorithms are an alternative toward generalization, they are still implemented separately for each dimension. The main reason is that each dimension is modeled using a different data structure that requires its own implementation details. In this article we use the *list* data structure to implement *n-simplexes*—as a data type that supports spatial objects of any dimension. Primitive operations on *n*-simplexes become manipulating functions over lists, which are independent of the number and type of the elements. We define spatial analyses as combinations of primitive operations on *n*-simplexes. Since the primitive operations on *n*-simplexes have been implemented independently of dimension, the spatial analyses are dimension independent, too. Construction of Delaunay triangulation of *n*D points, as the basic data structure for many geoscientific researches, is used here as the running example. The implementation results for Delaunay triangulation of some 2D and 3D points are presented and discussed. As a case study the implementations are used to calculate the area and volume of the reservoir of a dam at different water levels, which leads to a level–surface–volume diagram.
© 2010 Elsevier Ltd. All rights reserved.

1. Introduction

Many applications in geosciences (e.g., geological modeling, oceanography, pore-space modeling and landslide deformation) need to deal with 3D objects (Apel, 2004; Houlding, 1994; Monga et al., 2007; Pouliot et al., 2008; Turner, 1992; Tzenkov and Gospodinov, 2003; Yanbing et al., 2007). There are also applications where "the objects of interest are spatial distribution of 3D continuous geographical phenomena such as the salinity of a body of water, the humidity of the air or the percentage of gold in the rock" (Ledoux and Gold, 2007). There is much effort on the extension of geospatial information systems (GIS) – as the tool to manage spatial information in a wide range of issues from data collection and storage to manipulation and visualization - to support 3D objects (Abdul-Rahman and Pilouk, 2008; Raper, 2000). This extension, among other requirements, needs support for 3D spatial analyses.

Extension of 2D spatial analyses to 3D has been the subject of many studies (Ledoux, 2007; Nonato et al., 2001; Preparata and Hong, 1997). Some research has modified an existing algorithm for a 2D spatial analysis to work for 3D or higher dimensions. Such approaches result in different algorithms, and consequently different implementations, for different dimensions. Following such approaches the code for a package that supports spatial analyses for both 2D and 3D cases is nearly two times the code size for 2D. An alternative is a dimension independent approach. Its advantage is that the same algorithm works for any dimension. However, because of lack of efficient geometric data structures, they are still implemented separately for each dimension. For example, the *Bowyer–Watson* algorithm to construct Delaunay triangulation is an *n*-dimensional approach, but it is implemented differently in 2D (Bowyer, 1981) and 3D (Field, 1986). Therefore, from an implementation point of view, there is no advantage in using dimension independent approaches.

This article pushes more toward dimension independency in spatial analyses (Frank, 1999; Frank and Kuhn, 1986; Karimipour et al., 2008a, 2008b). We use the *list* data structure and its manipulating functions to implement dimension independent

* Corresponding author at: Department of Surveying and Geomatics Engineering, College of Engineering, University of Tehran, Tehran, Iran. Tel.: +98 21 88008839; fax: +98 21 88008841.
E-mail addresses: fkarimipr@ut.ac.ir (F. Karimipour), mdelavar@ut.ac.ir (M.R. Delavar), frank@geoinfo.tuwien.ac.at (A.U. Frank).
[1] Tel.: +43 1 58801-12711; fax: +43 1 58801-12799.

0098-3004/$ - see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cageo.2010.03.002

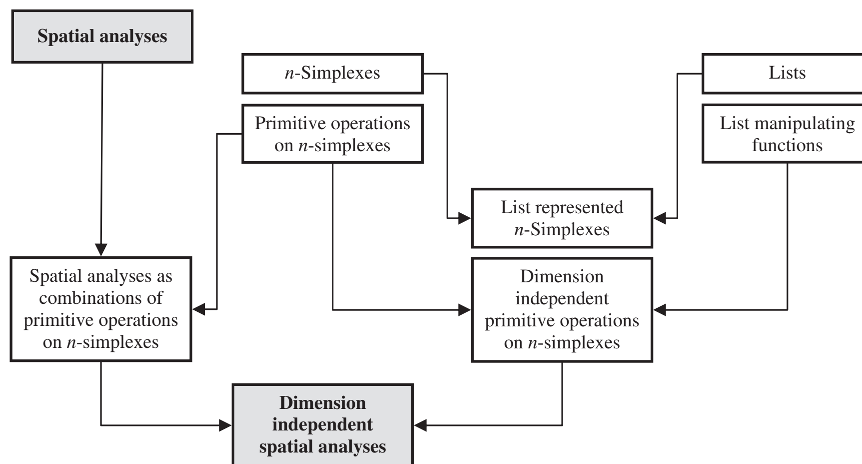


Fig. 1. Proposed approach of research.

spatial analyses. A *list* is a collection of any number of elements of the same type. A set of manipulating functions over lists are implemented, which are independent of the number and type of the elements of the list. As Fig. 1 shows, we represent an *n-simplex* – a data type that supports spatial objects of any dimension – as a list of *n*D points. It enables us to describe primitive operations on *n*-simplexes as manipulating functions over lists, which will be dimension independent. Finally, we define spatial analyses as combinations of primitive operations on *n*-simplexes. Since the primitive operations on *n*-simplexes have been implemented independently of dimension, the spatial analyses are also dimension independent.

The rest of the article explains the above diagram and shows how to use this for an example spatial operation. Section 2 introduces Delaunay triangulation as the running example. This is a frequently used spatial analysis in geoscientific applications. In this section, we explain a dimension independent algorithm for this spatial analysis, which will be implemented in the rest of the article. Section 3 reviews the concept of *n*-simplexes and their primitive operations. In Section 4, we describe the list data structure and introduce a set of manipulating functions over lists. Section 5 develops the *n*-simplex type and its primitive operations using the list data structure and its manipulating functions. Section 6 contains a dimension independent implementation of Delaunay triangulation using the algorithm described in Section 2. It starts with implementing a set of spatial operations that are required for constructing Delaunay triangulation. Their combination ends up in a dimension independent implementation of Delaunay triangulation. The implementation results for some 2D and 3D points are shown and discussed in Section 7. In Section 8, a case study from hydrographic data of the reservoir of a dam is presented. In this example, the *n*-dimensional Delaunay triangulations are used to calculate the area and volume of the reservoir at different water levels, which leads to a level–surface–volume diagram. Finally, Section 9 contains some conclusions and ideas for future works.

The algorithms and implementations presented here are *n*-dimensional. However, we limit the research to 2D and 3D, which are of interest in geosciences and their geometrical illustrations are possible.

2. Running example: Delaunay triangulation

There are two modeling approaches to treat the real world: *object-based* modeling where the real world is represented as

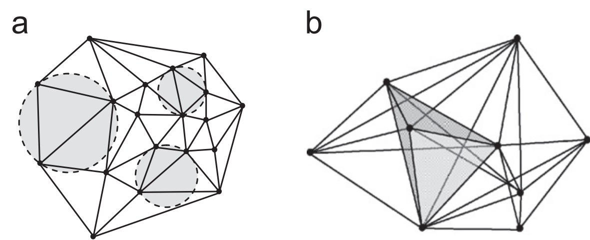


Fig. 2. Delaunay triangulation of some (a) 2D points and (b) 3D points. In 2D case, some of circumcircles are drawn. In 3D case, one of tetrahedra is highlighted.

discrete and identifiable entities and *field-based* modeling that represents the real world as a continuously changing field (Ledoux, 2007). Although the geoscientific phenomena are continuous, their representation through the field-based approach is problematic, because it is not possible to measure continuous phenomena everywhere. Therefore a sampling process at some finite locations is performed and the field is reconstructed from these samples. Delaunay triangulation is the widely used approach for this reconstruction (Aurenhammer, 1991; Delaunay, 1934; Ledoux and Gold, 2007).

