Received 27 December 2023, accepted 10 January 2024, date of publication 15 January 2024, date of current version 26 January 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3354709

SURVEY

A Comprehensive Survey on Delaunay Triangulation: Applications, Algorithms, and Implementations Over CPUs, GPUs, and FPGAs

YAHIA S. ELSHAKHS[1], **KYRIAKOS M. DELIPARASCHOS**[1], **(Member, IEEE)**, **THEMISTOKLIS CHARALAMBOUS**[2,3], **(Senior Member, IEEE)**, **GABRIELE OLIVA**[4], **(Senior Member, IEEE)**, **AND ARGYRIOS ZOLOTAS**[5], **(Senior Member, IEEE)**

[1]Department of Electrical Engineering and Computer Engineering and Informatics, Cyprus University of Technology, 50329 Limassol, Cyprus
[2]Department of Electrical and Computer Engineering, University of Cyprus, 1678 Nicosia, Cyprus
[3]Department of Electrical Engineering and Automation, Aalto University, FI-00076 Espoo, Finland
[4]Departmental Faculty of Engineering, Campus Bio-Medico University of Rome, 00128 Rome, Italy
[5]Centre for Autonomous and Cyber-Physical Systems, School of Aerospace, Transport and Manufacturing, Cranfield University, MK43 0AL Cranfield, U.K.

Corresponding author: Gabriele Oliva (g.oliva@unicampus.it)

ABSTRACT Delaunay triangulation is an effective way to build a *triangulation* of a cloud of points, i.e., a partitioning of the points into simplices (triangles in 2D, tetrahedra in 3D, and so on), such that no two simplices overlap and every point in the set is a vertex of at least one simplex. Such a triangulation has been shown to have several interesting properties in terms of the structure of the simplices it constructs (e.g., maximising the minimum angle of the triangles in the bi-dimensional case) and has several critical applications in the contexts of computer graphics, computational geometry, mobile robotics or indoor localisation, to name a few application domains. This review paper revolves around three main pillars: (I) algorithms, (II) implementations over *central processing units* (CPUs), *graphics processing units* (GPUs), and *field programmable gate arrays* (FPGAs), and (III) applications. Specifically, the paper provides a comprehensive review of the main state-of-the-art algorithmic approaches to compute the Delaunay Triangulation. Subsequently, it delivers a critical review of implementations of Delaunay triangulation over CPUs, GPUs, and FPGAs. Finally, the paper covers a broad and multi-disciplinary range of possible applications of this technique.

INDEX TERMS Delaunay triangulation, applications of Delaunay triangulation, algorithmic approaches to Delaunay triangulation, CPU implementation of Delaunay triangulation, GPU implementation of Delaunay triangulation, FPGA implementation of Delaunay triangulation, Voronoi diagram, CPU, GPU, FPGA.

I. INTRODUCTION

Given a set of data points, there are several ways to arrange them into a triangulated mesh as in Fig. 1, but not all possible triangle combinations can result in a favourable representation of the spatial relationships between points. In 1934, Boris Delaunay published his work titled "Sur la sphère vide: A la mémoire de Georges Voronoï" [2], in which he proposed a geometric algorithm for constructing a triangulated mesh from a set of points. The resulting triangulation is unique and has several useful properties that make it attractive for a wide variety of applications. For instance, it offers both an angle guarantee and a low latency during implementation, which gives it much potential in different fields [3] [4]. As a consequence, aided by the rapid development of technology, the method became one

The associate editor coordinating the review of this manuscript and approving it for publication was Walter Didimo.

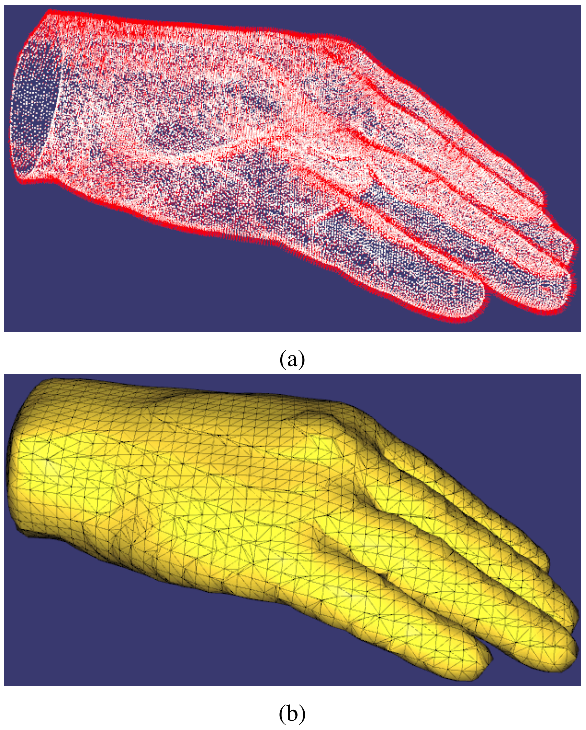


FIGURE 1. **Point cloud (Panel 1a) to Mesh (Panel 1b). Adapted from Wikimedia commons [1].**

of the essential algorithms in computational geometry [5]. Since then, Delaunay triangulation has become extremely popular, with numerous applications in computer vision [6], [7], medical imaging [8], [9], mobile robotics [10], [11] or indoor localisation [12], [13], to name a few scenarios. Such a tool essentially amounts to the triangulation of a set of points such that no point is inside the circumcircle of any triangle. In the literature, several algorithms have been developed in order to compute Delaunay triangulations, e.g., [14], [15], and [16]. Moreover, in the last few years, the increasing demand for high-performance computing has led to the implementation of Delaunay triangulation algorithms on different hardware platforms, including CPUs [17], [18], [19], [20], [21], GPUs [22], [23], [24], [25], and FPGAs [26], [27], [28], [29], [30], [31]. These implementations aim to improve the computational efficiency and reduce the running time of triangulation algorithms, especially for large data sets. In particular, the parallelisation of Delaunay triangulation algorithms on GPUs and FPGAs has shown promising results, providing a significant speedup compared to the traditional CPU implementation.

A. CONTRIBUTION

In the literature, there have been attempts to survey the different approaches for computing Delaunay Triangulation [32], [33], [34], [35], [36], [37], [38], [39], [40]. Table 1 compares the present survey with respect to previous attempts, including also the most significant textbooks, Ph.D. theses, and reports. Notably, only a few works provide an exhaustive discussion on the algorithms [34], [35], [36], [38] (e.g., by providing a pseudocode and/or by discussing its computational complexity and properties), while most surveys only briefly discuss the mechanisms underlying the different algorithms, while focusing on different aspects such as the refinement of an existing triangulation [35], [37]. Also, only [36] focuses on the implementation over CPUs and GPUs. Finally, most of the above surveys do not cover applications, while only a few of them consider a single application domain (i.e., [39] focuses on computer vision and [33] on multibeam echosounding).

TABLE 1. **Comparison with previous survey papers on Delaunay triangulation.**

	Algorithms	Exhaustive	2D	3D	>3D	CPU	GPU	FPGA	Application Domains
Fortune, 2017 [40]	2	–	✓	✓	✓	–	–	–	–
Dinas and Banon, 2014 [39]	6	–	✓	–	–	–	–	–	1
Aurenhammer et al., 2013 [38]	6	✓	✓	✓	✓	✓	–	–	–
Gonzaga de Oliveira, 2012 [37]	–	–	–	–	–	–	–	–	–
Nanjappa, 2012 [36]	7	✓	✓	✓	✓	✓	✓	–	–
Cheng et al., 2012 [35]	3	✓	✓	✓	✓	–	–	–	–
Hjelle and Dæhlen, 2006 [34]	6	✓	✓	–	–	–	–	–	–
Brouns et al., 2003 [33]	6	–	✓	–	–	–	–	–	1
Maur, 2002 [32]	5	✓	–	✓	–	–	–	–	–
This survey	**8**	✓	✓	✓	✓	✓	✓	✓	**12**

Unlike other works in the literature, the present survey paper aims at providing a holistic overview of Delaunay Triangulation, presenting:

- A comprehensive review and a tutorial-like presentation of all the main algorithms. Connections with Voronoi tessellation are also made.
- The state-of-the-art regarding the implementation of such algorithms over CPUs, GPUs, and FPGAs (note that this is the first survey paper to present an implementation on FPGAs).
- A wide variety of applications, in fields such as, in the context of distributed coverage, security, medical imaging, and Virtual/Augmented Reality.

B. OUTLINE OF THE PAPER

The paper is structured as follows: Section II provides some preliminary concepts and definitions; Section III reviews the different algorithmic approaches to compute the Delaunay triangulation; Section IV is divided into three sub-sections discussing the implementation on CPUs, GPUs, and FPGAs, respectively. In Section V, the paper discusses the main applications of the Delaunay triangulation. Finally, the last section presents the conclusions of the paper, as well as prospect challenges.

II. PRELIMINARY CONCEPTS AND DEFINITIONS

A. NOTATION AND DEFINITIONS

We denote vectors by boldface lowercase letters and matrices with uppercase letters. We refer to the (i, j)-th entry of a matrix A by A_{ij}. We represent by $\mathbf{0}_n$ and $\mathbf{1}_n$ vectors with n entries, all equal to zero and to one, respectively. We use $\| \cdot \|$ to denote the Euclidean norm. Let $G = \{V, E\}$ be a graph with n nodes $V = \{v_1, v_2, \ldots, v_n\}$ and e edges $E \subseteq V \times V$, where $(v_i, v_j) \in E$ captures the existence of a link from node v_i to node v_j. A graph is said to be undirected if the existence of an edge $(v_i, v_j) \in E$ implies the presence of $(v_j, v_i) \in E$, while it is said to be directed otherwise. In this paper, we consider undirected graphs. In fact, in the context of Delaunay Triangulations, graphs are used as a convenient framework for representing adjacent points in the triangulation. In this view, an edge only models the connection of two points; hence, no orientation is required [41]. An undirected graph is connected if each node can be reached by each other node via the edges.

Let the neighbourhood $\mathcal{N}_i$ of a node v_i be the set of nodes v_j such that $(v_j, v_i) \in E$. The degree d_i of a node v_i is the number of its incident edges, i.e., $d_i = |\mathcal{N}_i|$.

B. CPU, GPU AND FPGA

The three primary forms of hardware that can be used to implement algorithms are CPUs, GPUs, and FPGAs. Each form of hardware has its own advantages and disadvantages when it comes to using algorithms. Today, the most popular type of processor used in computers is the CPU. Such a processor can be used for a wide range of tasks and is designed to carry out general-purpose instructions. Typically, CPUs are proficient at swiftly and effectively executing sequential instructions. However, parallel processing tasks that typically occur in machine learning or image processing problems might be not well-suited for them.

On the other hand, GPUs are specialised computers designed for graphically demanding tasks such as gaming and 3D rendering. GPUs are the best choice for activities such as machine learning and image processing, since they feature hundreds or thousands of parallel computing cores. GPUs can quickly access massive datasets stored in memory, thanks to their high memory bandwidth. Despite their outstanding performance on graphics-intensive applications, GPUs lack support for several software libraries and have a limited set of instructions, making them unsuitable for general-purpose computing operations like web browsing or word processing.

FPGAs are programmable devices that can be configured to perform specific functions. FPGAs can be programmed with custom logic circuits that enable them to swiftly and effectively carry out complex computations, in contrast to CPUs and GPUs that have preset instruction sets. In addition, FPGAs typically consume less power than other processors, making them perfect for embedded applications where power efficiency is crucial. However, using FPGAs for general-purpose computing activities is more challenging than resorting to CPUs or GPUs, since programming them requires specialised skills and equipment.

In conclusion, each type of processor is characterised by both benefits and drawbacks. CPUs are efficient at quickly executing sequential instructions but are not suitable for parallel processing workloads. GPUs are ideal for graphics-intensive applications and parallel processing tasks but lack support for specific software libraries and are not suitable for general-purpose computing. FPGAs are highly customisable and energy-efficient, but using them for general-purpose computing activities is more challenging than using CPUs or GPUs since programming them requires specialised skills and equipment.

C. THE DELAUNAY TRIANGULATION CONCEPT

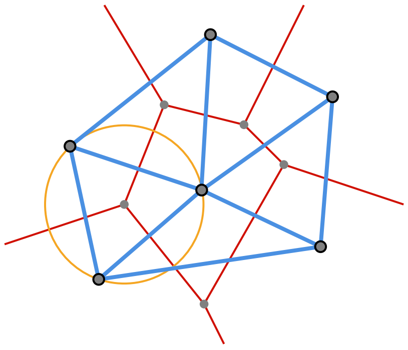


FIGURE 2. Delaunay triangulation with the circumcircles of each triangle and their centres.

Let a set of points $\mathcal{P} \subset \mathbb{R}^d$ be given (e.g., the gray dots with a black boundary in Fig. 2). A *triangulation* $\mathcal{T}(\mathcal{P})$ is a partitioning of the points into *simplices* (triangles in 2D, tetrahedra in 3D, and so on), such that no two simplices overlap and every point in the set is a vertex of at least one simplex [42] (e.g., the blue triangles in Fig. 2). In particular, the Delaunay triangulation $\mathcal{DT}(\mathcal{P})$ is a triangulation that satisfies the *Delaunay Condition*, i.e., such that no point in $\mathcal{P}$ is inside the circum-hypersphere of any simplex in $\mathcal{DT}(\mathcal{P})$ [2], [43] (e.g., in 2-D, no point is inside the circumcircle of any triangle in $\mathcal{DT}(\mathcal{P})$, an example of such circles is shown in yellow in Fig. 2). Interestingly, as stated in [2], if a set of points $\mathcal{P} \subset \mathbb{R}^d$ satisfies the condition that the affine hull of $\mathcal{P}$ is d-dimensional and no set of $d+2$ points in $\mathcal{P}$ lie on the boundary of a ball whose interior does not intersect $\mathcal{P}$, then the Delaunay triangulation for $\mathcal{P}$ is unique. Another important property is that each Delaunay triangulation is strongly related to convex hulls. In fact, the Delaunay triangulation of $\mathcal{P} \subset \mathbb{R}^d$ is the projection of the downward-facing faces of the convex hull of the set of points belonging to a paraboloid living in $\mathbb{R}^{d+1}$; this property can be leveraged upon in order to construct the Delaunay triangulation by first computing the convex hull in $\mathbb{R}^{d+1}$ and then projecting the result in $\mathbb{R}^d$.

The Delaunay triangulation has been shown to have several interesting properties in terms of optimality. For instance, it has been proved to minimise the maximum radius of the hyperspheres containing the simplices, and the weighted sum of squares of the edge lengths, where the weight is proportional to the sum of volumes of the triangles incident on the edge [44], [45]. Moreover, in $\mathbb{R}^2$ it has been

TABLE 2. Comparison of the state-of-the-art algorithms.

Algorithm	Computational Complexity	Higher Dimensions	Pros	Cons
Brute-force	$O(\vert\mathcal{P}\vert^4)$	No	Simple to implement	Exponentially slower for large data sets
Flip	$O(\vert\mathcal{P}\vert^2)$	Yes	Can optimise quality of triangulation, works well for small data sets	Slow for large data sets, may not terminate in higher-dimensions.
Incremental	$O(\vert\mathcal{P}\vert^2)$	Yes	Easy to implement, fast for small data sets, can be updated as new points are added to the dataset.	Slow and memory-intensive for large data sets, may suffer numerical stability issues when points are nearly collinear or cocircular.
Divide and conquer	$O(\vert\mathcal{P}\vert \log(\vert\mathcal{P}\vert))$	Yes	Efficient for large data sets, well-suited for parallelisation. Complexity can be lowered to $O(\vert\mathcal{P}\vert \log(\log(\mathcal{P})))$ for a large class of distributions that includes the uniform distribution in the unit square.	More complex to implement, slower than incremental for small data sets, requires additional storage space to store the subsets of points and the splitting lines.
DeWall	$O(\vert\mathcal{P}\vert^2)$ (experimental, on average), $O(\vert\mathcal{P}\vert^{\lceil \frac{d}{2} \rceil + 1})$ worst case	Yes	Works in any dimension.	Can be computationally expensive.
Sweep-Hull	$O(\vert\mathcal{P}\vert \log(\vert\mathcal{P}\vert))$	No	Fast, efficient, simple to implement	Requires sorting and is limited to planar triangulations
Fortune	$O(\vert\mathcal{P}\vert \log(\vert\mathcal{P}\vert))$	No	Can handle non-uniform point distributions; handles input with duplicate points well.	Requires sorting of the input points.
Jump flood	$O(\vert\mathcal{P}\vert \log(\vert\mathcal{P}\vert))$	Yes	Efficient and scalable, especially for large point sets; simple to implement and parallelise; works in any number of dimensions	Requires a predetermined jump distance; may produce slightly different results due to floating-point error

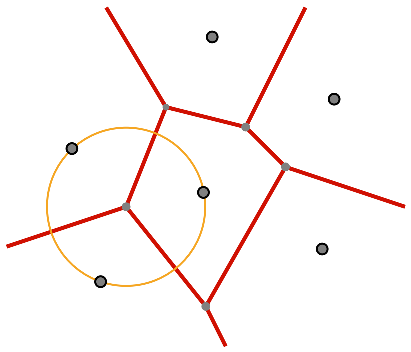


FIGURE 3. The centres of the circumcircles in Fig. 2 determine the vertices of a convex polygon, which is the Voronoi diagram.

shown to be the triangulation that maximises the minimum angle of all the triangles [46]. In particular, this property guarantees to avoid triangles with one or two highly acute angles (silver triangles[1]) during interpolation or rasterisation processes [48]. It is also important to note that connecting the centres of the circumcircles of the Delaunay triangulation produce the *Voronoi diagram* (see the red diagrams in Figs. 2 and 3), i.e., a dual structure that amounts to the partition of the space $\mathbb{R}^d$ in regions such that each point in a region is closest to one of the points in $\mathcal{P}$ [49].