Delaunay triangulation is well known in geosciences for many years (Mostafavi et al., 2003). It is the basic data structure for many geoscientific applications such as terrain modeling, spatial interpolation and geological mapping problem. It is also widely used in 3D geoscientific modeling. "3D Delaunay triangulation is used in many geoscientific applications that collect data about spatial objects and domains such as features of the solid earth (aquifers), oceans (currents) or atmosphere (weather fronts), which fill 3D space" (Lattuada and Raper, 1995). Furthermore, there are several applications in geosciences for which constructing the 3D Delaunay triangulation is the basis, e.g., surface modeling, isosurface extraction (Ledoux and Gold, 2007) and reconstruction of 3D complex geological objects (Yong et al., 2004).

Delaunay triangulation for a set of 2D points is the partitioning of the space into triangles that satisfies the empty circumcircle test: the circumcircle of each triangle does not contain any other point of the data set. Extension of this spatial analysis to *n*D points constructs a partitioning of the space to *n*D pyramids (with triangular bases) in which the *n*D circumsphere of each pyramid does not contain any other point of the data set. Fig. 2 shows Delaunay triangulation of some 2D and 3D points.

There are several algorithms to construct Delaunay triangulation (Guibas and Stolfi, 1985; Ledoux, 2006; Okabe et al., 2000;

O'Rourke, 1998). In this article we use an incremental algorithm called *Bowyer–Watson* algorithm, which is dimension independent (Bowyer, 1981; Watson, 1981). This algorithm is selected because it is among the simplest algorithms for constructing Delaunay triangulation and enables us to concentrate on the main goal of the research – which is *n*D implementation of a spatial analysis using lists – than describing the details and subtleties of the algorithms.

The *Bowyer–Watson* algorithm for a set of 2D points starts with constructing a big *triangle* that contains all of the points. Other points are inserted one by one into the construction and after each insertion the DT is modified: all the *triangles* that violate the *circumcircle* test, i.e., whose circumcircle contains the new point (Fig. 3a), are deleted from the construction (Fig. 3b). This creates a hole, which is filled by new *triangles* that are created by joining the new point to each *edge* of the boundary of the hole (Fig. 3c).

For 3D points, the overall procedure is similar, but with some changes in terminology:

- The algorithm starts with a big *tetrahedron* that contains all of the points.
- After each insertion, all the *tetrahedra* whose *circumsphere* contains the new point are deleted, and the hole is filled by new *tetrahedra* that are created by joining the new point to each *triangle* of the boundary of the hole.

The differences in terminology remain in the customary implementations, which lead to different implementations for 2D and 3D. The next section introduces the concept of *n*-simplexes that enables us to implement the *Bowyer–Watson* algorithm, absolutely independent of dimension.

3. *n*-Simplexes bring different dimensions together

The first step toward generalizing the *Bowyer–Watson* algorithm is to remove the expressions that depend on the dimension:

- Create an initial *nD pyramid* that contains all of the points
- Insert the other points to the construction and update the DT after each insertion as follows:
  - Delete all the *nD pyramids* that violate the *nD circumsphere* test, which results in creating a connected hole.
  - Fill the hole by new *nD pyramids*, which are created by joining the new point to each *element* of the boundary of the hole.

This definition explains the functionality of the *Bowyer–Watson* algorithm for points of any dimension. For its implementation, however, we need a data type that supports the general terminology used in this definition. This section introduces the concept of *n*-simplexes, as an *n*-dimensional data type for spatial objects, which has this capability. For this, the *n*-simplexes are introduced and some of their properties and operations are presented. The last subsection gives an implementable version for the above general definition, based on the concept of *n*-simplexes and their operations.

3.1. *What is an n-simplex?*

An *n*-simplex S_n is formally defined as "the smallest convex set in a Euclidian space (denoted as R^m, with $n \leq m$), containing $n+1$ points $v_0, \ldots, v_n$ that do not lie in a hyperplane of dimension less than n" (Hatcher, 2002). A simpler definition describes an *n*-simplex S_n as the simplest spanning geometric figure in the *n*-dimensional Euclidean space that contains $n+1$ points $v_0, \ldots, v_n$ of dimension n, providing that the vectors $v_1-v_0, \ldots, v_n-v_0$ are linearly independent. An *n*-simplex S_n is represented by the list of its vertexes as

$S_n = < v_0,\ldots,v_n >$

Note that each vertex itself is an *n*D point, so a detailed representation of an *n*-simplex is

$S_n = < (e_{01},\ldots,e_{0n}),\ldots,(e_{n1},\ldots,e_{nn}) >$

in which e_{ij} is the *j*th defining coordinate of the *i*th vertex. *n*-Simplexes are defined for any dimension. Table 1 shows 0- to 3-simplexes and their common names, representations and geometric configurations.

The concept of *n*-simplexes is extensively studied in the late 19th century by Henri Poincaré. It is the basis of the simplicial homology field, which is today considered as a part of algebraic or combinatorial topology (Hatcher, 2002).

For a given dimension *n*, an *n*-simplex is the elementary spatial objects from which other complex objects of that dimension are constructed. Any subset of the vertexes of S_n represents a face of S_n. A simplicial complex *C* is a finite set of simplexes that satisfies the following conditions (Fig. 4):

- Any face of a simplex from *C* is also in *C*.
- The intersection of any two simplexes $s_1, s_2 \in C$ is either empty or a face of both s_1 and s_2.

Simplicial complexes have interesting properties (see Alexandroff, 1961; Hatcher, 2002). They have been considered as a basic data type in developing spatial database systems (Penninga and van Oosterom, 2008; Schneider, 1997).

Simplicial complexes may consist of simplexes of different dimensions (Fig. 4a). A *homogeneous* simplicial *k*-complex is a simplicial complex where every simplex of dimension less than *k* is the face of some simplex of dimension exactly *k* (Alexandroff, 1961). For example, a triangulation of a set of 2D points is a homogeneous simplicial 2-complex. In this article, only sets of *n*-simplexes are used, so they are homogenous *n*-complexes and have their properties.

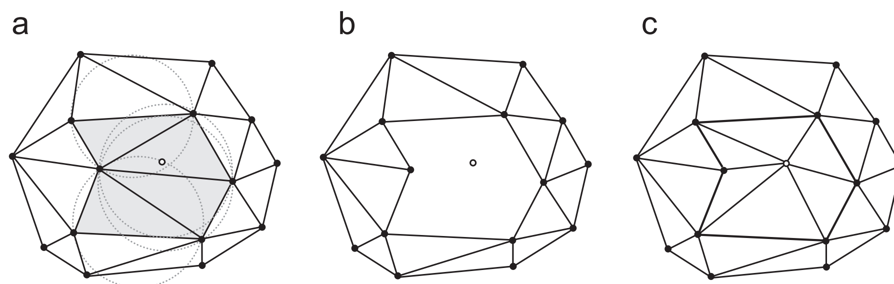
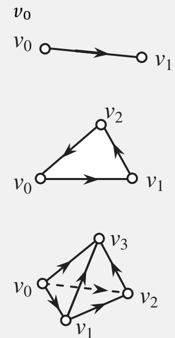


Fig. 3. New point indicated as white is added to DT: (a) and (b) all triangles whose circumcircle contains new point are detected and deleted. (c) Hole is filled by new triangles, which are created by joining new point to each edge of boundary of hole (Ledoux, 2006).

Table 1
0- to 3-simplexes and their common names, representations and geometric configurations.

Dimension	Name	Representation	Geometric configuration	
0	0-*simplex*	Node	$S_0 = \langle v_0 \rangle$	v_0
1	1-*simplex*	Edge	$S_1 = \langle v_0, v_1 \rangle$	 v_0 v_1
2	2-*simplex*	Triangle	$S_2 = \langle v_0, v_1, v_2 \rangle$	v_0 v_1 v_2
3	3-*simplex*	Tetrahedron	$S_3 = \langle v_0, v_1, v_2, v_3 \rangle$	v_0 v_1 v_2 v_3

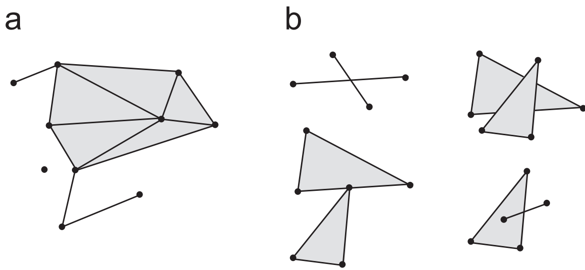


Fig. 4. (a) A simplicial complex that consists of 0-, 1- and 2-simplexes. (b) Few configurations of simplexes that are not simplicial complex, because they violate axioms.

3.2. Orientation of n-Simplexes

Vertexes of an n-simplex are ordered, which induces an orientation (either positive or negative) on the n-simplex. By convention, the orientation of a 0-simplex (node) is positive. The orientation of a 1-simplex (edge) is positive from vertex v_0 to vertex v_1 and negative from vertex v_1 to vertex v_0. For a 2-simplex (triangles), the orientation is defined based on the order in which the vertexes are listed: clockwise order is positive and counter-clockwise order is negative. The orientation of a 3-simplex (tetrahedron) is the sign of the volume constructed by its ordered vertexes (Alexandroff, 1961): based on the right-hand rule, a positive volume means that if the first three points are ordered so that they follow the direction of the curled fingers, then the thumb is pointing towards the 4th point.

Generally, the orientation of an n-simplex can be specified using the sign of the determinant of a matrix constructed as follows: for an n-simplex with vertexes $< v_0, \ldots, v_n >$, an element 1 is added to the end of each vertex and then they are arranged as the rows of a square matrix. For an n-simplex with vertexes $<(e_{01}, \ldots, e_{0n}), \ldots, (e_{n1}, \ldots, e_{nn})>$, the result is

$$\begin{vmatrix} e_{01} & \ldots & e_{0n} & 1 \\ e_{11} & \ldots & e_{1n} & 1 \\ \ldots & \ldots & \ldots & \ldots \\ e_{n1} & \ldots & e_{nn} & 1 \end{vmatrix} = \begin{vmatrix} e_{11}-e_{01} & \ldots & e_{1n}-e_{0n} \\ \ldots & \ldots & \ldots \\ e_{n1}-e_{01} & \ldots & e_{nn}-e_{0n} \end{vmatrix}$$

Non-negative values for this determinant indicate a positive orientation, while negative values mean a negative orientation. Similar to the determinant of a matrix, odd numbers of permutations of the vertexes of an n-simplex change the orientation, while even numbers of permutations maintain it unchanged (Stolfi, 1989). For instance, for the n-simplexes of Table 1:

$$S_0 = < v_0 >$$
$$S_1 = < v_0, v_1 > = - < v_1, v_0 >$$
$$S_2 = < v_0, v_1, v_2 > = - < v_0, v_2, v_1 > = < v_2, v_0, v_1 > = \ldots$$
$$S_3 = < v_0, v_1, v_2, v_3 > = - < v_0, v_1, v_3, v_2 > = < v_0, v_3, v_1, v_2 > = \ldots$$

3.3. Boundary of n-Simplexes

The boundary of the n-simplex $S_n = < v_0, \ldots, v_n >$, which is written as ∂S_n, is defined as follows:

$$\partial S_n = \sum_{i=0}^{n} (-1)^i < v_0, \ldots, \overline{v}_i, \ldots, v_n >$$

where $\overline{v}_i$ means omitting the vertex v_i from the vertex list. The boundary of an n-simplex is $n+1$ of $(n-1)$-simplexes (Stolfi, 1989):

- The boundary of a 0-simplex (node) is an empty set;
- The boundary of a 1-simplex (edge) is two 0-simplexes (nodes);
- The boundary of a 2-simplex (triangle) is three 1-simplexes (edges);
- The boundary of a 3-simplex (tetrahedron) is four 2-simplexes (triangles).

For instance, for the n-simplexes of Table 1:

$$\partial S_0 = \varphi$$
$$\partial S_1 = < v_1 > - < v_0 >$$
$$\partial S_2 = < v_1, v_2 > - < v_0, v_2, > + < v_0, v_1 >$$
$$\partial S_3 = < v_1, v_2, v_3 > - < v_0, v_2, v_3 > + < v_0, v_1, v_3 > - < v_0, v_1, v_2 >$$

3.4. Simplex-based definition of Bowyer–Watson algorithm

Using the concept of n-simplexes, an implementable version of the dimension independent definition of the *Bowyer–Watson* algorithm introduced at the start of this section is as follows:

Algorithm *Bowyer–Watson* (P)
Input: A set $P=\{p_0, \ldots, p_m\}$ of nD points ($m \geq n$)
Output: A homogenous simplicial n-complex D that is the n-dimensional Delaunay triangulation of P

1. $D \leftarrow$ a big n-simplex that contains all of the points $\{p_0, \ldots, p_m\}$
2. **for** all points $p \in P$
3. $S \leftarrow$ Set of all n-simplexes $e \in D$ whose circumsphere contains p
4. $B \leftarrow$ Set of $(n-1)$-simplexes that make the border of S
5. $N \leftarrow$ Set of n-simplexes constructed by adding p to all $(n-1)$-simplexes $b \in B$
6. $D \leftarrow \{D \backslash S\} \cup$ N
7. **return** D

This definition converts the *Bowyer–Watson* algorithm to a set of successive operations on n-simplexes. Dealing with n-simplexes and their operations is a case where the number of elements is not known:

- An nD point in the Euclidean space is represented by n numbers.
- An n-simplex is represented as a set of $(n+1)$ points of dimension n.
- The operations on an n-simplex can take any number of points as input each of which can have any number of elements, per se. The same applies to the output.

The situation is even worse if a number of operations are composed. In the next section, we introduce the *list* as an abstract data type that can model efficiently both of nD points and n-simplexes as well as their operations. Using the list has some advantages:

- Elements of a list can be from any type, so a list can model a point, an n-simplex (as a set of points), or even any other data structure that may be needed (e.g., a pair whose first and second elements are a point and a list of simplexes, respectively).
- A list can have any number of elements, so it can be used to model points and simplexes of any dimension.
- List operations are independent of the number of elements in the list, so the operations on points and simplexes can be equally used in any dimension.

The next section reviews the list data structure and some of its manipulating functions. We limit the functions to those that are required for the implementation of the operations on n-simplexes and dimension independent implementation of the *Bowyer–Watson* algorithm, which will be presented in Sections 5 and 6, respectively.

4. A review on *list* data structure

A list is a collection of any number of elements of the same type. For instance, all of the following collections are lists:

[1,2,7,5,1,4] → [Int]
['*a*','*c*','*a*','*d*','*k*'] → [Char]
[(1,4),(5,3),(9,3),(6,2),(3,3)] → [(Int, Int)]
[True, True, False, True, False, False] → [Bool]
[[1,2,7],[5,1],[8,5,2,9],[3],[5,1]] → [[Int]]

where [*a*] means a list of values of type '*a*'. The order of elements in a list is significant: [1, 2, 3] is different from [3, 2, 1], so we can talk about the first, the second, … and the last elements of a list. The number of occurrences of an element does also matter: [3] contains one element and [3, 3] contains two, which happen to be the same.

The operator ':', called *list constructor*, builds a list from an element and a list. Thus

[1,2,7] = 1 : [2,7] = 1 : 2 : [7] = 1 : 2 : 7 : []

Table 2
Some of standard manipulating functions over lists.