[1]A silver triangle is a triangle whose area is so thin that its interior does not contain a distinct span for each scan line. In other words, instead of each scan line having a beginning and an ending pixel, each of which defines one side of the triangle, each scan line has only one pixel that may be the beginning or ending pixel [47], the convex hull of $\mathcal{P}$ is the union of the simplices of the triangulation.

III. ALGORITHMS

This section provides an overview of the different algorithmic approaches developed in the literature for computing the Delaunay triangulation. In particular, we discuss algorithms based on triangulation (subsec. III-A) and methods that rely on Voronoi tessellation (subsec. III-B). For each algorithm, we discuss the main technical aspects, we provide a pseudocode and a critical discussion. The key aspects of the different methods are compared in Table 2.

A. DELAUNAY TRIANGULATION METHODS

Delaunay triangulation methods can be classified into the following main algorithms: Brute Force [50], Flip [15], Incremental [14], Divide and conquer [14], and Sweep-hull [16].

These algorithms will be discussed in turn in the sequel. There is also the possibility of having combinations of the above algorithms, offering improved performance with respect to the initial algorithms. Some combinations will be discussed later.

1) BRUTE-FORCE METHOD

The brute force algorithm for Delaunay triangulation is a simple approach to finding the Delaunay triangulation of a set of points in a two-dimensional plane. It works by checking all possible combinations of three points to see if they form a valid Delaunay triangle. The algorithm begins by selecting a point from the dataset and forming all possible combinations of three other points. For each combination of three points, the algorithm checks if the points form a valid Delaunay triangle. If so, the triangle is added to the triangulation. This process is repeated for every point in the dataset, resulting in a complete Delaunay triangulation of the dataset. The computational complexity of the brute force algorithm for Delaunay triangulation is $O(|\mathcal{P}|^4)$; this is because, for each point in the dataset, the algorithm needs to check all possible combinations of three points, resulting in a total of $|\mathcal{P}|^4/6$ computations.

One advantage of the brute force algorithm is its simplicity and ease of implementation. It also works well for small datasets with low dimensionality. However, the algorithm is computationally expensive and impractical for large datasets or high-dimensional data. To optimise the triangulation, we can perform list fusion, prune as soon as possible, and remove adjoining lines at the end. These steps can help streamline the algorithm and improve its performance. This variant of the Delaunay triangulation algorithm is cited from the git project [50].

Algorithm 1 Brute Force Algorithm

```
1:  procedure BruteForceDelaunayTriangulation($\mathcal{P}$)
2:   Initialise empty set $\mathcal{DT}(\mathcal{P})$ of Delaunay triangles
3:  for each point $p_i$ in $\mathcal{P}$ do
4:      for each pair of points $p_j, p_k$ in $\mathcal{P}$ do
5:          if the circle through $p_i$, $p_j$, and $p_k$ contains no other points in $\mathcal{P}$ then
6:              Add the triangle $p_i, p_j, p_k$ to $\mathcal{DT}(\mathcal{P})$
7:          end if
8:      end for
9:  end for
10: return $\mathcal{DT}(\mathcal{P})$
11: end procedure
```

The pseudocode of the algorithm is given in Algorithm 1.

2) FLIP ALGORITHM

The main idea of the flip algorithm [51] is summarised in Figure 4. Consider a quadrilateral in R^2, and assume the four points have been associated with two triangles with a common edge, as in Figure. 4.a. In particular, it can be shown that if the sum of the angles α and γ is larger than or equal to $\pi/2$, then the two triangles fail to satisfy the Delaunay condition (see Figure. 4.b), meaning that one of the points is contained in the circumcircle of the remaining ones [15]. In this case, by replacing the common edge with the one joining the other two endpoints, the result is still

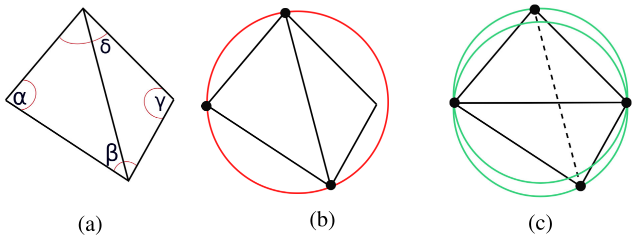


FIGURE 4. Flip algorithm in a nutshell: (a) quadrilateral decomposed in two triangles with a common edge and such that the sum of the angles α and γ is larger than $\pi/2$, (b) the two triangles do not fulfil the Delaunay condition, i.e., one of the points lies within the circumcircle of the others, (c) after flipping the common edge the two triangles satisfy the Delaunay condition.

a triangulation and satisfies the Delaunay condition [52] (i.e., the fact that the circle passing through the vertices of a triangle contains no other vertex, see Figure. 4.c); this procedure is commonly referred to as "edge flip" or Lawson flip. In this view, the flip algorithm starts by building any triangulation and then iteratively inspects pairs of triangles, eventually performing edge flips when the condition that no point lies within the circumcircle of the triangles is not met. Notably, in $\mathbb{R}^2$, this algorithm is guaranteed to converge in $O(|\mathcal{P}|^2)$ edge flips in the worst case [53]. Notice that any triangulation with $|\mathcal{P}|$ points has at least $\lfloor \frac{n-4}{2} \rfloor$ edges that can be flipped [54]. Although the flipping procedure can be extended to higher dimensional spaces [42], applying the overall algorithm to $\mathbb{R}^d$ with $d \geq 3$ is not straightforward, as the algorithm might get stuck before reaching a Delaunay triangulation [53]. Algorithm 2 summarises the procedure.

Algorithm 2 Flip Algorithm (in $\mathcal{R}^2$)

```
1: procedure Flip($\mathcal{P}$)
2:  Build any initial triangulation $\mathcal{T}(\mathcal{P})$
3: while $\mathcal{T}(\mathcal{P})$ not Delaunay do
4:     Choose quadrilateral with $\alpha + \gamma \geq \pi/2$
5:     flip common edge
6: end while
7: return $\mathcal{T}(\mathcal{P})$
8: end procedure
```

The flip algorithm is a simple and efficient method for computing the Delaunay triangulation. One of its main advantages is that it can be applied to higher dimensions, even though there might be instances that fail to converge. Moreover, the flip algorithm can handle non-uniform point distributions without needing to modify the algorithm or add any extra steps. However, the flip algorithm can get stuck in an infinite loop if it encounters a bad input or if the triangulation has a "Bowtie" structure, which occurs when four points lie on the same circle. Additionally, the flip algorithm can produce a triangulation with a higher number of flips than other algorithms, making it less efficient in terms of computational cost. Nevertheless, the flip algorithm is still a popular method for computing the Delaunay triangulation due to its simplicity and versatility.

3) INCREMENTAL ALGORITHM

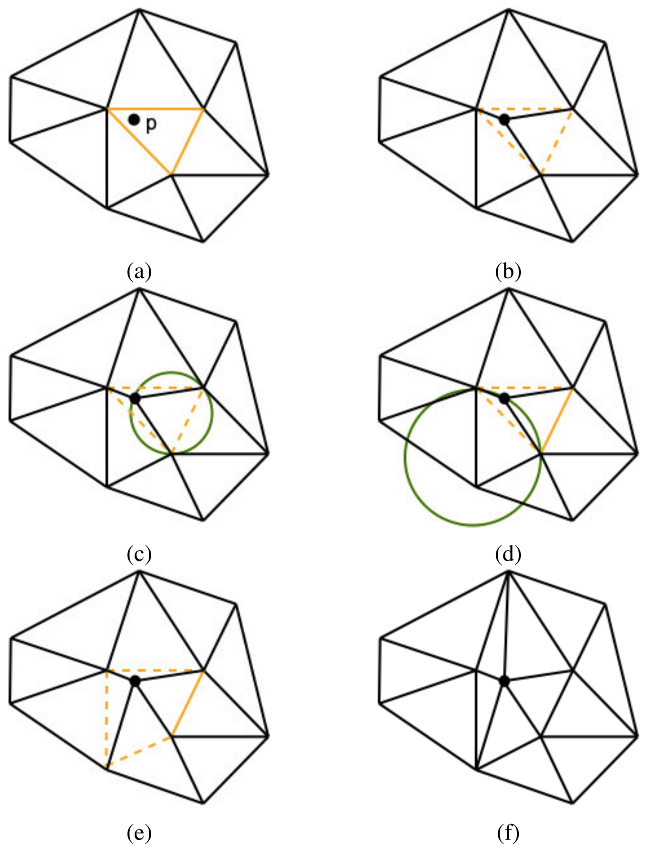


FIGURE 5. **Incremental Delaunay triangulation.**

The *incremental search* or *incremental construction algorithm* [14], [55] (see also the Bowyer-Watson algorithm [56], [57], which is similar) has the main goal of constructing a Delaunay triangulation in $\mathbb{R}^2$ of a set of points by inserting points one at a time and updating the existing triangulation to maintain the Delaunay property. To start the incremental Delaunay triangulation algorithm, a triangle is formed that is big enough to cover all the given points.[2] Then, the algorithm proceeds to include the remaining points into the triangulation, one after another, while preserving the Delaunay property throughout the process. To add a new point p, the algorithm first checks which existing triangle in the triangulation contains p. Then, its three vertices are connected to p, and the three edges of the triangle are inspected. For each such edge, the algorithm checks if they need to be flipped (i.e., replaced by the other diagonal of the surrounding quadrilateral) in order to maintain the Delaunay property. If an edge is flipped, then the other two edges that form a triangle with it become candidates for inspection. This process is repeated until all affected triangles have been updated.

Figure 5 shows an example of this procedure. In particular, in in Fig. 5a a new point p is added to an existing triangulation. Then, the algorithm splits the triangle that contains p into three triangles (as in Fig. 5b). Once these triangles have been identified, the algorithm runs a check to determine whether any of their edges need to be flipped (Fig. 5c-5e). Finally, every newly formed triangle that fails to satisfy the Delaunay Condition is flipped [58] (i.e., the triangles featuring a dashed yellow edge in Fig. 5f).

The pseudocode for the incremental algorithm is reported in Algorithm 3.

Algorithm 3 Incremental Algorithm (in $\mathbb{R}^2$)

```
 1: procedure IncrementalDT($\mathcal{P}$)
 2:     Create initial triangle containing all points in $\mathcal{P}$
 3: for all $p \in \mathcal{P}$ do
 4:     Find triangle containing $p$
 5:     Add edges of the triangle to set $E$
 6:     Add edges from $p$ to the three vertices of the triangle
 7:     while $E$ is not empty do
 8:         choose $e \in E$
 9:         Consider triangle obtained from $e$ and $p$
10:         if triangle is not Delaunay then
11:             Flip edge $e$ and remove $e$ from $E$
12:             put edges $e'$ forming a triangle with $p$ in $E$
13:         end if
14:     end while
15: end for
16: return Delaunay triangulation of $\mathcal{P}$
17: end procedure
```

The overall runtime of the incremental algorithm is $O(|\mathcal{P}|^2)$, as in the worst case $O(|\mathcal{P}|)$ edges might need to be flipped with each new point. However, if vertices are inserted in random order, then on average, only $O(1)$ flips will be performed with each insertion [59].

The incremental algorithm has several advantages. Firstly, it is a simple and intuitive algorithm that can handle large datasets with good efficiency. Secondly, it allows for incremental updates, meaning that the triangulation can be updated as new points are added to the dataset. Additionally, the algorithm guarantees the Delaunay property at each step of the triangulation process. However, one of the main disadvantages of the incremental algorithm is its worst-case time complexity of $O(|\mathcal{P}|^2)$, which can make it impractical for very large datasets. Moreover, the algorithm may also suffer from numerical stability issues when points are nearly collinear or cocircular, leading to the creation of poorly-shaped triangles. Interestingly, the algorithm was generalised[3] to three and higher dimensions in [60].

[2]Since the original points lie in a bounded region in $\mathbb{R}^2$, such an enclosing triangle always exists.

[3]Although presenting a divide-and-conquer approach, in [60] a "simplex wall" is constructed by resorting to a generalisation of the insertion algorithm to higher dimension spaces.

4) DIVIDE AND CONQUER APPROACHES IN $\mathbb{R}^2$ AND $\mathbb{R}^3$

The divide-and-conquer algorithm for Delaunay triangulation is an efficient approach for solving problems by breaking them down into smaller subproblems [14], [61], [62]. Based on this general idea, different approaches have been developed in the literature to address the bi-dimensional or three-dimensional cases. In $\mathcal{R}^2$, the main idea is to recursively separate the points into groups via a line, until each group contains three or less points. Every time the set of points is partitioned in two, the algorithm is recursively executed on each of the two subsets. Then, once the sets contain at most three points, the Delaunay triangulation is simply computed, and the results for the two sets are recursively merged along the previously identified splitting lines. This algorithm can be divided into three steps:

1) Dividing the frame into parts: The algorithm first divides the set of points $\mathcal{P}$ into smaller subsets. One way to achieve this is to use the median of the x-coordinates of the points as the splitting line. The points on one side of the line form one subset, while the points on the other side form another subset. This process is then recursively applied to each subset until a subset only contains three or fewer points.
2) Computing the triangulations: The Delaunay triangulation is computed for each subset of points. This can be done using any method such as the incremental algorithm or the flip algorithm.
3) Merging the results: Finally, the computed triangulations for each subset are merged along the previously drawn splitting lines.

The most important aspect of this method is how to merge two triangulations. Next, we describe the approach in [14], assuming the points are deployed in a bi-dimensional space; Fig. 6 provides an example of this procedure.

As a first step, the points in the left and right triangulations to be merged are sorted by increasing the x-coordinate (in the case of ties, the y-coordinate can be used). Let us refer to the Delaunay edges in the left and right triangulations as L-L edges and R-R edges, respectively. The merge operation requires the insertion of some edges between sites belonging to different triangulations (i.e., L-R or cross edges) and, consequently, the deletion of some L-L and/or R-R edges. Notice that, as demonstrated in [14], the merging procedure cannot add new L-L or R-R edges. Let us consider the line parallel to the y-axis that separates the left and right subsets of sites; notably, with respect to this line, the set of all possible L-R edges can be ordered (i.e., by increasing the value of the y-coordinate of the point that crosses the line). Based on this ordering, the merging algorithm in [14] selects cross edges incrementally, in ascending y-order.

At the beginning, the algorithm selects the lowermost L-R edge in the ordering and evaluates the next candidate (called basel). The algorithm considers the circle that passes through the three endpoints that define the current L-R edge and the basel; then, the circle continuously rises and changes size in order to maintain the basel as one of its chords. Interestingly, the centre of the circle is constrained to lie on the bisector of the basel. The circle will be point-free for a while, but unless basel is the last L-R edge, at some point, the circumference of the transforming circle will meet a new site, belonging either to L or R. The resulting triangle (i.e., the endpoints of the basel and of the newly met site) will be point-free. At this point, unless the edge thus found is the last one in the ordering, the algorithm will continue by considering an expanding circle having the new L-R edge as the basel, until a new point is found. As they are identified, the L-R edges thus selected are added to the overall triangulation. Upon addition of any of such edges, previous edges in the left and right triangulation that fail to meet the Delaunay condition are removed.

The overall time complexity of the divide-and-conquer algorithm is $O(|\mathcal{P}|\log(\mathcal{P}))$, which is faster than the incremental algorithm's time complexity of $O(|\mathcal{P}|^2)$. However, this algorithm requires additional storage space to store the subsets of points and the splitting lines. Notably, in [63] an extension is provided with a guaranteed $O(|\mathcal{P}|\log(\log(\mathcal{P})))$ complexity for a large class of distributions that includes the uniform distribution in the unit square.

Algorithm 4 reports the pseudocode for the divide-and-conquer method.