Function and syntax	Description	Example
length (x)	Returns the number of elements in the list *x*	length([2,3,4]) = 3 length([]) = 0
x++y	Concatenates two lists	[2,3,4] + +[4,5] = [2,3,4,4,5] [2,3,4] + +[] = [2,3,4]
concat (x)	For the list of lists *x*, puts all elements together in a single list	concat([1,2],[2,3,4],[3,4,5,6,[7,8]]) = [1,2,2,3,4,3,4,5,6,7,8]
concatMap (f, x)	For the list of lists *x*, applies the function *f* to all elements of *x* and then puts them together in a single list	concat Map(sum,[[1,2],[3,4,5],[5,6]]) = [3,12,11]
sum (x)	Calculates the sum of all elements of the list *x*	sum([1,2,3,4]) = 10
map (f, x)	Applies the function *f* to every elements of the list *x*	map((+2),[1,2,3]) = [3,4,5]
filter (c, x)	Returns all elements of the list *x* that fulfill the condition *c*	filter((>2),[1,2,3,4]) = [3,4] filter((= =2),[1,2,3,4]) = [2]
fold (f, a, x)	Combines the elements of the list *x* with the specified function *f* and the start value *a* (e.g., add all elements)	fold((+),0,[1,2,3,4]) = 10 fold((*),1,[1,2,3,4]) = 24
once (eq, x)	Returns the values that appear only once in the list *x* regarding the *eq* definition of equality	once((= =),[1,2,2,3,2,4,3,5,4,6]) = [1,5,6]
x\\y	Drops elements of the list *x* that exist in the list *y*, i.e., *x–y*	[1,2,3,4]\\[3,4,5] = [1,2]
sort (x)	Sorts the elements of the list *x*	sort([3,4,5,1,2,3,1,5,6,3]) = [1,1,2,3,3,3,4,5,5,6]
removeEach (x)	Returns a list of lists in *i*th element of which the *i*th element of *x* has been removed	remove Each([2,3,4,5]) = [[3,4,5],[2,4,5],[2,3,5],[2,3,4]

The example shows that every non-empty list is built from an empty list [] by the repeated use of the list constructor ':'. This characteristic is used to define most of the functions over list, recursively (see Appendix 1). Table 2 presents a set of manipulating functions over lists that we will use in the following sections. Their implementations are presented in the Appendix 1, except the ones that need some other functions as prerequisites. A complete list of standard manipulating functions over lists and their implementations can be found in (Thompson, 1999).

Some of the functions in the table (e.g., map and filter) have another function as input. Such functions are called second order functions and are efficiently handled in functional programming languages (Bird and Wadler, 1988; Thompson, 1999). Therefore, we have used such an environment for the implementations. However, no familiarity with functional languages is considered in this article and the mathematical equivalences of the functions are presented.

5. Using lists to implement *n*-simplexes and their primitive operations

This section implements the n-simplexes and their operations using the list data structure. Next section will use these materials to implement the *Bowyer–Watson* algorithm for Delaunay triangulation computation.

We begin by defining a type for nD points. In the n-dimensional Euclidean space, a point is represented by its coordinates in the Cartesian coordinate system. In 2D and 3D

spaces, they are usually defined as pairs (x, y) and triples (x, y, z), respectively. To have an n-dimensional representation, here we use a list of floating numbers:

Point :: [Float]

where :: presents the signature of the data type *point*. Definition of a vertex is the same as a point, i.e., the *Point* data type can be used equally for a vertex. However, their equality is explicitly indicated here, for the sake of clarity:

Vertex = Point

Then an n-simplex is a list of vertexes:

Simplex :: [Vertex]

Dimension of a point is the number of defining elements. For an n-simplex it is the number of its vertexes − 1. Thus, they can be specified using the *length* of a list:

ptDim (s) = length (s)

simpDim (s) = length (s)−1

The first operation is the orientation of an n-simplex, which uses the determinant of the matrix introduced in Section 3.2. We create the required matrix and calculate its determinant:

orn (s) = det (mat) > 0
  where
  mat = map ((1 :), s)

where *det* is a function that calculates the determinant of a square matrix. Note that '*map*((1:), s)' is a function that adds the value '1' to the front of each element of the list s.

For switching the orientation of an n-simplex, the first and the second elements are swapped, which change the sign of the above determinant:

switchOrn ([]) = []
switchOrn ([v]) = [v]
switchOrn ((v_1 : v_2 : vs)) = (v_2 : v_1 : vs)

The way we defined the function *switchOrn*, and it will be used henceforth as well, is called *pattern matching*:

- If the input of the *switchOrn* is null, the output is null, too.
- If the input is an one-element list [v], the output is [v], too.
- If the input has the form (v_1:v_2:vs) − where v_1 is the first, v_2 is the second and *vs* is the rest of the elements of the input list − the output is (v_2:v_1:vs).

This is as if we used the following mathematical notation to define *switchOrn*:

$$switchOrn(x) = \begin{cases} [] & \text{for} \quad x = [] \\ [v] & \text{for} \quad x = [v] \\ (v_2 : v_1 : vs) & \text{for} \quad x = (v_1 : v_2 : vs) \end{cases}$$

The checks whether two n-simplexes have the same vertexes or orientation are

eqVs (s_1, s_2) = sort (s_1) == sort (s_2)

eqOrn (s_1, s_2) = orn (s_1) == orn (s_2)

Thus, the equality of two n-simplexes (i.e., consisting of the same vertexes and having the same orientation) is defined as follows:

eqSimps (s_1,s_2) = eqVs (s_1, s_2) ∧ eqOrn (s_1, s_2)

The boundary operation for an n-simplex is implemented as

boundary (s) = setOrn . removeEach (s)

setOrn ([]) = []
setOrn ([s]) = [s]
setOrn (s1 : s2 : ss) = s1 : switchOrn (s2) : setOrn (ss)

where '.' means function composition, i.e., applying the first function to the result of applying the second: *f.g* (x)=f(g(x)). In this definition, *removeEach* (vs) creates the boundary (n−1)-simplexes and *setOrn* is a recursive function that switches the orientation of the even elements.

6. Dimension independent implementation of *Bowyer–Watson* algorithm

This section aims to implement the dimension independent *Bowyer–Watson* algorithm presented in Section 3.4. It needs a number of spatial analyses, which will be implemented first.

The first required operation is *addVertex*, which adds a vertex to an n-simplex (Fig. 5):

addVertex(v,s) = (v : s)

Another required operation is the *border* operation. As Fig. 6 shows, this operation extracts the bordering (n−1)-simplexes from a set of connected n-simplexes (A set of n-simplexes S={s_1, s_2, …, s_m} are connected if and only if for each $s_i \in S$, there is at least one $s_j \in S(i \neq j)$ such that $s_i \cap s_j$ is an (n−1)-simplex). Note the difference between this operation and the *boundary* operation, which extracts the boundary of an individual n-simplex.

To implement this operation, we use the fact that bordering simplexes appear once and only once (simplexes with the same vertexes are considered equal here). Thus, to get the bordering simplexes, we extract and concatenate the boundaries of all n-simplexes and then take the simplexes that appear once in this list:

border (s) = once (eqVs) . (concatMap (boundary)) (s)

Note that if f is a function of n variables v_1, …, v_n, i.e., f(v_1, …, v_n), then f(v_1, …, v_{n-1}) is a unary function of v_n. Therefore

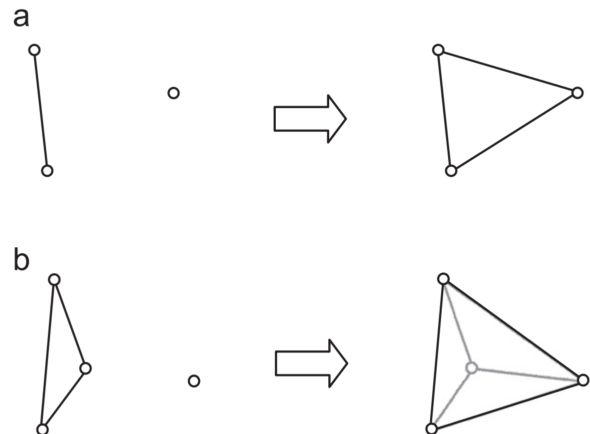


Fig. 5. Functionality of addVertex operation for 1- and 2-simplexes: a new vertex is added to (a) 1-simplex and (b) 2-simplex.

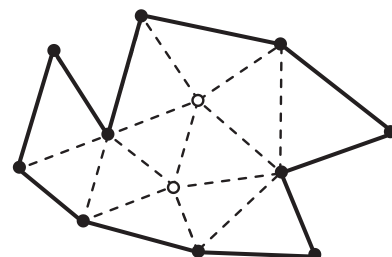


Fig. 6. Functionality of border operation for a set of connected 2-simplexes (dotted triangles), which results in their bordering 1-simplexes (bold edges).

(*concatMap* (*boundary*)) is a function that has already been given its first input and now it is a function of its second input.

The test whether an nD point is inside the nD circumsphere of an nD pyramid is achieved by using the sign of the determinant of a matrix constructed as follows: for an nD point $(e_1, \ldots, e_n)$ and an n-simplex with vertexes $\langle v_0, \ldots, v_n \rangle$, the sum of square of all points (including the nD points and vertexes of the n-simplex) and an element "1" are added to the end of each vertex and then they are arranged as the rows of a square matrix. For an n-simplex with vertexes $\langle (e_{01}, \ldots, e_{0n}), \ldots, (e_{n1}, \ldots, e_{nn}) \rangle$, the result is

$$\begin{vmatrix} e_1 & \ldots & e_n & {e_1}^2+\ldots+{e_n}^2 & 1 \\ e_{01} & \ldots & e_{0n} & {e_{01}}^2+\ldots+{e_{0n}}^2 & 1 \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ e_{n1} & \ldots & e_{nn} & {e_{n1}}^2+\ldots+{e_{nn}}^2 & 1 \end{vmatrix} = \begin{vmatrix} e_{01}-e_1 & \ldots & e_{0n}-e_n & (e_{01}-e_1)^2+\ldots+(e_{0n}-e_n)^2 \\ \ldots & \ldots & \ldots & \ldots \\ e_{n1}-e_1 & \ldots & e_{nn}-e_n & (e_{n1}-e_1)^2+\ldots+(e_{nn}-e_n)^2 \end{vmatrix}$$

Positive values for this determinant indicate that the point is inside the nD circumsphere, while negative values mean that the point is outside. If the determinant is zero, all of the points (the new point and vertexes of the n-simplex) are co-spherical and both configurations can be equally used (here, we leave it as it is). Implementation of this test is

```
inSphere (p,s) = det (mat) > = 0
  where
  mat = map (tr (s))
  tr (x) = dx++[sum.map (sq(dx))]
  dx = x−p
  sum (x) = fold ((+), 0, x)
```

Having implemented all of the required data types and spatial analyses independent of dimension, the implementation of the *Bowyer–Watson* algorithm presented in Section 3.4 is as follow:

```
Delaunay(pts) = fold(updateDT,bigSimp,pts)
  where
    bigSimp = ...(some simple computations left for readers)
updateDT(dt,pt) = (dt\\s)++n
  where
    s = filter(inSphere,pt,dt)
    n = map(addVertex(pt),border (s))
```

The complexity of the described algorithm is $O(n^2)$ in 2D, where n is the number of input points: The number of triangles is kn (k is a constant); The procedure runs for each point and in each iteration, all of the existing rn triangles (r is a constant) are checked against the new point (*InSphere* test), which makes the complexity $O(n^2)$. In 3D, however, the complexity is $O(n^3)$, because some configurations of n points in 3D have kn^2 tetrahedra, so in each iteration, *InSphere* test runs for all of the existing rn^2 tetrahedra (r is a constant).

To improve the efficiency of the algorithm, two steps are taken:

- Instead of running *InSphere* test on all of the triangles, we use the fact that the violating n-simplexes are connected, so after detecting the first violating n-simplex, its adjacent n-simplexes are checked and it continues until all of the adjacent n-simplexes satisfy the test. This modification is also required to prevent degenerate cases, where roundoff error may cause creating more than one hole (Field, 1986).
- If *InSphere* test is used to find the first violating n-simplex, all of the n-simplexes must be checked in the worst case and so it destroys the efficiency achieved in the first step. Instead, a strategy called *walking* is used to detect the n-simplex that contains the new point, which certainly violates the empty sphere rule: Using the orientation test, the position of the new point P is checked against the first n-simplex S. If the point P is not inside the S, the next n-simplex that is checked is the adjacent of a boundary of S that has the point P on the wrong side (Mostafavi et al., 2003). By repeating this procedure, we walk directly toward the n-simplex that contains the point P. This algorithm finds the first violating n-simplex in $k \log n$ and kn iterations for 2D and 3D, respectively (k is a constant).

In order to support the above improvements, the following data structure is used, which stores adjacency information of n-simplexes:

$i\ V_1\ \ldots\ V_n\ S_1\ \ldots\ S_n$

where i is the index of the n-simplex, V_is are the vertexes of the n-simplex i, and S_k is the index of the opposite n-simplex to V_k. Thus, after detecting the first violating n-simplex, its adjacent n-simplexes are detected and checked in a constant time.

The above improvements reduce the overall complexity to $O(n \log n)$ in 2D and $O(n^2)$ in 3D. These modifications were applied in the code, but not explained here in order to keep the implementations simple.

7. Implementation results

The definitions given in Sections 5 and 6 were implemented using the functional programming language Haskell (Thompson, 1999). Some 2D and 3D points, whose positions are presented in the Appendix 2, were used as examples. Fig. 7 illustrates the results of the Delaunay triangulation of these points.

To investigate the efficiency of the implementations, the program was executed with different numbers of randomly generated 2D and 3D points (Table 3 and Fig. 8). The results

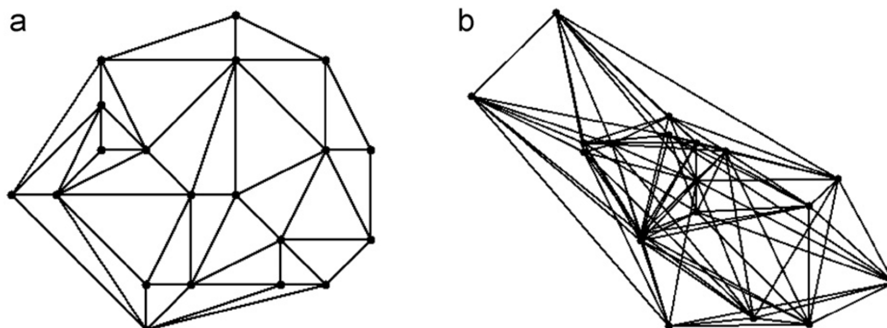


Fig. 7. Delaunay triangulation of example points presented in Appendix 2. (a) 2D points and (b) 3D points.

Table 3
Running time as a function of number of input points for 2D and 3D Delaunay triangulation.

Number of points		10	50	250	500	1000	2000	4000	8000	16,000	32,000	64,000
Time (s)	2D	0.01	0.02	0.14	0.33	0.74	1.67	3.67	7.83	17.12	36.70	78.00
	3D	0.01	0.06	0.15	0.32	1.17	2.76	7.55	21.49	60.63	212.95	873.69

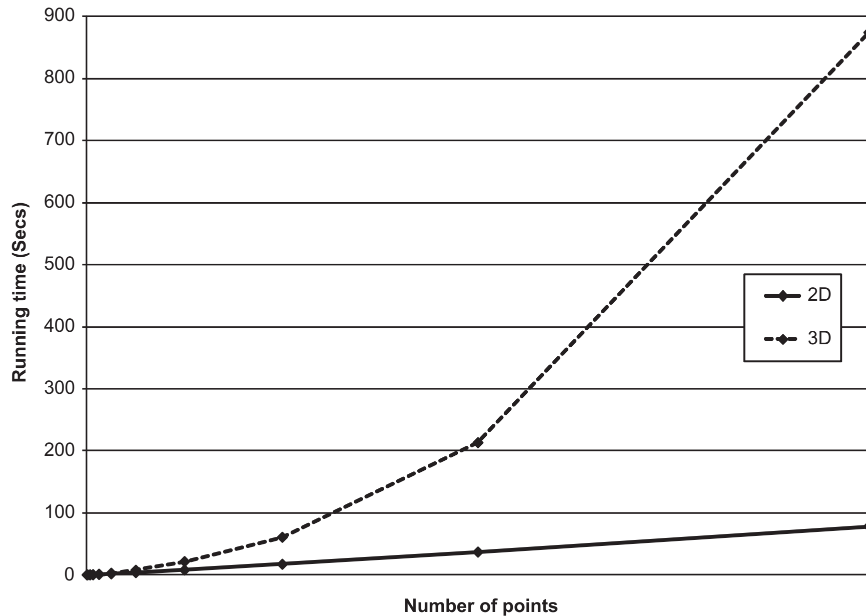


Fig. 8. Running time as a function of number of input points for 2D and 3D Delaunay triangulation.