Algorithm 4 Divide-and-Conquer Algorithm for Computing the Delaunay Triangulation

```
1:  function Divide-and-conquer($\mathcal{P}$)
2:      if $|\mathcal{P}| \leq 3$ then
3:          return Compute $\mathcal{DT}(\mathcal{P})$ using any method
4:      end if
5:      Compute $m$ median of the $x$-coordinates of $\mathcal{P}$
6:      $\mathcal{P}_L \subseteq \mathcal{P}$ with $x$-coordinate $\leq m$
7:      $\mathcal{P}_R \subseteq \mathcal{P}$ with $x$-coordinate $> m$
8:      $\mathcal{DT}_L(\mathcal{P}_L) =$ Divide-and-conquer($\mathcal{P}_L$)
9:      $\mathcal{DT}_R(\mathcal{P}_R) =$ Divide-and-conquer($\mathcal{P}_R$)
10:     Merge $\mathcal{DT}_L(\mathcal{P}_L)$ and $\mathcal{DT}_R(\mathcal{P}_R)$
11:     return the merged Delaunay triangulation
12: end function
```

Notice that the above divide-and-conquer algorithm cannot be applied in $\mathbb{R}^d$ for $d > 2$. However, in the literature, divide-and-conquer approaches for the three-dimensional case have been developed, e.g., the *Delaunay Wall* (DeWall) algorithm [64]. The algorithm iteratively subdivides the points in two subsets via a hyperplane, then it constructs a part of the triangulation, namely *simplex wall*, that involves the simplices that intersect the hyperplane. Iterating the procedure over the subset new simplices are added to the triangulation. Overall, in $\mathbb{R}^3$, the DeWall algorithm has an average complexity that has been experimentally assessed to be $O(|\mathcal{P}|^2)$, although there may be degenerate instances where the complexity rises to $O(|\mathcal{P}|^3)$; in general, the theoretical worst-case complexity in $\mathbb{R}^d$ is $O(|\mathcal{P}|^{\lceil \frac{d}{2} \rceil + 1})$ [64].

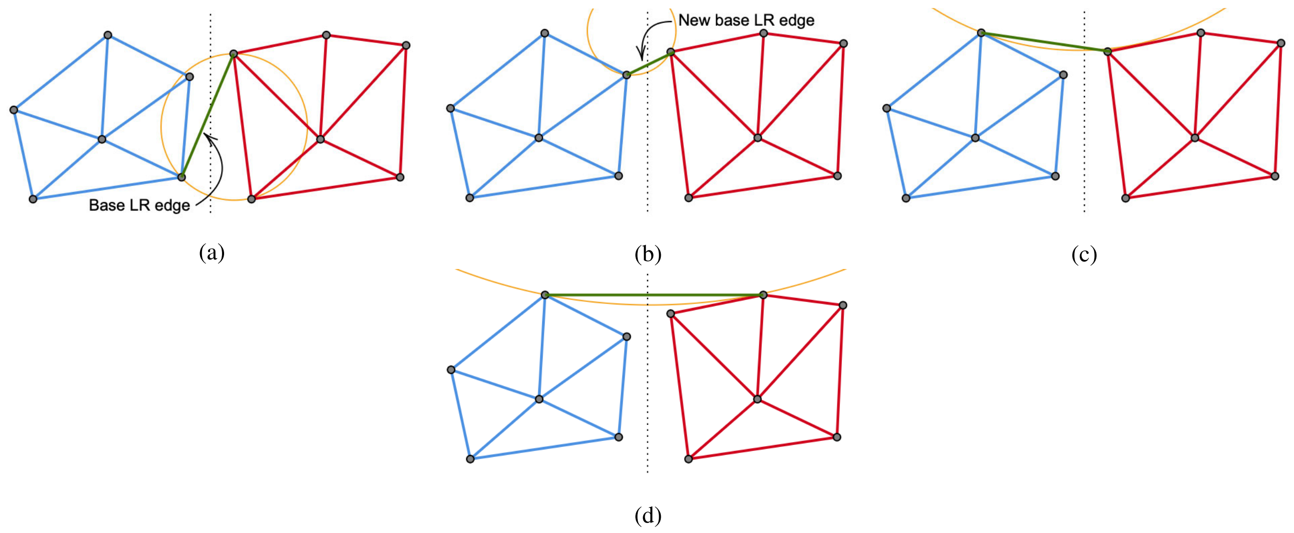


FIGURE 6. Example of rising bubble in divide and conquer algorithm.

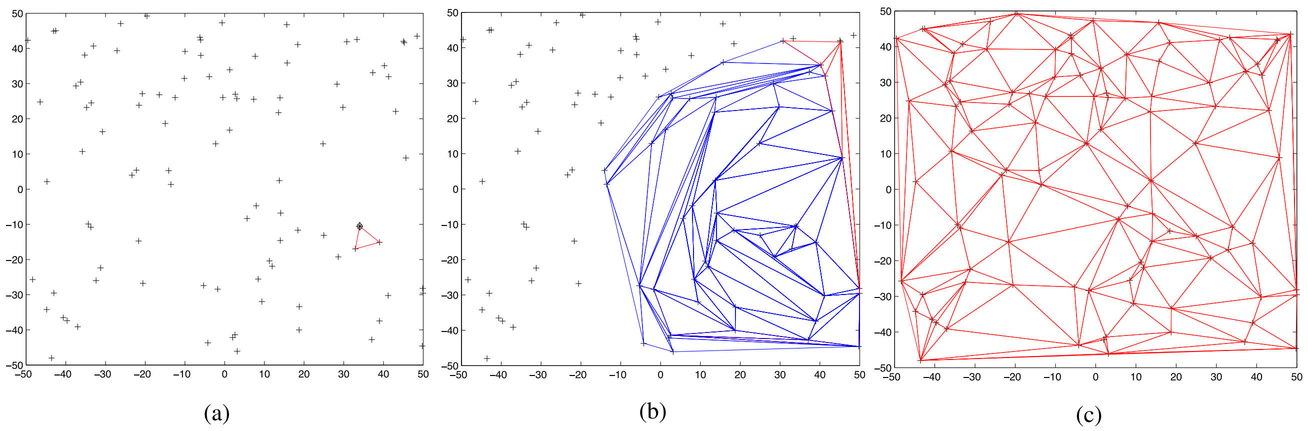


FIGURE 7. Example of the sweep-hull algorithm [16]: Panel 7a shows the initial seed; Panel 7b shows the evolution of the convex hull; Panel 7c shows the resulting Delaunay triangulation.

5) SWEEP-HULL

The Sweep-Hull algorithm [16] is a technique for efficiently computing the Delaunay triangulation in $\mathbb{R}^2$. This algorithm combines a sweeping technique with a flipping algorithm to generate a convex hull and subsequently, the Delaunay triangulation. First, the algorithm sorts the points p_i in the dataset $\mathcal{P}$ by their x-coordinates $p_{i,x}$ in ascending order with respect to a point $p_{0,x}$, i.e., according to $|p_{i,x} - p_{0,x}|^2$. The algorithm then sweeps a line from left to right across the points, creating a convex hull around the points encountered so far. The point that creates the smallest circumcircle with p_0 and p_i is identified as p_ℓ, and its centre c_ℓ is recorded. Points p_0, p_i, and c_ℓ are then ordered to form a right-handed system, which serves as the initial seed convex hull.

Next, the remaining points are sorted by their distance from c_ℓ and sequentially added to the convex hull seeded with the initial triangle. As each new point is added, the visible facets of the convex hull form new triangles. This process continues until all points have been processed and the convex hull is complete. Once the convex hull has been constructed, the Sweep-Hull algorithm uses it to construct the Delaunay triangulation by connecting each point on the convex hull to its nearest neighbours on either side, creating triangles between them. This process continues until all points have been connected, resulting in a complete Delaunay triangulation of the point set. The algorithm is illustrated in Fig. 7.

The time complexity of the Sweep-Hull algorithm is $O(|\mathcal{P}|\log(|\mathcal{P}|))$, and thus scales well with the size of the input dataset [16]. Moreover, the memory complexity of the Sweep-Hull algorithm depends on the implementation. In the worst case, where all the input points lie on the convex hull, the algorithm needs to store $O(|\mathcal{P}|)$ points. However, in practice, the number of points on the convex hull is usually much smaller than $|\mathcal{P}|$. The memory usage also depends on the choice of data structures used in the implementation, such as the priority queue and the data structures for storing the convex hull and the Delaunay triangulation.

It is also robust and can handle input datasets with duplicate points or collinear points [65].

Algorithm 5 Sweep-Hull Algorithm for Computing the Delaunay Triangulation (in $\mathcal{R}^2$)

```
1:  procedure SweepHull($\mathcal{P}$)
2:   Choose $p_0 \in \mathcal{P}$
3:   Sort $\mathcal{P}$ by $x$-coordinate in ascending order according to $|p_i - p_0|^2$
4:   Initialise a stack $S$ and add $p_0, p_1, p_2$ to $S$ in clockwise order to form the initial seed convex hull (Fig. 7)
5:  for $i = 3$ to $n$ do
6:      Let $p_i$ be the next point in $\mathcal{P}$ sorted order
7:      while the angle formed by the last two points on $S$ and $p_i$ is not convex do
8:          Pop the top point $p$ from $S$
9:          Add the triangle formed by $p$, the last point on $S$, and $p_i$ to the Delaunay triangulation
10:     end while
11:     Push $p_i$ onto $S$
12: end for
13: return Delaunay triangulation constructed from the convex hull in $\mathcal{P}$
14: end procedure
```

The pseudocode of the Sweep-Hull algorithm is given in Algorithm 5; note that the pseudocode assumes that the input points $\mathcal{P}$ have already been preprocessed, such as removing duplicates and ensuring that no three points are collinear.

B. VORONOI BASED METHODS

The Voronoi diagram is a powerful tool for analysing and visualising data. It is a mathematical construct that divides a space into regions based on the distance from a set of points (an example is given in Fig. 3).

Let $\mathcal{P}$ be a set of points, and $\{p, q, r\}$ be three non-collinear points in $\mathcal{P}$ that do not have any other points inside the circle $\mathcal{C}$ passing through p, q, and r. The centre of $\mathcal{C}$ is a Voronoi node of the Voronoi diagram $\mathcal{V}(\mathcal{P})$ of $\mathcal{P}$.

Note that $\mathcal{V}(\mathcal{P})$ is the dual graph of the Delaunay triangulation $\mathcal{DT}(\mathcal{P})$ of $\mathcal{P}$ (in this view, the triangle formed by $\{p, q, r\}$ corresponds to the Voronoi node). This means that every Voronoi node belongs to a Delaunay triangle (and vice versa). Voronoi diagrams are also the dual of Delaunay triangulations and can be generated from Delaunay triangulation and vice versa [14], [49]. The remainder of this subsection is devoted to reviewing such approaches.

1) FORTUNE'S ALGORITHM

Fortune's algorithm is based on a *sweep line* and a *beach line* that move through the plane as the algorithm progresses [67]. More in detail, the sweep line is a vertical line that moves through the plane from left to right, while the beach line is the set of points that are equidistant to the sweep line and the points in the Voronoi diagram that have already been constructed.

The input to the algorithm is a set of points in the plane, as shown in Fig. 8. The algorithm starts by sorting the points by their x-coordinate and introducing a sweep line that moves from left to right through the plane. As the sweep line arrives at each point, it creates a beach line that grows as the sweep line moves forward.

Notably, the beach line is composed of pieces of parabolas. As the sweep line continues to move, the points at which two parabolas cross, i.e., the vertices of the beach line, become the points where two or more edges of the Voronoi diagram are adjacent. In this way, the Voronoi diagram is constructed incrementally. Once the Voronoi diagram is constructed, it can be easily converted into a Delaunay triangulation [68], [69].

In particular, two types of events can occur during the algorithm's execution, *site events* and *circle events*. Site events occur when a new point is added to the set of points being processed by the algorithm. When a site event occurs, the algorithm needs to update the structure representing the beach line (i.e., the set of parabolic arcs defined by the Voronoi diagram). Specifically, the algorithm needs to insert a new parabolic arc into the beach line to represent the new site and adjust the intersection points between neighbouring arcs as needed. Circle events occur when the sweep line passes over a point where three or more sites' Voronoi regions intersect. At this point, the parabolic arcs representing the Voronoi regions around those sites all meet at a common point, and this point is a Voronoi vertex. The circle event is the point in time when the sweep line reaches the lowest point of the circle that passes through the three sites that define the Voronoi vertex. When a circle event occurs, the algorithm needs to remove the three parabolic arcs that meet at the Voronoi vertex from the beach line and replace them with a new arc representing the edge of the Voronoi diagram that connects the two adjacent sites that are not involved in the circle event. The algorithm also needs to check for new circle events that might be triggered by the removal and insertion of new edges. In summary, site events correspond to the addition of new points to the Voronoi diagram, while circle events correspond to the intersection of three or more points' Voronoi regions, which results in the creation of new Voronoi nodes and edges. The algorithm handles each type of event differently, but both types are necessary to fully construct the Voronoi diagram.

Algorithm 6 contains the pseudocode of Fortune's algorithm (while site and circle events are handled as reported in Algorithms 7 and 8, respectively).

Fortune's algorithm has a computational complexity of $O(|\mathcal{P}| \log(|\mathcal{P}|))$, making it an efficient algorithm for planar point sets. However, it is not efficient for point sets in higher dimensions.

2) JUMP FLOOD

The Jump Flood method is an effective approximated approach for computing Voronoi diagrams [70], [71]. In the beginning, the algorithm creates an $M \times M$ grid such that

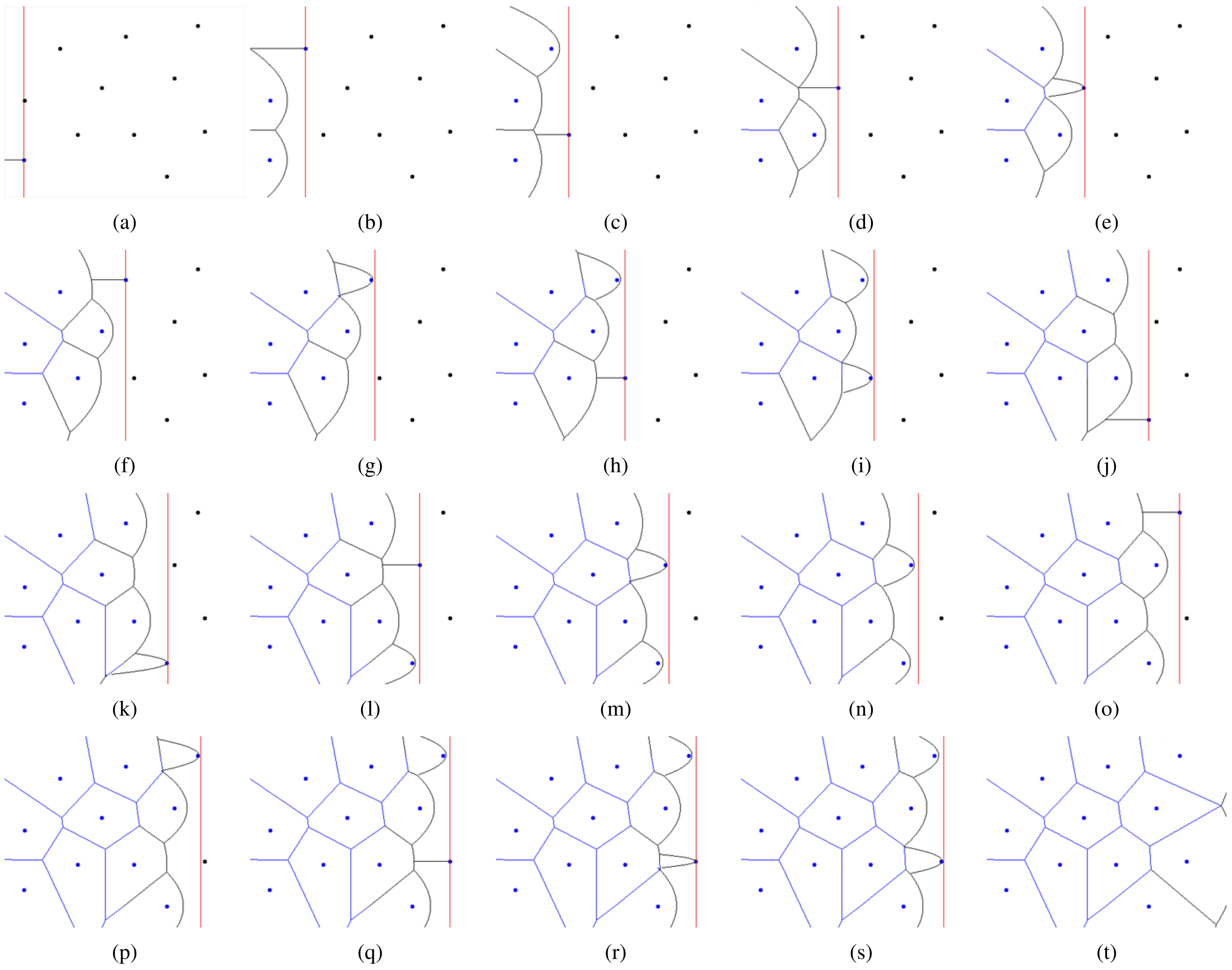


FIGURE 8. **Example of usage of Fortune's algorithm. Adapted from Wikimedia commons [66].**

each cell only contains one point $p \in \mathcal{P}$ (M is assumed to be a power of two); then, each point $p \in \mathcal{P}$ is declared to be the *seed* of the cell it belongs to. At this point the algorithm iterates a series of rounds each characterised by a parameter r, which is initialised to $M/2$ and is halved at each iteration, stopping when $r < 1$. During each round, all cells c and all its "neighbour" cells q at distance r are evaluated. If c has no seed and the seed of q is s, then s is declared to be the seed of c. If c has a seed s and q a seed s', and if p is closer to s' than to s, then the seed of q becomes s'.