Fig. 9. Satellite image of Latyan dam and its reservoir.

show that the running times goes as $O(n \log n)$ and $O(n^2)$ for 2D and 3D points, respectively, as it was expected. They also illustrate that for the same number of points, the running time for 3D is larger than 2D. This is because of the more and bigger size of the matrixes that must be dealt with in 3D: To check if a point is inside a tetrahedron (the case for 3D) it computes four 4×4matrixes, while this is three 3×3matrixes for a point against a triangle (the case for 2D), or to check if a point is inside the circumsphere of a tetrahedron (the case for 3D) it computes a 4×4matrix, while this is a 3×3matrix for a point against the circumcircle of a triangle (the case for 2D). These tests are frequently used in *Bowyer–Watson* algorithm. In abstract, the running time is a function of the number of n-simplexes as well as the size of the computation units, which depends on the dimension.

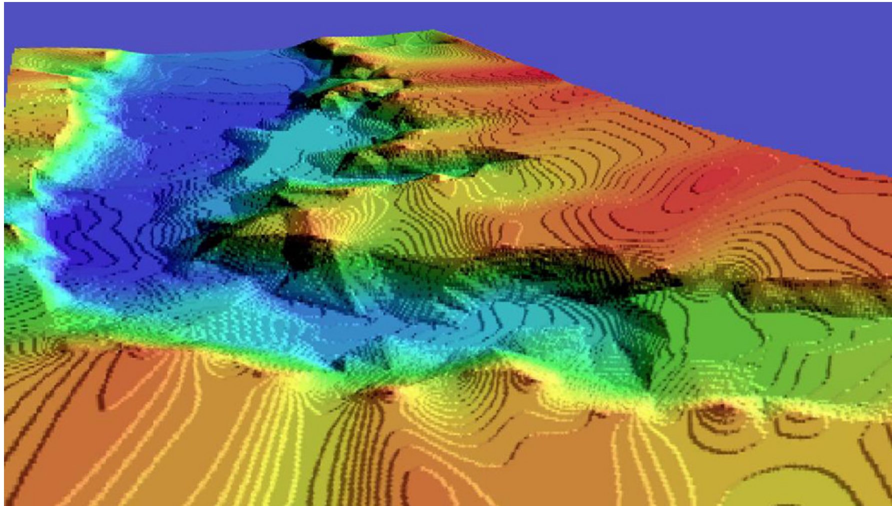


Fig. 10. 3D view of Latyan dam and its reservoir.

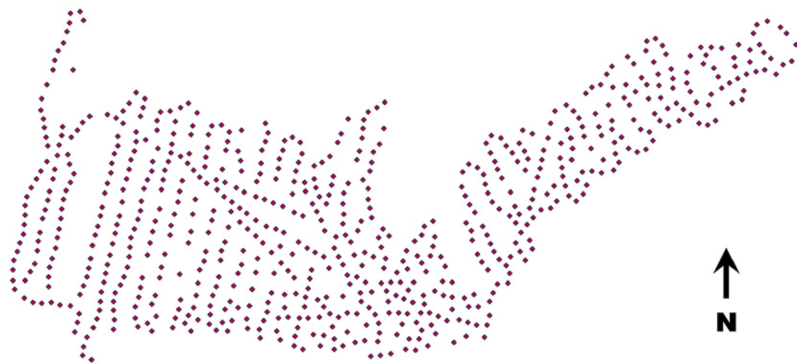


Fig. 11. Points resulted from hydrography of Latyan dam reservoir.

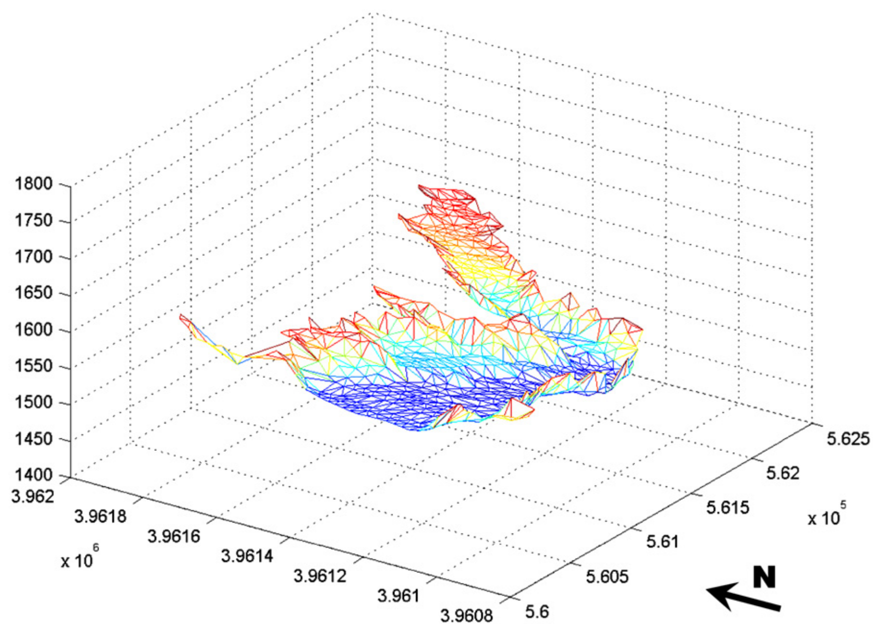


Fig. 12. 3D TIN of Latyan dam reservoir.

8. Case study

In this section, the dimension independent implementation of Delaunay triangulation is used to calculate the area and volume of the reservoir of a dam at different water levels, which leads to a level–surface–volume diagram. This diagram is important for managing the water consumption and monitoring the dam construction: observing the daily water level, this diagram is used to estimate the surface area and water amount of the reservoir. This information helps the decision makers in applications like water usage allocation, dam deformation control and managing water release behind the dam.