The pseudocode of the algorithm is given in Algorithm 9, while an example with $|\mathcal{P}| = 6$ and $M = 128$ is given in Fig. 9.

The computational complexity of the Jump Flood method for Delaunay triangulation is $O(|\mathcal{P}| \log(|\mathcal{P}|))$. This complexity is due to the sorting operation used to compute the neighbouring points in each round.

One advantage of the Jump Flood method is that it is easy to implement and can handle non-uniform point distributions. Additionally, the algorithm is memory-efficient as it only stores the coordinates of the vertices and a small amount of metadata. On the other hand, one potential issue with the Jump Flood algorithm is that the use of discrete grid-based representations can introduce quantisation errors, particularly when the grid resolution is not fine enough to accurately represent the input point set. These errors can result in a loss of precision in the computed Delaunay triangulation and potentially affect its quality. However, this issue can be mitigated by carefully selecting the grid resolution and employing appropriate error analysis and correction techniques.

The Jump Flood algorithm can be extended to higher dimensions than 2 by essentially following the same procedure as in 2D (e.g., see [70] and references therein). In 3D, for example, the algorithm works by dividing the space into a grid of cubes of a certain size and then performing the flooding process within each cube. The jump distance is defined as the length of the cube edge, and the process

Algorithm 6 Fortune's Algorithm

```
1:  procedure Fortune($\mathcal{P}$)
2:  Create empty priority queue $\mathcal{Q}$
3:  for all $p_i \in \mathcal{P}$ do
4:      Event($p_i$) $\leftarrow$ Create new site event at $p_i$
5:      Insert Event($p_i$) into $\mathcal{Q}$
6:  end for
7:  Create empty beach line $\mathcal{B}$
8:  while $\mathcal{Q}$ is not empty do
9:      $e \leftarrow$ Extract minimum event from $\mathcal{Q}$
10:     if $e$ is a site event then
11:         HandleSiteEvent($e$, $\mathcal{B}$)
12:     else
13:         HandleCircleEvent($e$, $\mathcal{B}$, $\mathcal{Q}$)
14:     end if
15: end while
16: return Delaunay triangulation of $\mathcal{P}$
17: end procedure
```

Algorithm 7 HandleSiteEvent(e, $\mathcal{B}$)

```
1: Insert new edge $\alpha$ associated with site $e$ into $\mathcal{B}$
2: $\alpha_{\text{left}} \leftarrow$ edge to the left of $\alpha$
3: $\alpha_{\text{right}} \leftarrow$ edge to the right of $\alpha$
4: if $\alpha_{\text{left}}$ and $\alpha_{\text{right}}$ converge at a point $p$ then
5:     Create new circle event $c$ at $p$
6:     Associate $c$ with $\alpha_{\text{left}}$ and $\alpha_{\text{right}}$
7:     Insert $c$ into $\mathcal{Q}$
8: end if
```

Algorithm 8 HandleCircleEvent(q, $\mathcal{Q}$, $\mathcal{T}$, $\mathcal{B}$)

```
1:  let $p$ be the point associated with the event
2:  let $n_1$ be the node in $\mathcal{B}$ directly above $p$
3:  let $n_2$ be the node in $\mathcal{B}$ directly below $p$
4:  let $n_3$ be the node in $\mathcal{B}$ directly below $n_1$
5:  let $e_1$ be the edge associated with $n_1$ and $n_3$
6:  let $e_2$ be the edge associated with $n_2$ and $n_3$
7:  if $e_1$ and $e_2$ intersect at point $v$ then
8:      create new edge $e$ from $p$ to $v$
9:      delete $e_1$ and $e_2$ from $\mathcal{T}$
10:     add new edges $e$ and $e_1$ to $\mathcal{T}$
11:     if $p$ is below the segment of $e_1$ in $\mathcal{B}$ then
12:         add new node $n$ to $\mathcal{B}$ with $p$ as associated point
13:         set $e_1$ and $e$ as edges associated with $n$
14:     else
15:         add new node $n$ to $\mathcal{B}$ with $p$ as associated point
16:         set $e$ and $e_2$ as edges associated with $n$
17:     end if
18:     check circle events for $n_1$, $n_2$, $n$ in $\mathcal{Q}$
19: end if
```

of updating the closest vertices is done in a similar way as in 2D, but considering the 3D coordinates of the vertices. In general, the algorithm can be applied in any number of dimensions by dividing the space into a grid of hypercubes

Algorithm 9 Jump Flood

```
1:  procedure JumpFlood($\mathcal{P}$)
2:   Create $M \times M$ grid with $M$ power of 2
3:  for all cells $c$ in grid do
4:      if $c$ contains $p \in \mathcal{P}$ then
5:          seed($c$)= $p$
6:      else
7:          seed($c$)= $\emptyset$
8:      end if
9:  end for
10: $r = \frac{M}{2}$
11: while $r \geq 1$ do
12:     for all cells $c$ in grid do
13:         for all cells $q$ with distance $r$ from $c$ do
14:             $s$ =seed($c$);
15:             $s'$ =seed($q$);
16:             if $s = \emptyset$ and $s' \neq \emptyset$ then
17:                 seed($c$)= $s'$
18:             end if
19:             if $s \neq \emptyset$, $s' \neq \emptyset$ and $d(c,s) > d(c,s')$ then
20:                 seed($c$)= $s'$
21:             end if
22:         end for
23:     end for
24:     $k = k/2$
25: end while
26:  Construct $\mathcal{DT}(\mathcal{P})$ from seed information
27: return $\mathcal{DT}(\mathcal{P})$
```

and performing the flooding process within each hypercube. The jump distance is defined as the length of the hypercube edge, and the process of updating the closest vertices is done by considering the coordinates of the vertices in the appropriate dimension.

IV. IMPLEMENTATIONS ON DIFFERENT HARDWARE PLATFORMS

Central Processing Units (CPUs), Graphic Processing Units (GPUs), and Field-Programmable Gate Arrays (FPGAs) are all types of silicon mediums used for different purposes in computing. CPUs are the most common type of silicon medium and are designed for general-purpose computing. GPUs (Graphics Processing Units) are specialised hardware components that were originally designed for handling graphical computations but have evolved to become highly parallelised and efficient computing engines for a wide range of applications beyond graphics. GPUs have a high number of processing cores, typically in the hundreds or even thousands, that can work in parallel to execute multiple tasks simultaneously. This allows GPUs to perform complex computations quickly and efficiently. FPGAs are programmable chips that can be tailored to specific tasks. They are designed to be highly configurable, with logic blocks that can be programmed to perform different operations. This

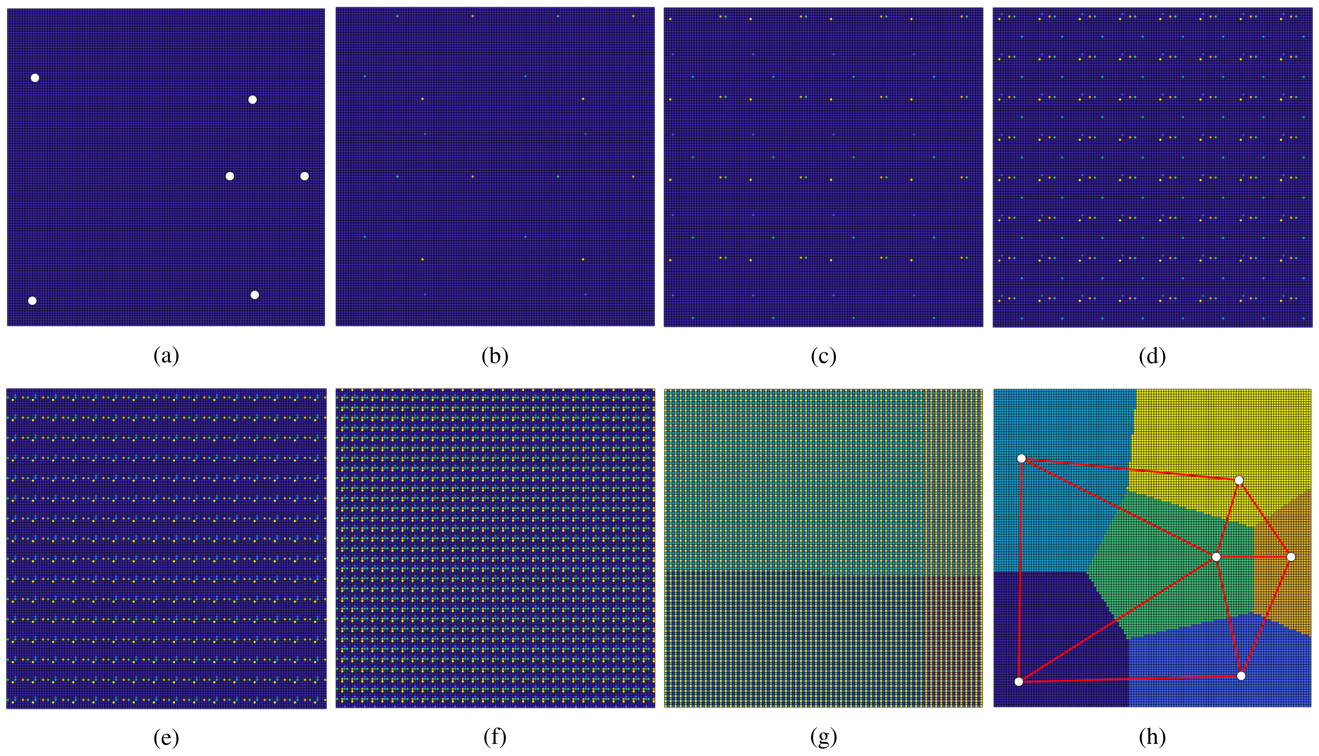


FIGURE 9. Illustration of the Jump Flood algorithm for $|\mathcal{P}| = 6$ points and $M = 128$. Panels 9a-9h show the different iterations of the method (the cells' colours correspond to their currently associated seed). The nodes are shown by white circles in Panels 9a and 9h, while the resulting Delaunay triangulation is shown in Panel 9h via red segments.

flexibility allows developers to tailor the FPGA to a specific workload, which can result in higher performance than a GPU. In addition, FPGAs can be optimised for power consumption, making them more energy-efficient than GPUs for certain tasks. However, it's important to note that FPGAs are not always the best choice for all workloads. They can be more difficult to program than GPUs, requiring specialised knowledge and expertise in hardware design and hardware description languages like Verilog and VHDL. In addition, FPGAs can be more expensive than GPUs, which can be a barrier to adoption for some applications. Overall, FPGAs can provide higher performance than GPUs for specific workloads because they are more flexible and customisable. However, the decision to use an FPGA over a GPU should be based on the specific requirements of the application, including the complexity of the workload, power constraints, and cost considerations. Each type of silicon medium has its own strengths and weaknesses, and the best option depends on the specific requirements of the application. In brief, the main differences are listed below:

- CPUs are versatile and cost-effective but are less powerful than GPUs and FPGAs in terms of raw computing power.
- GPUs are more powerful than CPUs, but they are also more expensive and consume more power.
- Unlike CPUs and GPUs, which are fixed-function devices optimised for general-purpose computing or graphics processing respectively, FPGAs can be reconfigured to perform specific functions. This means that an FPGA can be optimised for a specific workload, allowing it to perform that workload more efficiently than a GPU, which is not tailored for that specific function.
- Additionally, FPGAs have a more parallel architecture than GPUs, which can result in lower latency and higher performance for certain applications. FPGAs can also perform more operations per clock cycle than CPUs and GPUs, which also contributes to higher performance.

The remainder of this section is dedicated on understanding the merits by each type of silicon medium for the specific application of Delaunay Triangulations. More specifically, it focuses on comparing the different implementations of the aforementioned algorithms used for computing the Delaunay triangulation on CPUs, GPUs, and FPGAs.

A. IMPLEMENTATION ON CPU

Over time, several CPU implementations have been developed. In this section, we will review some of the most notable CPU implementations.

1) STUDY OF PARALLEL DELAUNAY TRIANGULATION USING MANY-CORE PROCESSOR IN 3D

To improve the performance of Voronoi diagram partition in [18], the researchers implemented and optimised Delaunay triangulation on the Xeon Phi SE7110p processor with 61 cores, each able to execute four threads, and with a frequency of 1.091 GHz. Specifically, the authors developed a parallel implementation of the flipping algorithm (e.g., see Algorithm 2 in Section III-A2). In particular, the algorithm develops parallel implementations of the insertion and flipping subroutines. In particular, the parallel approach in [18] aims to insert multiple points at once and then flipping can improve the quality of the mesh.

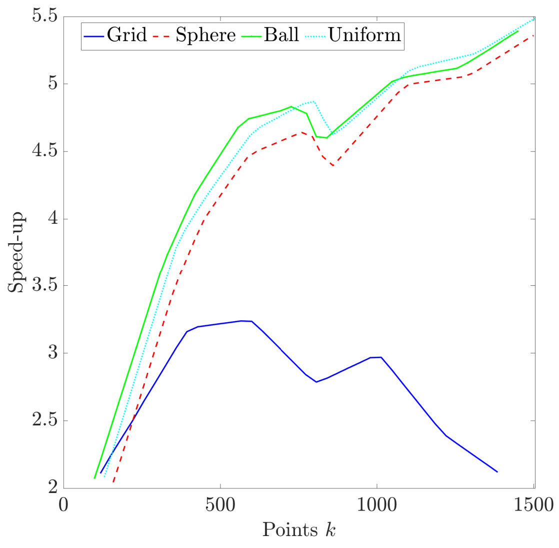


FIGURE 10. Parallel vs CGAL speed-up comparison, considering three-dimensional data points (Source data: [18]).

Figure 10 shows the speed-up with respect to CGAL for large-scale problems with a large number of points sampled in 3D from different distributions, such as the Uniform, Grid, Ball, Sphere, and Gaussian ones.[4] According to the figure, the parallel algorithm showed significant performance acceleration, i.e., it exhibited a speed-up that was always above a factor of two and, except for the Grid distribution, it reached a factor of about 5.5 for a large number of points.

2) SCALING UP TO A BILLION POINTS

In [72] the authors demonstrate how a parallel implementation of the incremental algorithm is able to handle up to one billion points, even when there is not enough RAM to handle all the points at the same time. In particular, the authors demonstrate this over a machine with an Intel, CoreTM, i7 CPU, 870@2.93 GHz with 16 GB RAM, and show that their algorithm exhibits a linear complexity in the number of points. In particular, triangulations with one billion points in 3D only require about 2000 s for uniformly distributed points and 25000 s for non-uniformly distributed points.

[4]The uniform distribution considered points sampled uniformly at random from a cube in 3D. The Gaussian one was such that the points were sampled in [0, 1] via a Gaussian function. The Grid distribution was sampled uniformly at random from the range [0, 1024]. The Ball distribution was a point set evenly distributed within a sphere with a radius of 0.5. The Sphere distribution encompassed evenly distributed points on the surface of a sphere with a thickness of 0.05.

3) DISTRIBUTED AND PARALLEL DELAUNAY TRIANGULATION ON CLUSTER/CLOUD IN 2D

Generating unstructured meshes for extremely large point sets is still a major challenge for scientists working with large-scale or high-resolution data sets. To tackle this problem, several hybrid algorithms that combine parallel and incremental approaches have been developed. These algorithms have been implemented on clusters, as described in [19], or on the cloud, as demonstrated in [17] and [20]. In the latter work, the authors compared the performance of a hybrid cloud-based algorithm with that of a sequential implementation on a single machine.

In particular, the algorithm in [17] considers points in 2D and combines the divide and conquer approach and the incremental method. Specifically, if the number of points of the problem at hand is below a threshold, then triangulation is directly computed via the incremental algorithm. Conversely, if the problem features more points than the threshold, it is broken down in two subproblems with approximately equal size, and each problem is solved recursively.

In [17] a comparison is provided for the hybrid algorithm in two different environments: a sequential implementation on a single machine (shown in Fig. 11a) and a distributed D-TIN service deployed on the GeoKSCloud platform (shown in Fig. 11b). The experiments were conducted using various problem sizes, with a threshold ranging from $|\mathcal{P}|/8$ to $|\mathcal{P}|/4$. The figures also report theoretical times that were estimated by the authors via a cost model (see [17] for details). According to the figure, in the single machine case, the theoretical and experimental execution times are quite in accordance (the discrepancies are due to system activities in the background). As for the cloud case, it can be noted that the time required for communication is not negligible, but in any case, the latter implementation allows a relevant time reduction (e.g., a speedup of about 3.5 times is observed).