The Latyan dam – located in North East of Tehran, Iran – was selected as the case study (Figs. 9 and 10). The bed of the dam reservoir was surveyed in a hydrographic process (Fig. 11) and its 3D TIN was produced (Fig. 12). To calculate the area and volume of the reservoir at a certain water level, say h, the 3D TIN was intersected with the plan $z=h$, which results in the volume of the reservoir where $z<h$ and the surface of the reservoir at $z=h$. Fig. 13 shows the results for the water level of 1570 m. To calculate the area and volume of the results, the implemented n-dimensional Delaunay triangulation was used: For a convex nD structure, it is triangulated to a set of n-simplexes and then sum of the nD-volume (i.e., area for 2D, volume for 3D, etc.) of the components is calculated. The absolute value of the determinant used to specify the orientation of an n-simplex yields its nD-volume:

$$\mathrm{vSimp}\,(s) = \mathrm{abs}\,(\det(\mathrm{map}((1:), s)))$$

$$\mathrm{vConv}\,(p) = \mathrm{sum}\ .\ \mathrm{map}\,(\mathrm{vSimp})\ .\ \mathrm{dt}\,(p)$$

As Fig. 13 shows our structures are non-convex. Therefore, first they must be decomposed to a set of convex components and then the above calculation is applied separately to each component. For this, a dimension independent decomposition of polytopes was implemented (for implementation details of this analysis, see Bulbul et al., 2009). Each component is triangulated

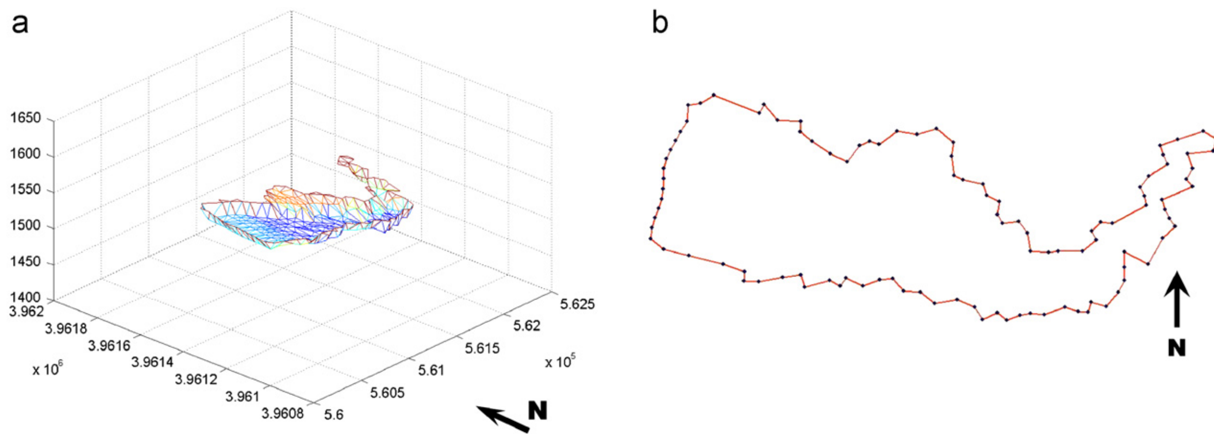


Fig. 13. 3D TIN and surface of Latyan dam reservoir at water level of 1570 m.

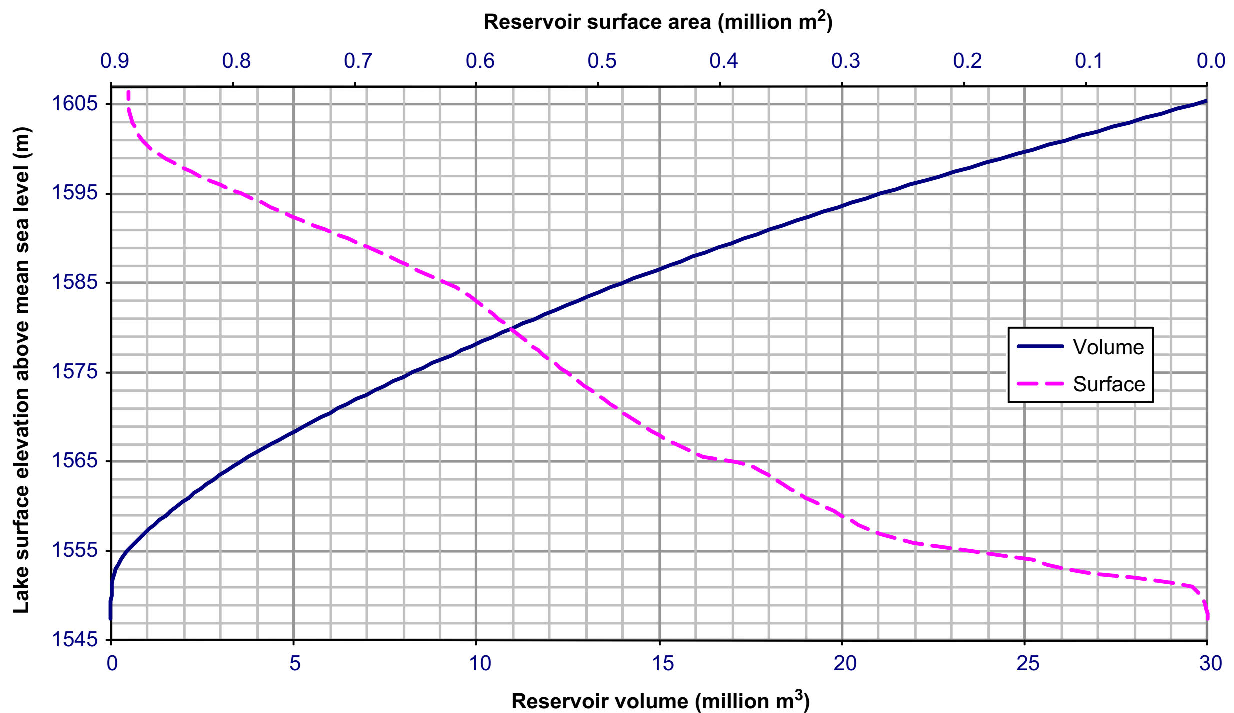


Fig. 14. Level–Surface–Volume diagram of Latyan dam reservoir.

using the implemented n-dimensional Delaunay triangulation. Calculating the area/volume of each component and summing up the results will provide us with the total area/volume of the reservoir at the desired water level. The function that takes an n-dimensional polytope and calculates its nD-volume is as follow:

$$\mathrm{vPoly}\,(p) = \mathrm{sum}\ .\ \mathrm{map}\,(\mathrm{vConv})\ .\ \mathrm{decompose}\,(p)$$

By applying the explained process for different water levels, the level–surface–volume diagram was produced for the reservoir of the Latyan dam that shows the surface area and volume of the reservoir at different water levels (Fig. 14).

9. Conclusions and future work

Providing spatial analyses for 3D objects is an essential requirement for many geoscientific applications. The current approach is to use separate implementations of spatial analyses for 2D and 3D objects. This article pushes more toward dimension independency in spatial analyses. It shows how to have not only a dimension independent algorithm, but also a dimension independent implementation. It shows the elegancy of the *list* data structure and its manipulating functions, which are treated efficiently in functional programming languages. The goal here is to show how to use the same code for applying a spatial analysis (Delaunay triangulation, here) to spatial objects of any dimension. The results for the case study that produces the level–surface–volume diagram of the reservoir of the Latyan dam show the beauty of the approach.

The same approach can be applied to other spatial analyses. We have used this approach to implement decomposition of non-convex polytopes of any dimensions as a set of convex ones, from which binary operations (e.g., intersection, union, difference, etc.) on polytopes - which are frequently used in geoscientific applications such as geological modeling - will follow.

Appendix 1. Implementation of some of the functions over lists presented in Table 2.

Table 4

Table 4
Implementation of some of functions over lists presented in Table 2.

length	length([]) = 0 length((x : xs)) = a + length(xs)
++	[] ++ y = y (x : xs) ++ y = x : (xs + y)
concat	concat(x) = fold((++),[],x)
concatMap	concatMap(f,x) = concat.map(f,x)
sum	sum(x) = fold((+),0,x)
map	map(f,[]) = [] map(f,(x : xs)) = f(x) : map(f,xs)
filter	filter(f,[]) = [] filter(f,(x : xs)) = if f(x) == true then x : filter(f,xs) else filter(f,xs)
fold	fold(f,a,[]) = a fold(f,a,(x : xs)) = fold((f a x),xs)
sort	sort(x : xs) = sort(filter(< x)xs) ++ filter(== x)xs ++ sort(filter(> x)xs)

Appendix 2. Positions of the example points used in Section 7.