B. IMPLEMENTATION ON GPU

CPUs are obviously the first choice for implementing algorithms and testing them; however, when big data is considered or when there are real-time constraints, approaches based on CPUs might not be sufficient. In this view, in the literature several approaches have been adopted to take advantage of the large computational capabilities of GPUs.

1) CGAL LIBRARY

In 1996, eight European research institutions teamed up to create the Computational Geometry Algorithms Library (CGAL) [21], an open-source software library of computational geometry algorithms. Figure 12 shows the results of using CGAL's Delaunay triangulation in 3D (using a 3D version of the Bowyer-Watson algorithm [56], [57]) on a control data-set. Specifically, the figure reports the computational time (red stars) over a machine with an Intel i7 26000K 3.4 GHz processor, 16GB DDR3 RAM, and an NVIDIA GTX 580 Fermi graphics card with 3GB of

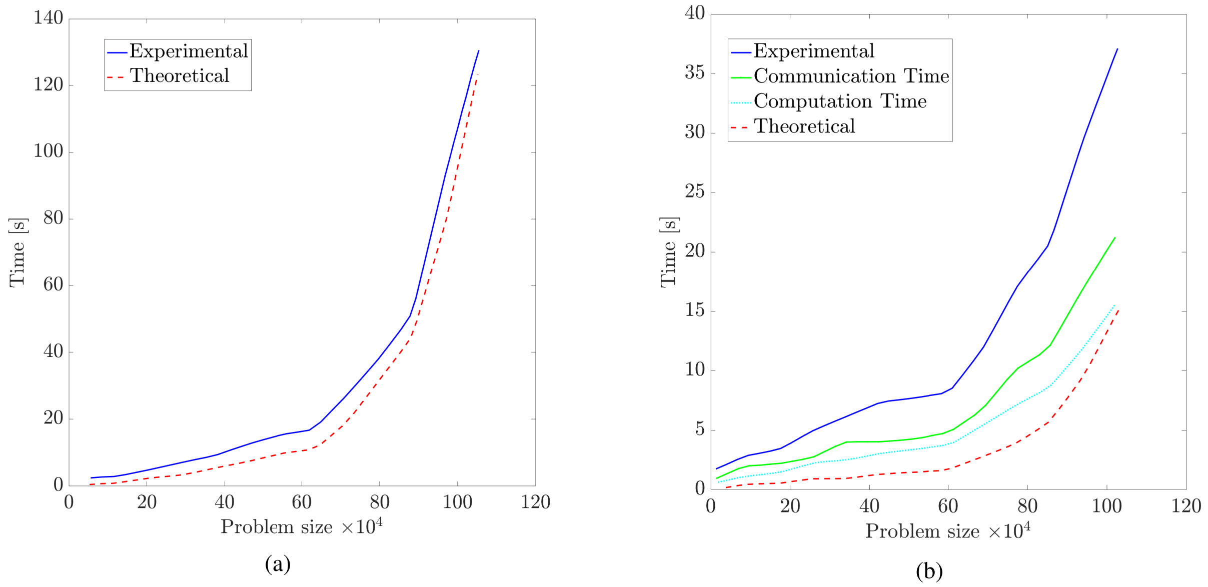


FIGURE 11. Comparison of the execution time of the hybrid algorithm developed in [17] over: (a) a single machine, or (b) a parallel cloud. For each curve, a theoretical computational time is also provided. Panel (b) also explicitly breaks down the overall experimental time into computation and communication times (data source: [17]).

video memory, considering datasets with different sizes; for each dataset, the average over ten runs is reported [36]. Notably, such numerical results (and datasets) are often regarded as a standard for comparison with newer results in several papers. For comparison, we report a linear and a nonlinear fitting of the data in Figure 12, with a blue dashed line and a black dotted line, respectively. In particular, the linear fitting amounts to $\alpha|\mathcal{P}|$, where we estimated (using MATLAB's `fitlm` function) $\alpha \approx 1.20 \times 10^{-5}$, while the nonlinear fitting is $\beta|\mathcal{P}|\log(|\mathcal{P}|)$, where we estimated (using MATLAB's `lsqcurvefit` function) $\beta = 8.93 \times 10^{-7}$. Interestingly, the linear fitting appears to be more accurate than the nonlinear one, suggesting that the library exhibits (at least experimentally) a complexity that is linear in the number of points.

2) DELAUNAY TRIANGULATION IN 3D ON THE GPU

In [36], among other approaches, the author presents the so-called gFlip3D algorithm. The algorithm is based on parallel point insertion and flipping and is implemented on GPUs, based on the CUDA parallel computing platform.

Also in this case, experiments are undertaken on a personal computer with an Intel i7 26000K 3.4 GHz processor, 16GB DDR3 RAM, and an NVIDIA GTX 580 Fermi graphics card with 3GB of video memory [36].

Figure 13 shows the running time of the gFlip3d algorithm while considering synthetic random data sampled from different distributions, while Table 3 shows the results for real datasets with different sizes.

According to Figure 13, the algorithm exhibits a linear convergence rate and all distributions yield comparable results, except for the grid distribution which has the worst performance. Notably, for synthetic data, the speedup of gFlip3D over CGAL was estimated to be above a factor of 6.5, except for the grid distribution, which reached a maximum speedup of 3. Moreover, as shown in Table 3, for real

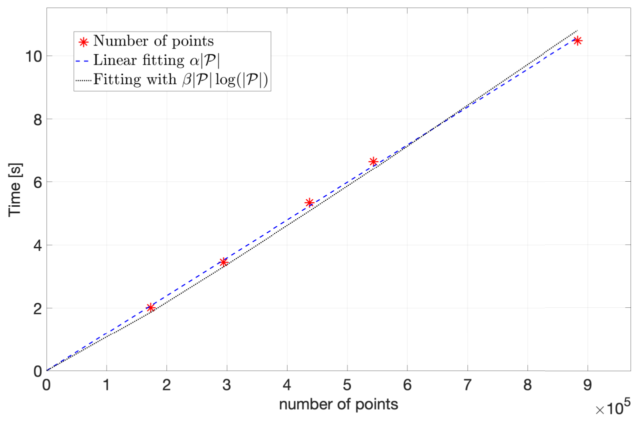


FIGURE 12. CGAL computational time over a machine with an Intel i7 2600K 3.4 GHz CPU and 16GB of DDR3 RAM, and an NVIDIA GTX 580 Fermi graphics card with 3GB of video memory, considering datasets with different sizes (data source: [36]). Results are the average over 10 runs (red stars). A linear and nonlinear fitting are reported for comparison.

TABLE 3. Computational time of the gFlip3d algorithm, speedup with respect to CGAL, and fraction of non-flippable facets for real datasets (Source data: [36]).

# points	CGAL [s]	gFlip3d [s]	Speedup	Non-flippable facet ratio
172974	2.01	0.87	2.16	0.00050
294012	3.45	0.90	3.87	0.00040
437645	5.335	1.75	3.2	0.00082
543652	6.65	2.19	3.02	0.00035
882954	10.47	5.44	1.91	0.00062

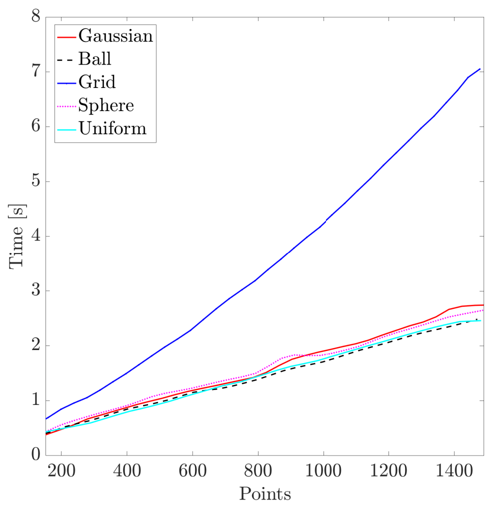


FIGURE 13. Running time of the gFlip3d algorithm, plotted against the number of points while considering synthetic data sampled from different distributions (Source data: [36]).

data, the gFlip3D algorithm showed a speedup ranging from 1.91 to 3.87.

However, a drawback of this approach is that the result does not always guarantee that all facets satisfy the Delaunay condition. In particular, the experimental results in [36] showed that the fraction of such facets over the total was less than 0.0001 for synthetic data, while in the case of real data, it ranged from 0.00035 to 0.00082.

3) 3D DELAUNAY TRIANGULATION BASED ON A GPU ACCELERATED ALGORITHM

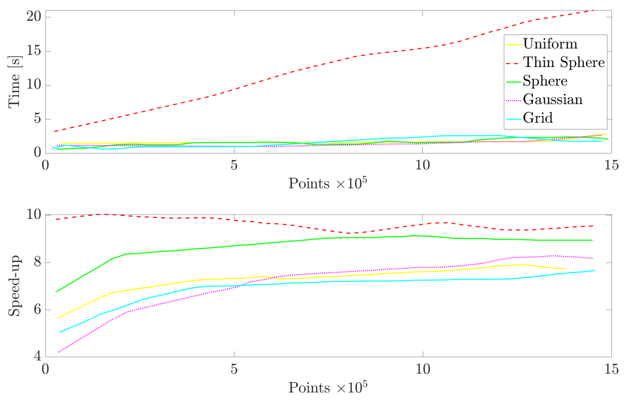


FIGURE 14. Accelerated DT running time - speedup/CGAL (source data: [22]).

In [22] a GPU implementation of the Delaunay triangulation is developed, where points are inserted in parallel and then a procedure called splaying is used to identify edges that locally violate the Delaunay condition; such edges are then flipped in order to obtain a valid triangulation.

A version of this algorithm using the CUDA programming model was implemented on a personal computer with an Intel i7 26000K 3.4 GHz processor, 16GB DDR3 RAM, and an NVIDIA GTX 580 Fermi graphics card with 3GB of video memory. The performance of the algorithm implemented using the CUDA programming model was evaluated using input sizes ranging from 10^5 to 15×10^5. Fig. 14 shows the running time and speedup of the algorithm compared to CGAL, considering random points obtained from different distributions.[5] Interestingly, the algorithm achieved a speedup of about 4-6 times for smaller instances and quickly increased to about 8-10 times as the number of points increased.

To complement the analysis, in [22] the algorithm was tested with respect to real-world datasets with a number of points ranging from about 180.000 up to 3.600.000. Notably, also in this case, the algorithm exhibited a speedup of a factor ranging from 6.1 to 9.2, i.e., the algorithm exhibited comparable speed-up factors for both synthetic and realistic datasets.

In [22], the authors also provide a breakdown of the time requirements for the different subroutines that compose this approach. Interestingly, except for a particular random distribution (i.e., the sphere, where splaying took about 60% of time, the most time-consuming operation was flipping (above 60% of time), while the insertion of points ranked second (taking about $20 - 30\%$ of time).

4) A FLIPPING APPROACH IN 2D AND 3D

Reference [24] presents an in-depth study of flip algorithms in low dimensions, specifically for regular triangulation and convex hull in 2D and 3D. The authors propose a series of provably correct flip algorithms that allow for non-restrictive execution order. The algorithms are implemented for both CPUs and GPUs. Experimental results show that the GPU implementation achieves significant speedup over existing single-threaded CPU implementations.

The GPU implementation follows the CUDA programming model and utilises a parallel workflow. The geometric structures are represented as arrays, and flipping is done in multiple iterations using checking and flipping kernels. To prevent simultaneous flipping of edges whose affecting regions overlap, two kernels[6] are utilised in every iteration: a checking kernel and a flipping kernel. Within the checking kernel, a single thread assesses the validity of a given candidate. Should candidate r require flipping, the responsible thread assigns the index of r to all the facets within the induced sub-complex of r. Within the flipping kernel, r undergoes flipping solely if all these facets retain the consistent labeling of the same index. The paper also describes the incremental insertion approach for constructing inputs on the GPU.

The experiments compare the CPU and GPU implementations for constructing 2D regular triangulations. The GPU implementation achieves significant speedup over

[5] In particular, within the thin sphere distribution, points lie in between the surface of two balls of slightly different radii (see [22]).

[6] A function that is meant to be executed in parallel on an attached GPU is called a kernel.

the CPU implementation, especially when the number of non-redundant points is small. The time breakdown analysis reveals that the flipping phase becomes more time-consuming as the number of non-redundant points increases.

The paper acknowledges that GPU implementations may require more memory due to the need for auxiliary buffers. The experiments demonstrate that the GPU implementation can handle a certain number of points depending on the available GPU memory and the point distribution.

It is also noted that the parallel workflow for flip algorithms does not explicitly handle load balancing, which may lead to inefficiency in cases of very non-uniform point distributions. However, such cases are rare in practice.

5) QUASI-DELAUNAY TRIANGULATIONS USING GPU-BASED EDGE-FLIPS

In [23], an iterative GPU-based algorithm for enhancing triangulations based on the Delaunay criterion is proposed. The algorithm incorporates a threshold value to handle co-circular or close to co-circular point configurations, resulting in a small fraction of triangles that do not satisfy the Delaunay condition. Comparative evaluations were performed against the Triangle software[7] [73] and the CGAL library to assess the quality of the generated triangulations. The results indicate that the algorithm produces less than 0.05% different triangles for fully random meshes and less than 1% for noise-based meshes.

The algorithm's implementation leverages GPU processing and demonstrates compatibility with OpenGL,[8] effective handling of co-circular point configurations, and avoidance of deadlocks. The behaviour of the algorithm is analysed, revealing a rapid decrease in the number of edge-flips per iteration, emphasising the significance of the initial iterations. Furthermore, the algorithm exhibits a low exclusion rate of threads, indicating the usefulness of parallelism. The number of iterations as a function of mesh size follows an asymptotic growth of $O(\log(|\mathcal{P}|))$, indicating its suitability for large-scale problems and enhanced parallelism.

The performance evaluation of the algorithm encompasses different input scenarios, including bad-quality random 2D triangulations, noise-based 2D triangulations, and popular 3D surface meshes. The experimental results demonstrate that the percentage of missed triangles in the algorithm's triangulations compared to CGAL is less than 0.1% for both random and noise-based meshes. The algorithm outperforms Lawson's $O(|\mathcal{P}|^2)$ edge-flip method on CPU by achieving speedups of up to 50 times on bad-quality random meshes and 3 times compared to the 2D $O(|\mathcal{P}|\log(|\mathcal{P}|))$ algorithms in CGAL and Triangle. Although the comparison involves GPU versus CPU implementations and quasi-Delaunay versus exact triangulations, the authors argue that the algorithm's sensitivity to input triangulation topology distinguishes it from the constructive methods employed by CGAL and Triangle. This claim is supported by noise-based tests, which demonstrate speedups of up to 36 and 27 over CGAL and Triangle, respectively, and 55 times over Lawson's method.

6) PARALLEL CONSTRAINED DELAUNAY TRIANGULATION ON THE GPU

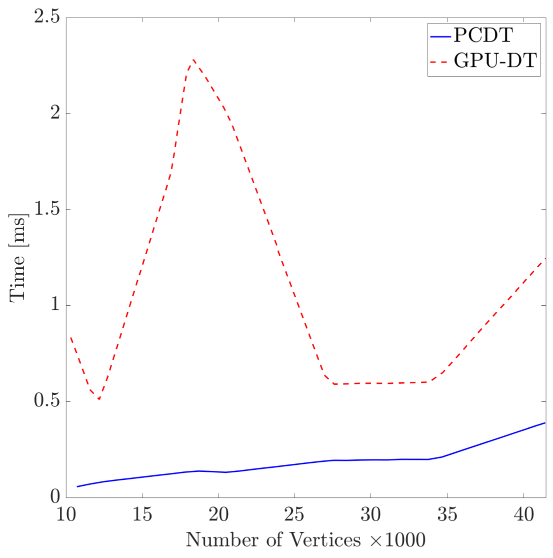


FIGURE 15. Running time versus number of vertices (Source data: [25]).

In [25], a method is presented for calculating the 2D-constrained Delaunay triangulation (CDT) of a planar straight-line graph (PSLG) composed of points and segments. This method involves the simultaneous insertion of points and segments into the triangulation while carefully handling conflicts that may arise from the concurrent insertion of points or edge flips. The implementation makes use of NVIDIA GPUs.

The approach follows an iterative process that terminates when all elements of the PSLG (points and segments) are inserted into the Delaunay triangulation (DT) without requiring any further edge flips. Each iteration consists of four steps: (1) locating the triangle containing each non-inserted point through a walking process, (2) inserting at most one point per triangle into the triangulation, (3) marking specific edges as segments or for flipping due to segment crossings or violation of local Delaunay conditions, and (4) flipping at most one marked edge per triangle.