- 2D points

  [3,4]; [1,3]; [4,1]; [8,1]; [7,2]; [9,2]; [5,3]; [8,4]; [6,3]; [5,1]; [3,4]; [6,7]; [6,3]; [8,6]; [3,5]; [4,0]; [7,1]; [2,3]; [3,6]; [9,4]

- 3D points

  [1,2,1]; [6,2,1]; [4,2,5]; [4,5,6]; [3,3,2]; [3,1,2]; [1,3,4]; [8,4,2]; [9,1,4]; [4,5,4]; [8,6,7]; [5,4,3]; [9,2,6]; [5,6,8]; [3,1,4]; [2,8,6]; [8,4,2]; [1,6,8]; [9,3,9]; [9,1,1]

References

Abdul-Rahman, A., Pilouk, M., 2008. Spatial Data Modeling for 3D GIS. Springer-Verlag, Berlin, Heidelberg 289pp.

Alexandroff, P., 1961. Elementary Concepts of Topology. Dover Publications, New York 57pp.

Apel, M., 2004. A 3D geoscience information system framework. Ph.D. Dissertation, Technische Universitaet Freiberg, Freiberg, Germany, 105pp.

Aurenhammer, F., 1991. Voronoi diagrams—a survey of a fundamental geometric data structure. ACM Computing Surveys 23 (3), 345–405.

Bird, R., Wadler, P., 1988. Introduction to Functional Programming. Prentice Hall, Hertfordshire, UK 293pp.

Bowyer, A., 1981. Computing Dirichlet tessellation. The Computer Journal 24 (2), 162–166.

Bulbul, R., Karimipour, F., Frank, A.U., 2009. A simplex-based dimension independent approach for convex decomposition of nonconvex polytopes. In: Proceedings 10th International Conference on GeoComputation, Sydney, Australia, 8pp. 〈http://www.biodiverse.unsw.edu.au/geocomputation/proceedings/PDF/Bulbul_et_al.pdf〉.

Delaunay, B.N., 1934. Sur la Sphere Vide (On the empty sphere). Izvestia Akademia Nauk SSSR, Otdelenie Matematicheskii i Estestvennyka Nauk 7, 793–800.

Field, D.A., 1986. Implementing Watson's algorithm in three dimension. In: Proceedings 2nd Annual Symposium on Computational Geometry, New York, USA, pp. 246–259.

Frank, A.U., 1999. One step up the abstraction ladder: combining algebras—from functional pieces to a whole. In: Freksa, C., Mark, D.M. (Eds.), Proceedings International Conference COSIT'99, Stade, Germany, Lecture Notes in Computer Sciences, vol. 1661. Springer-Verlag, pp. 95–107.

Frank, A.U., Kuhn, W., 1986. Cell graphs: a provable correct method for the storage of geometry. In: Marble, D. (Ed.), Proceedings 2nd International Symposium on Spatial Data Handling, pp. 411–436.

Guibas, L., Stolfi, J., 1985. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. ACM Transactions on Graphics 4 (2), 74–123.

Hatcher, A., 2002. Algebraic Topology. Cambridge University Press, Cambridge, UK 544pp. accessed 30 April 2010.

Houlding, S.W., 1994. 3D Geoscience Modeling: Computer Techniques for Geological Characterization. Springer-Verlag, Berlin 309pp.

Karimipour, F., Delavar, M.R., Frank, A.U., 2008a. A mathematical tool to extend 2D spatial operations to higher dimensions. In: Proceedings of the International Conference on Computational Science and Its Applications (ICCSA 2008), Perugia, Italy, Lecture Notes in Computer Science, vol. 5072, Springer-Verlag, pp. 153–164.

Karimipour, F., Frank, A.U., Delavar, M.R., 2008b. An operation-independent approach to extend 2D spatial operations to 3D and moving objects. In: Proceedings of the 16th ACM (Association for Computing Machinery) SIGSPATIAL International Conference on Advances in Geographic Information Systems (ACM GIS 2008), Irvine, CA, USA, pp. 51–56.

Lattuada, R., Raper, J., 1995. Applications of 3D Delaunay triangulation algorithms in geoscientific modeling. In: Proceedings of the 3rd National Conference on GIS Research UK, Newcastle, UK, pp. 150–153. 〈http://geog.bbk.ac.uk〉 (accessed 30 April 2010).

Ledoux, H., 2006. Modelling three-dimensional fields in geo-science with the Voronoi diagram and its dual. Ph.D. Dissertation, University of Glamorgan, Pontypridd, Wales, UK, 168pp.

Ledoux, H., 2007. Computing the 3D Voronoi diagram robustly: an easy explanation. In: Proceedings of the 4th International Symposium on Voronoi Diagrams in Science and Engineering, Pontypridd, Wales, UK, pp. 117-129.

Ledoux, H., Gold, C., 2007. The 3D Voronoi diagram: a tool for the modelling of geoscientific datasets. In: Proceedings of the GeoCongres 2007, Quebec, Canada, 13pp. 〈http://www.gdmc.nl/publications/2007/3D_Voronoi_Diagram_Tool.pdf〉.

Monga, O., Ngom, F.N., Delerue, J.F., 2007. Representing geometric structures in 3D tomography soil images: application to pore-space modeling. Computers & Geosciences 33, 1140–1161.

Mostafavi, M.A., Gold, C., Dakowicz, M., 2003. Delete and insert operations in Voronoi/Delaunay methods and applications. Computers & Geosciences 29, 523–530.

Nonato, L.G., Minghim, R., Oliveira, M.C.F., Tavares, G., 2001. A novel approach for Delaunay 3D reconstruction with a comparative analysis in the light of applications. Journal of Computer Graphics 20 (2), 1–14.

Okabe, A., Boots, B., Sugihara, K., Chiu, S.N., 2000. Spatial Tessellations: Concepts and Applications of Voronoi Diagrams second ed. John Wiley, Chichester, UK 671pp.

O'Rourke, J., 1998. Computational Geometry in C second ed. Cambridge University Press, Cambridge, UK 350pp.

Penninga, F., van Oosterom, P.V., 2008. A simplicial complex-based DBMS approach to 3D topographic data modeling. International Journal of Geographical Information Science 22 (7), 751–779.

Pouliot, J., Bedard, K., Kirkwood, D., Lachance, B., 2008. Reasoning about geological space: coupling 3D geomodels and topological queries as an aid to spatial data selection. Computers & Geosciences 34, 529–541.

Preparata, F.P., Hong, S.J., 1997. Convex hulls of finite sets of points in two and three dimensions. Communications of the ACM 20 (2), 87–93.

Raper, J., 2000. Multidimensional Geographic Information Science. Taylor & Francis, London 317pp.

Schneider, M., 1997. Spatial Data Types For Database Systems—Finite Resolution Geometry For Geographic Information Systems, Lecture Notes in Computer Sciences. Berlin-Heidelburg, 1288. Springer-Verlag 275pp.

Stolfi, J., 1989. Primitives for computational geometry. Ph.D. Dissertation, Computer Science Department, Stanford University, Palo Alto, CA, 228pp.

Thompson, S., 1999. Haskell: The Craft of Functional Programming. Addison-Wesley, Dorchester, UK 500pp.

Turner, K. (Ed.), 1992. Three Dimensional Modelling with Geoscientific Information Systems. Kluwer Academic Publishers, Dordrecht, The Netherlands.

Tzenkov, T., Gospodinov, S., 2003. Geometric analysis of geodetic data for investigation of 3D landslide deformations. Natural Hazards Reviews 4 (2), 78–81.

Watson, D.F., 1981. Computing the *n*-dimensional Delaunay tessellation with application to Voronoi polytopes. The Computer Journal 24, 167–172.

Yanbing, W., Lixin, W., Wenzhong, S., Xiaomeng, L., 2007. On 3D GIS spatial modeling. In: Proceedings of the ISPRS Workshop on Updating Geo-spatial Databases with Imagery and the 5th ISPRS Workshop on DMGISs, Urumchi, Xinjiang, China, pp. 237–240.

Yong, X., Sun, M., Ma, A., 2004. On the reconstruction of three-dimensional complex geological objects using Delaunay triangulation. Journal of Future Generation Computer Systems 20, 1227–1234.