Figure 15 presents a plot showing the running time as a function of the number of vertices. It can be observed that the proposed approach (PC-DT) outperforms other input models. Notably, the performance of GPU-DT is significantly influenced by the distribution of input points,

[7]Triangle produces precise Delaunay triangulations, constrained Delaunay triangulations, conforming Delaunay triangulations, Voronoi diagrams, and top-notch triangular meshes. The latter can be crafted devoid of acute or obtuse angles, making them apt for finite element analysis.

[8]OpenGL, known as Open Graphics Library, serves as a cross-language, cross-platform API for creating 2D and 3D vector graphics. This interface is commonly employed to interact with GPUs, enabling accelerated graphic rendering. OpenGL finds widespread use in CAD, virtual reality, scientific visualisation, and gaming.

and interestingly, the expected running time of this parallel approach exhibits linear growth relative to the input size.

C. IMPLEMENTATION ON FPGA

To the best of our knowledge, there are only a few instances in the literature where Delaunay triangulation has been implemented on an FPGA.

For instance, Gao et al. [30] and Rahnama et al. [29], have implemented Delaunay blocks on FPGA as part of more complex algorithms, but in both cases, the Delaunay triangulation itself was developed on the CPU block.

In [27], a 2D Delaunay triangulation core for surface reconstruction was implemented on a Field Programmable Gate Array (FPGA) chip using a high-level synthesis.[9] In particular, the FPGA implemented the Incremental Algorithm (see Algorithm 3).

Fig. 16 provides a comparison of the FPGA implementation against a software implementation over a machine with an Intel(R) Core(TM) i3-3220 CPU at 3.30 GHz. In particular, the figure considers a popular dataset (i.e., the "Stanford Bunny" [75]), sampled at 477, 953, 1906, and 3811 points. Notably, regardless of the size of the dataset, the FPGA implementation exhibited an overall improvement of about 19 times with respect to the CPU implementation. In particular, Fig. 16a shows the execution time of the software algorithm versus the FPGA core implementation, and Fig. 16b shows the corresponding speedup.

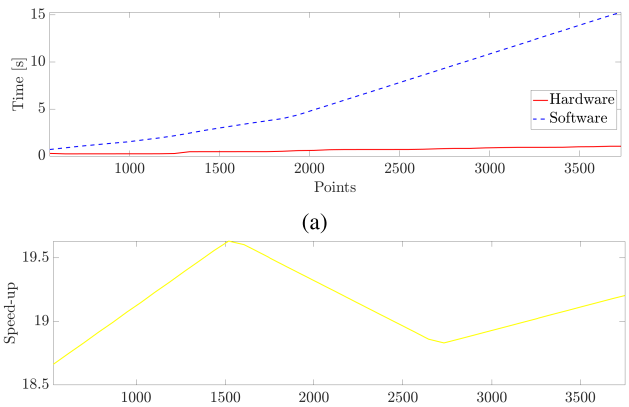


FIGURE 16. Comparison between the FPGA implementation (labelled as HW for hardware) and a software implementation of the incremental algorithm (labelled as SW) for Delaunay triangulation, considering 2D examples with 477, 953, 1906, and 3811 points. Panel (a) shows the execution time, while panel (b) shows the speedup of the FPGA implementation with respect to the software implementation (Source data: [27]).

Finally, although not based on Delaunay, it is worth mentioning the FPGA implementation of a triangulation function in [26], where the authors propose an architecture that employs linear triangulation with an inhomogeneous solution to the equation system. Specifically, [26] involved a comparison between an FPGA implementation and a software counterpart utilising MATLAB, C++, and OpenCV. Notably, the FPGA implementation was able to process about 572 times more points per second with respect to the C++ software implementation and about 2122 times more than the MATLAB one. This supports the conclusion that FPGA implementation, although challenging, has the potential to greatly outperform software implementations.

[9]High-level synthesis is the process of transforming a high abstraction level design description into a register transfer level (RTL) description for input into conventional ASIC and FPGA implementation workflows (e.g., see [74]).

D. DISCUSSION

The performance of CPU, GPU, and FPGA implementations of Delaunay triangulation differs in various aspects. While CPU implementation may be slower overall, it excels in the sorting part of the algorithm, making it the most effective for this particular task. On the other hand, GPU implementation is well-suited for parallel processing, enabling it to handle large amounts of data quickly. Its ability to perform multiple calculations simultaneously makes it highly suitable for tasks that require high-speed processing.

FPGA implementation, although less discussed in existing literature, offers significant advantages. It provides a high level of customisation and allows for both parallelisation and concurrency, making it a powerful alternative to CPU and GPU implementations. Moreover, FPGAs often demonstrate superior power efficiency compared to CPUs or GPUs, which holds great promise for achieving impressive results and further advancements in the future.

V. DELAUNAY TRIANGULATION APPLICATIONS

Practical applications of Delaunay triangulation include a plethora of cases. In what follows, we provide several examples to highlight its practical nature.

A. LLOYD'S ALGORITHM

The first application considered is the use of Delaunay triangulation in an algorithm with many applications; namely, Lloyd's algorithm [76]. It is an iterative optimisation method (which corresponds to a gradient descent algorithm [77]) used for finding evenly spaced sets of points in subsets of Euclidean spaces and partitions of these subsets into well-shaped and uniformly sized convex cells, thus improving the quality of a mesh or grid [78].

The algorithm starts by selecting an initial set of ℓ points (e.g., by using a Monte Carlo method) as the centres and then repeatedly executes the following steps:

1. computes the Voronoi diagram of the ℓ points;
2. computes the centroid of each cell of the Voronoi diagram;
3. updates the centres to be the centroids of their respective Voronoi cells.
4. If this new set of centroids meets some convergence criterion, terminate; otherwise, return to step 1.

This process is repeated until some convergence criterion is met (since, in principle, the method converges asymptotically), and then a good approximation of the Delaunay triangulation has been obtained. Mathematically, the algorithm can be expressed as finding a set of ℓ centres $\mathcal{C}$ that minimises the sum of squared distances between each point $\mathbf{p}_i$ in the set $\mathcal{P}$ and its assigned centre, as shown in Equation (1), i.e.,

$$L(\mathcal{P}, \mathcal{C}) := \min \sum_{i=1}^{n} \|\mathbf{p}_i - A(\mathbf{p}_i, \mathcal{C})\|^2, \qquad (1)$$

where

$$A(\mathbf{p}_i, \mathcal{C}) = \arg\min_{\mathbf{c}\in\mathcal{C}} \|\mathbf{p}_i - \mathbf{c}\| \; . \qquad (2)$$

At each iteration, the closest centre to each point is found by considering $A_t(\mathbf{p}_i, \mathcal{C}_t)$, where $\mathcal{C}_t$ is the estimate of the centroids at iteration t.

Algorithm 10 summarises the above procedure.

Algorithm 10 Lloyd's Algorithm for Delaunay Triangulation

```
1:  procedure Lloyd($\mathcal{P}$, $\ell$)
2:    Choose initial centres $\mathcal{C}_0 \subseteq \mathcal{P}$
3:    Set $t = 0$
4:  repeat
5:        Assign each point in $\mathcal{P}$ to the nearest centre:
              $A_t(\mathbf{p}_i, \mathcal{C}_t) = \arg\min_{\mathbf{c}\in\mathcal{C}_t} \|\mathbf{p}_i - \mathbf{c}\|; ,$   (3)
6:        Update the centers for all $j \in \{1, \ldots, \ell\}$:
              $\mathbf{c}_j^{t+1} = \frac{1}{|\{i | A_t(\mathbf{p}_i, \mathcal{C}_t) = \mathbf{c}_j\}|} \sum_{i | A_t(\mathbf{p}_i, \mathcal{C}_t) = \mathbf{c}_j} \mathbf{p}_i; ,$   (4)
7:        Increment $t \leftarrow t + 1$
8:  until Convergence is reached or a maximum number of
    iterations is exceeded
9:  return $\mathcal{DT}(\mathcal{P})$, where $\mathcal{DT}(\mathcal{P})$ is the Delaunay Triangu-
    lation of $\mathcal{P}$
10: end procedure
```

By iterating through the steps described in Algorithm 10, the point set gradually converges towards a more uniform distribution within the region.[10],[11] This can improve the quality of the Delaunay triangulation by reducing the occurrence of elongated or overly acute triangles, which is the source of numerical instability or other issues in certain applications. This algorithm was first proposed in 1957 by Stuart P. Lloyd at Bell Labs as a technique for pulse code modulation [81]. Lloyd's algorithm has been extensively used for (scalar and vector) quantisation [82] and other applications, such as smoothing of triangle meshes in the finite element method (discussed in Section V-J).

An interesting application of Lloyd's algorithm is the so-called coverage control problem, where agents belonging to a multi-agent system are required to be optimally (with respect to some coverage metric) dispersed over a designated area [83], [84]. Specifically, each agent moves towards the centroid of its cell and re-calculates its Voronoi cell based on the centroid until the whole area is covered [85], [86], [87]. Applications of distributed coverage include, among others, surveillance or search and rescue.

B. SPATIAL CLUSTERING

Spatial clustering refers to the segregation of geographic regions with distinct features into non-overlapping subsets (clusters). Such clustering can help the extraction of meaningful spatial patterns that are useful in several applications, such as, earthquake analysis [88], [89], [90], epidemics [91], traffic incident analysis [92], urban hot-spot detection [93], and climate analysis [94]. One such spatial clustering method is the adaptive spatial clustering algorithm based on Delaunay triangulation, called ASCDT [95]. ASCDT utilises statistical features of the edges of Delaunay triangulation and a unique spatial proximity definition based on Delaunay triangulation to detect spatial clusters. This algorithm has the ability to automatically identify clusters with complex shapes and non-uniform densities in a spatial database, without requiring the setting of parameters or prior knowledge. Users can also adjust parameters to suit specific applications. Additionally, ASCDT is robust to noise, making it reliable in real-world scenarios. Experiments conducted on simulated and real-world spatial databases, such as an earthquake dataset in China, showcase the effectiveness and advantages of the ASCDT algorithm.

C. BEHAVIOUR MONITORING AND CLASSIFICATION

Similar to spatial clustering, Delaunay triangulation has potential application uses in monitoring and classification. An example of its use is given in [96], in which a machine vision-based monitoring method was developed for studying pig lying behaviour. More specifically, in a study conducted at a commercial pig farm, top-view cameras were used to monitor four pens with 22 pigs each for 15 days. Delaunay triangulation (DT) was employed as an image-processing technique to analyse the lying patterns of pigs in different thermal categories relative to room setpoint temperature. Different lying patterns, such as close, normal, and far, were defined based on the perimeter of each DT triangle, and

[10]Lloyd's algorithm has a time complexity of $O(k|\mathcal{P}|)$, where k is the number of iterations required for convergence. Each iteration of the algorithm requires calculating the Voronoi diagram, which can be done in $O(|\mathcal{P}|\log(|\mathcal{P}|) + |\mathcal{P}|^{\lceil d/2 \rceil})$ time using efficient algorithms [79] (i.e., $O(|\mathcal{P}|\log(|\mathcal{P}|))$ in $\mathbb{R}^2$ and $O(|\mathcal{P}|^2)$ in $\mathbb{R}^3$). However, the centroid computation can take $O(|\mathcal{P}|)$ time, making the overall time complexity $O(k|\mathcal{P}|)$. Notice that the worst-case time complexity of Lloyd's algorithm can be superpolynomial [80] if the algorithm is performed until convergence, as the number of iterations required for convergence in the worst case can be of the order of $2^{\sqrt{|\mathcal{P}|}}$.

[11]In terms of memory complexity, the Lloyd algorithm requires storing the point cloud and the set of centres, which each require $O(d|\mathcal{P}|)$ memory, where d is the dimensionality of the points. Moreover, the Voronoi diagram computation may require additional memory to store auxiliary data structures, which can be up to $O(|\mathcal{P}|)$ in size.

the percentages of each lying pattern were calculated for each thermal category. To automatically classify the group lying behaviour of pigs into the three thermal categories, a multilayer perceptron (MLP) neural network was developed and tested. The DT features, including the mean value of perimeters, maximum and minimum length of sides of triangles, were used as inputs for the MLP classifier. The results showed that the MLP classifier could accurately classify lying features into the three thermal categories with a high overall accuracy of 95.6%. This study demonstrates the potential of using Delaunay triangulation in combination with MLP classification and mathematical modelling as a precise method for quantifying pig lying behaviour in welfare investigations.

D. FORMATION CONTROL

Distance-based formation control has been studied extensively; see, e.g., survey paper [97] and references therein. A gradient-based control scheme utilising artificial potential fields is employed to guide the agents into a distance-based formation. During the movement of agents into the formation, their relative position often changes, and as a consequence the communication links vary (some are removed and new ones are introduced), resulting to a switching communication topology. The authors in [11] apply the Delaunay triangulation as a switching communication graph. Specifically, a distributed Delaunay triangulation algorithm to maintain a proximity communication network among mobile agents proposed in [98] is deployed in order to maintain the switching Delaunay triangulation in real-time. The deployment of a distributed Delaunay algorithm comes as a natural selection in this case, since i) it has always a spanning tree, which guarantees that the Delaunay triangulation is maintained with distributed algorithms [98], and ii) the pairwise closest agents are always connected, which is a valuable property of the Delaunay triangulation concerning the collision avoidance.

E. PRIVACY AND SECURITY

Delaunay triangulation holds potential application uses in privacy and security. In what follows, we provide two examples, one for privacy [99] and one for security [100].

The objective of the privacy problem is to transfer a message using an image without changing the image itself. Thus, a potential attacker does not notice the hidden message in the picture. Specifically, in [99] a new method of steganography[12] is presented based on a combination of Catalan objects (cryptomorphic descriptions of the same thing) and Delaunay triangulation. Specifically, in the encryption process of the steganography algorithm proposed in [99], an image is encoded into a binary record, converting the hidden information into a binary record, and creating a Delaunay triangulation of a binary record of an image whose vertices are the carriers of the secret message bit. Then, by applying an encryption technique (in this case, the stack permutation method and Catalan objects) over the coordinates of the Delaunay vertex, a new encrypted triangulation emerges whose vertex coordinates are placed in a sequence.

[12]Steganography is the technique of hiding data within an ordinary, nonsecret file or message to avoid detection.

The second example uses Delaunay triangulation for fingerprint identification, as presented in [100]. The proposed approach utilises Delaunay triangulation to associate a unique topological structure with fingerprint minutiae, allowing for more meaningful minutiae grouping during indexing. This approach preserves index selectivity, reduces memory requirements without compromising recognition accuracy, and improves recognition time. Unlike other approaches that consider $O(|\mathcal{P}|^3)$ triangles, the proposed approach (assuming N minutiae per fingerprint on average) considers only $O(|\mathcal{P}|)$ minutiae triangles, resulting in significant memory savings and improved recognition time. The minutiae triangles used for indexing are particularly effective in discrimination since they satisfy the properties of Delaunay triangulation. The unique characteristics of Delaunay triangulation, such as its efficiency in computation and local resilience to noise or distortions, make it a suitable choice for fingerprint identification. Experimental results on a database of 300 fingerprints demonstrate the effectiveness of the proposed approach.

F. MODELLING AND DESCRIBING OF STOCHASTIC SYSTEMS

Poisson Voronoi diagrams (PVDs) are diagrams in which the centres are randomly and uncorrelated distributed (correspondingly, one can define the Poisson Delaunay triangulation). Its stochastic nature extends the traditional Voronoi diagram to a random point process, called a Poisson process. Analytically, the PVDs can be described using concepts from stochastic geometry and computational geometry.

Such diagrams facilitate the analysis of spatial patterns and distributions in a stochastic setting. They find applications in modelling and describing several natural/social phenomena and stochastic systems in various scientific and engineering disciplines; for example, for telecommunication networks [101], for biology [102], for evaluating the actual galaxy distribution [103], and for constructing random lattices in quantum field theory [104].

G. WIRELESS SENSORS

The trade-offs between sensing and communication coverage are crucial in the design of Wireless Sensor Networks (WSNs). Establishing a minimum bound for sensing coverage is essential in scheduling, target tracking, redeployment, and communication coverage. However, existing methods that measure coverage as a percentage value often lack detailed information, resulting in varying Quality of Coverage (QoC) for scenarios with equal coverage percentages. To address this limitation, [105] proposes a new coverage measurement method utilising Delaunay Triangulation (DT). This DT-based approach provides coverage values that are compatible with all coverage measurement tools. Furthermore, it categorises sensors as 'fat', 'healthy', or 'thin', indicating areas with dense, optimal, or scattered sensor deployments, respectively. Additionally, the proposed method can identify the largest empty area without any sensors in the field. Simulation results demonstrate that the proposed DT method achieves accurate coverage information and offers multiple tools for comparing QoC across different scenarios.

H. MEDICAL IMAGING

In the field of medical imaging, specifically in the context of developing automatic diagnostic tools for early detection of skin cancer lesions in dermoscopic images, a fast and fully automatic algorithm for skin lesion segmentation is presented in [9]. The algorithm utilises Delaunay Triangulation to extract a binary mask of the lesion region, eliminating the need for any training stage. A quantitative experimental evaluation is conducted on a publicly available database, comparing the proposed algorithm with six well-known state-of-the-art segmentation methods. The results of the experimental analysis reveal that the proposed approach exhibits high accuracy in segmenting benign lesions, while the segmentation accuracy decreases significantly when processing melanoma images. This observation prompts the consideration of geometrical and colour features extracted from the binary masks generated by the algorithm for classification, resulting in promising results for melanoma detection.

I. GEOGRAPHIC INFORMATION SYSTEMS (GIS)

In geographic information systems (GIS), the Delaunay triangulation is a commonly employed technique for creating triangulated irregular network (TIN) models, which accurately represent surface morphology in various applications. The research paper in [43] introduces a novel linear-time Convex Hull Insertion algorithm designed to construct TIN models for a given set of points, including specific features such as constraint break lines and exclusion boundaries. Empirical results obtained from experiments conducted on personal computers using diverse point datasets, ranging in size up to 50,000 points, demonstrate the efficiency of the proposed algorithm. It achieves expedited TIN model construction, operating approximately in $O(N)$ time complexity for randomly distributed points, where N represents the number of points.

J. FINITE ELEMENT ANALYSIS (FEA)

Finite element analysis (FEA) is a computer-aided method for predicting how a product behaves under various physical effects, such as mechanical stress, vibration, forces, heat, electrostatic, and fatigue. Such an analysis allows the designers to investigate whether their product will work the way it is supposed to. Finite element mesh generation is a crucial step in FEA and while there exist several methods to create a mesh, Delaunay is the most commonly used.

In [106] the application of adaptive Delaunay triangulation in finite element modelling for two-dimensional crack propagation problems is studied. It provides a comprehensive description of the proposed procedure, which combines the Delaunay triangulation algorithm with an adaptive remeshing technique. This technique generates smaller elements around crack tips and larger elements in other regions. The effectiveness of the procedure is evaluated by analysing the resulting stress intensity factors and simulated crack propagation behaviour. To assess its performance, [106] presents three sample problems: a centre cracked plate, a single edge cracked plate, and a compact tension specimen. The results of these simulations are thoroughly examined and analysed.

There exist several commercial software tools that serve as FEA solvers available in the market; most of them use Delaunay triangulation and offer customised and automated solutions for analysing various design scenarios.

K. VIRTUAL REALITY AND AUGMENTED REALITY

The paper in [107] introduces a method for constructing the Delaunay triangulation, which is essential for creating polygon patch models used in computer graphics and virtual reality (VR) applications. The method builds upon the conventional incremental approach used to construct the Voronoi diagram but incorporates two key characteristics specifically suited for three-dimensional geometric modelling.

Firstly, the method allows for the removal of points from the mesh, which is particularly useful for interactive modelling in VR, where points are frequently added, removed, or manipulated. Secondly, it enables the constraint of lines between two points to serve as mesh edges, ensuring the creation of accurate three-dimensional models. Without this feature, the resulting mesh often deviates significantly from the actual object's three-dimensional structure.

Additionally, it also provides insights into applying this method to radial range images and offers detailed explanations of the technique itself. Additionally, experimental results and evaluations of this method are presented, demonstrating its effectiveness in VR applications.

L. COSMIC STRUCTURE FORMATION

Delaunay triangulation is also the building block for a mathematical tool for reconstructing a continuous density or intensity field covering a volume from a set of discrete points, called the Delaunay tessellation field estimator (DTFE) [108], in order to determine the density or intensity of point samplings. It is based on the stochastic geometric concept of the Delaunay triangulation generated by the point set. A salient feature of DTFE is that it automatically adjusts to changes in density and shape. The fact that multi-dimensional discrete data sets are a major source of astrophysical information makes DTFE suitable for studying astrophysical applications, such as cosmic structure formation and large-scale galaxy distribution.

VI. CONCLUSION AND PROSPECT CHALLENGES

Delaunay triangulation is a critical computational geometry algorithm that has a wide range of applications, including mesh generation, image processing, and geographic information systems.

This paper provides a comprehensive overview of the Delaunay Triangulation, presenting the main algorithms, reviewing the state of the art with respect to the implementation approaches based on CPUs, GPUs, and FPGAs, and discussing the most important applications.

Although Delaunay triangulation is a well-studied problem, significant challenges still exist. Below are some of the most important ones:

1) Robustness: Delaunay triangulation algorithms can be sensitive to input data with degeneracies or numerical issues, such as floating-point roundoff errors. Ensuring the robustness of Delaunay triangulation is an ongoing challenge in computational geometry. To address this issue, various techniques can be used, including the use of exact arithmetic or adaptive precision techniques [73], [109].
2) Scalability: Computing the Delaunay triangulation can be computationally expensive and memory-intensive for large datasets. There are several techniques to address scalability, such as incremental algorithms, parallel algorithms, and hierarchical techniques [110].
3) Quality: Delaunay triangulation can produce low-quality triangles in certain cases, such as when the input data is highly non-uniform or when there are sharp features in the data. To address this issue, several techniques can be used, such as mesh optimisation or constrained Delaunay triangulation algorithms [111].
4) Dimensionality: Computing Delaunay triangulation in high-dimensional spaces presents a significant challenge, as the runtime and memory requirements grow exponentially with the dimension. To address this issue, several techniques can be used, such as randomised algorithms or approximation algorithms [112].
5) Constraints: Certain applications require specific constraints to be imposed on the Delaunay triangulation, such as avoiding particular regions or satisfying specific boundary conditions. Several techniques can address this issue, such as using constrained Delaunay triangulation algorithms or adding constraints to the optimisation objective [113].

REFERENCES

[1] G. W. Commons. *Mesh Reconstruction*. Accessed: Mar. 28, 2023. [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/d/d0/Meshreconstruction.gif
[2] B. Delaunay, "Sur la sphre vide. A la mmoire de Georges Vorono," *Bulletin de l'Acadmie des Sci. de l'URSS, Classe des Sci. Mathmatiques et na*, vol. 6, pp. 793–800, Jan. 1934.
[3] F. Bassi, S. Rebay, and M. Savini, "Transonic and supersonic inviscid computations in cascades using adaptive unstructured meshes," in *Proc. ASME Int. Gas Turbine Aeroengine Congr. Expo.*, Mar. 2015, Art. no. V001T01A095.
[4] M. Fisher, B. Springborn, A. I. Bobenko, and P. Schroder, "An algorithm for the construction of intrinsic Delaunay triangulations with applications to digital geometry processing," in *Proc. ACM SIGGRAPH Courses (SIGGRAPH)*, NY, NY, USA, 2006, pp. 69–74.
[5] N. P. Weatherill, "Delaunay triangulation in computational fluid dynamics," *Comput. Math. With Appl.*, vol. 24, pp. 129–150, Sep. 1992.
[6] P. Labatut, J.-P. Pons, and R. Keriven, "Efficient multi-view reconstruction of large-scale scenes using interest points, Delaunay triangulation and graph cuts," in *Proc. IEEE 11th Int. Conf. Comput. Vis.*, Sep. 2007, pp. 1–8.
[7] Q. Ma, Q. Zou, Y. Huang, and N. Wang, "Dynamic pedestrian trajectory forecasting with LSTM-based Delaunay triangulation," *Appl. Intell.*, vol. 52, no. 3, pp. 3018–3028, 2022.
[8] J.-D. Boissonnat and B. Geiger, "Three-dimensional reconstruction of complex shapes based on the Delaunay triangulation," *Proc. SPIE*, vol. 1905, pp. 964–975, Jul. 1993.
[9] A. Pennisi, D. D. Bloisi, D. Nardi, A. R. Giampetruzzi, C. Mondino, and A. Facchiano, "Skin lesion image segmentation using Delaunay triangulation for melanoma detection," *Computerized Med. Imag. Graph.*, vol. 52, pp. 89–103, Sep. 2016.
[10] S. G. Lee, Y. Diaz-Mercado, and M. Egerstedt, "Multirobot control using time-varying density functions," *IEEE Trans. Robot.*, vol. 31, no. 2, pp. 489–493, Apr. 2015.
[11] A. Schwab and J. Lunze, "Formation control over Delaunay triangulation networks with guaranteed collision avoidance," *IEEE Trans. Control Netw. Syst.*, vol. 10, no. 1, pp. 419–429, Mar. 2023.
[12] J.-G. Park, B. Charrow, D. Curtis, J. Battat, E. Minkov, J. Hicks, S. Teller, and J. Ledlie, "Growing an organic indoor location system," in *Proc. 8th Int. Conf. Mobile Syst., Appl., Services*, Jun. 2010, pp. 271–284.
[13] G. Oliva, S. Panzieri, F. Pascucci, and R. Setola, "Sensor networks localization: Extending trilateration via shadow edges," *IEEE Trans. Autom. Control*, vol. 60, no. 10, pp. 2752–2755, Oct. 2015.
[14] L. Guibas and J. Stolfi, "Primitives for the manipulation of general subdivisions and the computation of Voronoi," *ACM Trans. Graph.*, vol. 4, no. 2, pp. 74–123, Apr. 1985.
[15] Y. Jiang, Y.-t. Liu, and F. Zhang, "An efficient algorithm for constructing Delaunay triangulation," in *Proc. 2nd IEEE Int. Conf. Inf. Manage. Eng.*, Apr. 2010, pp. 600–663.
[16] D. A. Sinclair, "S-hull: A fast radial sweep-hull routine for Delaunay triangulation," Tech. Rep., 2016.
[17] J. Lin, R. Chen, C. Yang, Z. Shu, C. Wang, Y. Lin, and L. Wu, "Distributed and parallel Delaunay triangulation construction with balanced binary-tree model in cloud," in *Proc. 15th Int. Symp. Parallel Distrib. Comput. (ISPDC)*, Jul. 2016, pp. 107–113.
[18] Z. Jinzhu, S. Qingzeng, H. Hua, and Z. Minglu, "Study of parallel Delaunay triangulation using many-core processor," *Acta Technica*, vol. 62, no. 1B, pp. 993–1002, 2017.
[19] C. Nguyen and P. J. Rhodes, "TIPP: Parallel Delaunay triangulation for large-scale datasets," in *Proc. 30th Int. Conf. Sci. Stat. Database Manage.*, Jul. 2018, pp. 1–12.
[20] T. Su, W. Wang, H. Liu, Z. Liu, X. Li, Z. Jia, L. Zhou, Z. Song, M. Ding, and A. Cui, "An adaptive and rapid 3D Delaunay triangulation for randomly distributed point cloud data," *Vis. Comput.*, vol. 38, no. 1, pp. 197–221, Jan. 2022.
[21] *The CGAL Project, CGAL User and Reference Manual*, 5.4 ed., CGAL Editorial Board, 2022.
[22] T.-T. Cao, A. Nanjappa, M. Gao, and T.-S. Tan, "A GPU accelerated algorithm for 3D Delaunay triangulation," in *Proc. 18th Meeting ACM SIGGRAPH Symp. Interact. 3D Graph. Games*, Mar. 2014, pp. 47–54.
[23] C. Navarro, N. Hitschfeld, and E. Scheihing, "Quasi-Delaunay triangulations using GPU-based edge-flips," in *Proc. Int. Conf. Comput. Vis., Imag. Comput. Graph.*, S. Battiato, S. Coquillart, R. S. Laramee, A. Kerren, and J. Braz, Eds. Berlin, Germany: Springer, 2014, pp. 36–49.
[24] M. Gao, T.-T. Cao, and T.-S. Tan, "Flip to regular triangulation and convex hull," *IEEE Trans. Vis. Comput. Graphics*, vol. 23, no. 2, pp. 1056–1069, Feb. 2017.
[25] N. Coll and M. Guerrieri, "Parallel constrained Delaunay triangulation on the GPU," *Int. J. Geographical Inf. Sci.*, vol. 31, no. 7, pp. 1467–1484, Jul. 2017.

[26] S. Wonneberger, M. Köhler, W. Derendarz, T. Graf, and R. Ernst, "Efficient 3D triangulation in hardware for dense structure-from-motion in low-speed automotive scenarios," in *Proc. 24th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2014, pp. 1–6.

[27] C. Kallis, K. M. Deliparaschos, G. P. Moustris, A. Georgiou, and T. Charalambous, "Incremental 2D Delaunay triangulation core implementation on FPGA for surface reconstruction via high-level synthesis," in *Proc. 22nd IEEE Int. Conf. Emerg. Technol. Factory Autom. (ETFA)*, Sep. 2017, pp. 1–4.

[28] O. Rahnama, D. Frost, O. Miksik, and P. H. S. Torr, "Real-time dense stereo matching with ELAS on FPGA-accelerated embedded devices," *IEEE Robot. Autom. Lett.*, vol. 3, no. 3, pp. 2008–2015, Jul. 2018.

[29] O. Rahnama, T. Cavallari, S. Golodetz, A. Tonioni, T. Joy, L. Di Stefano, S. Walker, and P. H. S. Torr, "Real-time highly accurate dense depth on a power budget using an FPGA-CPU hybrid SoC," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 66, no. 5, pp. 773–777, May 2019.

[30] T. Gao, Z. Wan, Y. Zhang, B. Yu, Y. Zhang, S. Liu, and A. Raychowdhury, "IELAS: An ELAS-based energy-efficient accelerator for real-time stereo matching on FPGA platform," in *Proc. IEEE 3rd Int. Conf. Artif. Intell. Circuits Syst. (AICAS)*, Jun. 2021, pp. 1–4.

[31] S. Liu, Z. Wan, B. Yu, and Y. Wang, "Robotic computing on FPGAs," *Synth. Lectures Comput. Archit.*, vol. 16, no. 1, pp. 1–218, 2021.

[32] P. Maur, "Delaunay triangulation in 3D," Dept. Comput. Sci. Eng., Tech. Rep., 2002.

[33] G. Brouns, A. De Wulf, and D. Constales, "Delaunay triangulation algorithms useful for multibeam echosounding," *J. Surveying Eng.*, vol. 129, no. 2, pp. 79–84, May 2003.

[34] Ø. Hjelle and M. Dæhlen, *Triangulations and Applications*. Cham, Switzerland: Springer, 2006.

[35] S.-W. Cheng, T. K. Dey, and J. Shewchuk, *Delaunay Mesh Generation*. Boca Raton, FL, USA: CRC Press, 2012.

[36] A. Nanjappa, "Delaunay triangulation in R3 on the GPU," Ph.D. thesis, Nat. Univ. Singapore, Singapore, 2012.

[37] S. L. Gonzaga de Oliveira, "A review on Delaunay refinement techniques," in *Proc. Int. Conf. Comput. Sci. Its Appl.*, Salvador de Bahia, Brazil. Cham, Switzerland: Springer, Jun. 2012, pp. 172–187.

[38] F. Aurenhammer, R. Klein, and D.-T. Lee, *Voronoi Diagrams and Delaunay Triangulations*. Singapore: World Scientific, 2013.

[39] S. Dinas and J. M. Banon, "A review on Delaunay triangulation with application on computer vision," *Int. J. Comput. Sci. Eng*, vol. 3, pp. 9–18, 2014.

[40] S. Fortune, "Voronoi diagrams and Delaunay triangulations," in *Handbook of Discrete and Computational Geometry*. London, U.K.: Chapman & Hall, 2017, pp. 705–721.

[41] M. D. P. C. Vila, "Generalized Delaunay triangulations: Graph-theoretic properties and algorithms," Ph.D. thesis, Universitat Politècnica de Catalunya, Barcelona, Spain, 2020.

[42] J. De Loera, J. Rambau, and F. Santos, *Triangulations: Structures for Algorithms and Applications*, vol. 25. Cham, Switzerland: Springer, 2010.

[43] V. J. D. Tsai, "Delaunay triangulations in TIN creation: An overview and a linear-time algorithm," *Int. J. Geographical Inf. Syst.*, vol. 7, no. 6, pp. 501–524, Nov. 1993.

[44] V. T. Rajan, "Optimality of the Delaunay triangulation in R^d," in *Proc. 7th Annu. Symp. Comput. Geometry (SCG)*, 1991, pp. 357–363.

[45] O. R. Musin, "Properties of the Delaunay triangulation," in *Proc. 13th Annu. Symp. Comput. Geometry (SCG)*, 1997, pp. 424–426.

[46] R. Sibson, "Locally equiangular triangulations," *Comput. J.*, vol. 21, no. 3, pp. 243–245, Mar. 1978.

[47] R. Stevens, *Computer Graphics Dictionary*, 1st ed. Hingham, MA, USA: Charles River Media, Feb. 2002.

[48] H. Edelsbrunner, X.-Y. Li, G. Miller, A. Stathopoulos, D. Talmor, S.-H. Teng, A. Üngör, and N. Walkington, "Smoothing and cleaning up slivers," in *Proc. 32nd Annu. ACM Symp. Theory Comput.*, May 2000, pp. 273–277.

[49] F. Aurenhammer, R. Klein, and D.-T. Lee, *Voronoi Diagrams and Delaunay Triangulations*. Singapore: World Scientific, 2013.

[50] L. Maydwell, *Brute-Force Voronoi Diagram Algorithm*. San Francisco, CA, USA: GitHub Gist Repository, 2012.

[51] C. L. Lawson, "Software for C1 surface interpolation," in *Mathematical Software*. Amsterdam, The Netherlands: Elsevier, 1977, pp. 161–194.

[52] S. Fortune, "Numerical stability of algorithms for 2D Delaunay triangulations," in *Proc. 8th Annu. Symp. Comput. Geometry (SCG)*, NY, NY, USA, 1992, pp. 83–92.

[53] C. D. Toth, J. O'Rourke, and J. E. Goodman, *Handbook of Discrete and Computational Geometry*. Boca Raton, FL, USA: CRC Press, 2017.

[54] F. Hurtado, M. Noy, and J. Urrutia, "Flipping edges in triangulations," in *Proc. 12th Annu. Symp. Comput. Geometry (SCG)*, 1996, pp. 214–223.

[55] P. J. Green and R. Sibson, "Computing Dirichlet tessellations in the plane," *Comput. J.*, vol. 21, no. 2, pp. 168–173, May 1978.

[56] A. Bowyer, "Computing Dirichlet tessellations," *Comput. J.*, vol. 24, no. 2, pp. 162–166, Feb. 1981.

[57] D. F. Watson, "Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes," *Comput. J.*, vol. 24, no. 2, pp. 167–172, Feb. 1981.

[58] M. Varshosaz, H. Helali, and D. Shojaei, "The methods of triangulation," in *Proc. 1st Annu. Middle East Conf. Exhib. Geospatial Inf., Technol. Appl.*, Jan. 2005.

[59] L. J. Guibas, D. E. Knuth, and M. Sharir, "Randomized incremental construction of Delaunay and Voronoi diagrams," *Algorithmica*, vol. 7, nos. 1–6, pp. 381–413, Jun. 1992.

[60] P. Cignoni, C. Montani, and R. Scopigno, *A Merge-First Divide & Conquer Algorithm for Ed Triangulations*. Pisa, Italy: Istituto CNUCE-CNR, 1992.

[61] M. I. Shamos and D. Hoey, "Closest-point problems," in *Proc. 16th Annu. Symp. Found. Comput. Sci. (SFCS)*, Oct. 1975, pp. 151–162.

[62] D. T. Lee and B. J. Schachter, "Two algorithms for constructing a Delaunay triangulation," *Int. J. Comput. Inf. Sci.*, vol. 9, no. 3, pp. 219–242, Jun. 1980.

[63] R. A. Dwyer, "A faster divide-and-conquer algorithm for constructing Delaunay triangulations," *Algorithmica*, vol. 2, nos. 1–4, pp. 137–151, 1987.

[64] P. Cignoni, C. Montani, and R. Scopigno, "DeWall: A fast divide and conquer Delaunay triangulation algorithm in Ed," *Comput.-Aided Des.*, vol. 30, no. 5, pp. 333–341, Apr. 1998.

[65] M. Berg, O. Cheong, M. Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications*. Cham, Switzerland: Springer-Verlag, 2008.

[66] M. Wikimedia Commons. *Fortunes-Algorithm-Slowed*. Accessed: Mar. 28, 2023. [Online]. Available: https://en.wikipedia.org/wiki/File:Fortunes-algorithm-slowed.gif

[67] S. Fortune, "A sweepline algorithm for Voronoi diagrams," *Algorithmica*, vol. 2, nos. 1–4, p. 153, Nov. 1987.

[68] D. Dobkin and W. Thurston, "The geometry of circles: Voronoi diagrams, Moebius transformations, convex hulls, Fortune's algorithm, the cut locus, and parametrization of shapes," Princeton Univ.Princeton, NJ, USA, Tech. Rep., 1986.

[69] V. Domiter and B. Žalik, "Sweep-line algorithm for constrained Delaunay triangulation," *Int. J. Geographical Inf. Sci.*, vol. 22, no. 4, pp. 449–462, Apr. 2008.

[70] G. Rong and T.-S. Tan, "Variants of jump flooding algorithm for computing discrete Voronoi diagrams," in *Proc. 4th Int. Symp. Voronoi Diagrams Sci. Eng. (ISVD)*, Jul. 2007, pp. 176–181.

[71] R. Guodong, "Jump flooding algorithm on graphics hardware and its applications," M.S. thesis, Dept. Comput. Sci., Nat. Univ. Singapore, Singapore, Mar. 2008.

[72] S. H. Lo, "3D Delaunay triangulation of 1 billion points on a PC," *Finite Elements Anal. Des.*, vols. 102–103, pp. 65–73, Sep. 2015.

[73] J. R. Shewchuk, "Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator," in *Proc. Workshop Appl. Comput. Geometry*. Philadelphia, PA, USA: Springer, May 1996, pp. 203–222.

[74] D. D. Gajski, N. D. Dutt, A. C. Wu, and S. Y. Lin, *High—Level Synthesis: Introduction to Chip and System Design*. Cham, Switzerland: Springer, 2012.

[75] B. Curless and M. Levoy, "A volumetric method for building complex models from range images," in *Proc. 23rd Annu. Conf. Comput. Graph. Interact. Techn.*, Aug. 1996, pp. 303–312.

[76] S. Lloyd, "Least squares quantization in PCM," *IEEE Trans. Inf. Theory*, vol. IT-28, no. 2, pp. 129–137, Mar. 1982.

[77] Q. Du, V. Faber, and M. Gunzburger, "Centroidal Voronoi tessellations: Applications and algorithms," *SIAM Rev.*, vol. 41, no. 4, pp. 637–676, Jan. 1999.

[78] Q. Du, M. Emelianenko, and L. Ju, "Convergence of the Lloyd algorithm for computing centroidal Voronoi tessellations," *SIAM J. Numer. Anal.*, vol. 44, no. 1, pp. 102–119, Jan. 2006.

[79] F. Aurenhammer and R. Klein, "Voronoi diagrams," *Handbook Comput. Geometry*, vol. 5, no. 10, pp. 201–290, 2000.

[80] D. Arthur and S. Vassilvitskii, "How slow is the k -means method?" in *Proc. 22nd Annu. Symp. Comput. Geometry*, Jun. 2006, pp. 144–153.

[81] *Lloyd's Algorithm—Wikipedia, the Free Encyclopedia*, Wikipedia Contributors, San Francisco, CA, USA, 2023.

[82] Y. Ephraim and R. M. Gray, "A unified approach for encoding clean and noisy sources by means of waveform and autoregressive model vector quantization," *IEEE Trans. Inf. Theory*, vol. 34, no. 4, pp. 826–834, Jul. 1988.

[83] J. Cortes, S. Martinez, T. Karatas, and F. Bullo, "Coverage control for mobile sensing networks," *IEEE Trans. Robot. Autom.*, vol. 20, no. 2, pp. 243–255, Apr. 2004.

[84] M. Schwager, D. Rus, and J.-J. Slotine, "Decentralized, adaptive coverage control for networked robots," *Int. J. Robot. Res.*, vol. 28, no. 3, pp. 357–375, Mar. 2009.

[85] T. Elmokadem, "Distributed coverage control of quadrotor multi-UAV systems for precision agriculture," *IFAC-PapersOnLine*, vol. 52, no. 30, pp. 251–256, 2019.

[86] S. Chen, C. Li, and S. Zhuo, "A distributed coverage algorithm for multi-UAV with average Voronoi partition," in *Proc. 17th Int. Conf. Control, Autom. Syst. (ICCAS)*, Oct. 2017, pp. 1083–1086.

[87] R. N. Duca and M. K. Bugeja, "Multi-robot energy-aware coverage control in the presence of time-varying importance regions," *IFAC-PapersOnLine*, vol. 53, no. 2, pp. 9676–9681, 2020.

[88] T. Pei, J. Gao, T. Ma, and C. Zhou, "Multi-scale decomposition of point process data," *GeoInformatica*, vol. 16, no. 4, pp. 625–652, Oct. 2012.

[89] Q. Liu, M. Deng, J. Bi, and W. Yang, "A novel method for discovering spatio-temporal clusters of different sizes, shapes, and densities in the presence of noise," *Int. J. Digit. Earth*, vol. 7, no. 2, pp. 138–157, Feb. 2014.

[90] Q. Liu, J. Tang, M. Deng, and Y. Shi, "An iterative detection and removal method for detecting spatial clusters of different densities," *Trans. GIS*, vol. 19, no. 1, pp. 82–106, Feb. 2015.

[91] Y. Wan, T. Pei, C. Zhou, Y. Jiang, C. Qu, and Y. Qiao, "ACOMCD: A multiple cluster detection algorithm based on the spatial scan statistic and ant colony optimization," *Comput. Statist. Data Anal.*, vol. 56, no. 2, pp. 283–296, Feb. 2012.

[92] Y. Zhang, N. Ye, R. Wang, and R. Malekian, "A method for traffic congestion clustering judgment based on grey relational analysis," *ISPRS Int. J. Geo-Inf.*, vol. 5, no. 5, p. 71, May 2016.

[93] P. Zhao, K. Qin, X. Ye, Y. Wang, and Y. Chen, "A trajectory clustering approach based on decision graph and data field for detecting hotspots," *Int. J. Geographical Inf. Sci.*, vol. 31, no. 6, pp. 1101–1127, 2017.

[94] F. Xu, Y. Shi, M. Deng, J.-Y. Gong, Q.-L. Liu, and R. Jin, "Multi-scale regionalization based mining of spatio-temporal teleconnection patterns between anomalous sea and land climate events," *J. Central South Univ.*, vol. 24, no. 10, pp. 2438–2448, Oct. 2017.

[95] M. Deng, Q. Liu, T. Cheng, and Y. Shi, "An adaptive spatial clustering algorithm based on Delaunay triangulation," *Comput., Environ. Urban Syst.*, vol. 35, no. 4, pp. 320–332, Jul. 2011.

[96] A. Nasirahmadi, O. Hensel, S. A. Edwards, and B. Sturm, "A new approach for categorizing pig lying behaviour based on a Delaunay triangulation method," *Animal*, vol. 11, no. 1, pp. 131–139, 2017.

[97] K.-K. Oh, M.-C. Park, and H.-S. Ahn, "A survey of multi-agent formation control," *Automatica*, vol. 53, pp. 424–440, Mar. 2015.

[98] A. Schwab and J. Lunze, "A distributed algorithm to maintain a proximity communication network among mobile agents using the Delaunay triangulation," *Eur. J. Control*, vol. 60, pp. 125–134, Jul. 2021.

[99] F. Selimović, P. Stanimirović, M. Saračević, and P. Krtolica, "Application of Delaunay triangulation and Catalan objects in steganography," *Mathematics*, vol. 9, no. 11, p. 1172, May 2021.

[100] G. Bebis, T. Deaconu, and M. Georgiopoulos, "Fingerprint identification using Delaunay triangulation," in *Proc. Int. Conf. Inf. Intell. Syst.*, 1999, pp. 452–459.

[101] K. Feng and M. Haenggi, "Joint spatial-propagation modeling of cellular networks based on the directional radii of Poisson Voronoi cells," *IEEE Trans. Wireless Commun.*, vol. 20, no. 5, pp. 3240–3253, May 2021.

[102] B. P. J. de Lacy Costello, P. Hantz, and N. M. Ratcliffe, "Voronoi diagrams generated by regressing edges of precipitation fronts," *J. Chem. Phys.*, vol. 120, no. 5, pp. 2413–2416, Feb. 2004.

[103] S. Yoshioka and S. Ikeuchi, "The large-scale structure of the universe and the division of space," *Astrophys. J.*, vol. 341, p. 16, Jun. 1989.

[104] J. M. Drouffe and C. Itzykson, "Random geometry and the statistics of two-dimensional cells," *Nucl. Phys. B*, vol. 235, no. 1, pp. 45–53, May 1984.

[105] H. Chizari, M. Hosseini, T. Poston, S. A. Razak, and A. H. Abdullah, "Delaunay triangulation as a new coverage measurement method in wireless sensor network," *Sensors*, vol. 11, no. 3, pp. 3163–3176, Mar. 2011.

[106] P. Dechaumphai, S. Phongthanapanich, and T. Sricharoenchai, "Combined Delaunay triangulation and adaptive finite element method for crack growth analysis," *Acta Mechanica Sinica*, vol. 19, no. 2, pp. 162–171, Apr. 2003.

[107] H. Yamamoto, S. Uchiyama, and H. Tamura, "The Delaunay triangulation for accurate three-dimensional graphic model," *Syst. Comput. Jpn.*, vol. 27, no. 1, pp. 58–68, Jan. 1996.

[108] W. Schaap, "DTFE: The Delaunay tessellation field estimator," Ph.D thesis, Univ. Groningen, Groningen The Netherlands, 2007.

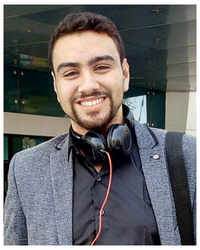
[109] M. Qi, K. Yan, and Y. Zheng, "GPredicates: GPU implementation of robust and adaptive floating-point predicates for computational geometry," *IEEE Access*, vol. 7, pp. 60868–60876, 2019.

[110] J.-D. Boissonnat and F. Cazals, "Smoothed analysis of algorithms: why the simplex algorithm usually takes polynomial time," *Journal ACM*, vol. 52, no. 6, pp. 917–997, 2005.

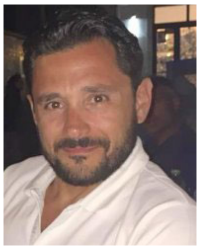
[111] S.-W. Cheng, T. K. Dey, and J. R. Shewchuk, *Delaunay Mesh Generation*. Boca Raton, FL, USA: CRC Press, 2010.

[112] N. Amenta and M. Bern, "Surface reconstruction by Voronoi filtering," *Discrete Comput. Geometry*, vol. 22, no. 4, pp. 481–504, Dec. 1999.

[113] H. Edelsbrunner and N. R. Shah, "Incremental topological flipping works for regular triangulations," *Algorithmica*, vol. 15, no. 3, p. 223, 1996.



YAHIA S. ELSHAKHS received the B.Sc. degree in engineering science in electrical science from Riga Technical University, Riga, Latvia, in 2022. After graduating, he completed an internship with the Electrical Department, Cyprus University of Technology, under the supervision of Dr. Kyriakos M. Deliparaschos. Since 2023, he has been an Embedded Engineer with Theobroma Systems Design and Consulting GmbH.



KYRIAKOS M. DELIPARASCHOS (Member, IEEE) received the B.Eng. degree (Hons.) in electronics engineering from De Montfort University, Leicester, U.K., the M.Sc. degree in mechatronics from De Montfort University and the National Technical University of Athens (NTUA), Greece, and the Ph.D. degree from the Department of Signals, Control and Robotics, School of Electrical and Computer Engineering, NTUA.

With a background in academia and research, he has held various roles, including a Visiting Lecturer with New York College and IST College, Athens, Greece, and a Postdoctoral Research Associate with the Intelligent Robotics and Automation Laboratory, NTUA, and the Robotics Control and Decision Systems Laboratory, Cyprus University of Technology (CUT). He was a Postdoctoral Research Fellow with the Centre for Telecommunications Research, Trinity College, Dublin University, and a Senior Research Fellow with Cranfield University. He is currently a Special Teaching Staff with the Department of Electrical and Computer Engineering and Informatics, CUT. His main research interests include embedded systems, hardware acceleration (particularly FPGA and system-on-a-chip implementations), sensor fusion in navigation and control, resilient cyber-physical systems, intelligent control of industrial systems, robotics real-time navigation, and medical robotics for minimally invasive surgery.

THEMISTOKLIS CHARALAMBOUS (Senior Member, IEEE) received the B.A. and M.Eng. degrees in electrical and information sciences from the Trinity College, University of Cambridge, and the Ph.D. degree from the Control Laboratory, Engineering Department, University of Cambridge.

Following the Ph.D. degree, he was a Postdoctoral Researcher with Imperial College London, the Royal Institute of Technology (KTH), and the Chalmers University of Technology. In January 2017, he joined Aalto University, as a Tenure-Track Assistant Professor. In September 2018, he was awarded the Academy of Finland Research Fellowship. In July 2020, he was appointed as a tenured Associate Professor. In September 2021, he joined the University of Cyprus, as a Tenure-Track Assistant Professor. He remains associated as a Visiting Professor with Aalto University. His research interests include the design and analysis of (wireless) networked control systems that are stable, scalable, and energy efficient. The study of such systems involves the interaction between dynamical systems, their communication, and the integration of these concepts. As a result, his research is interdisciplinary combining theory and applications from control theory, communications, and network and distributed optimization.

GABRIELE OLIVA (Senior Member, IEEE) received the M.Sc. and Ph.D. degrees in computer science and automation engineering from Roma Tre University, Rome, Italy, in 2008 and 2012, respectively. He is currently an Associate Professor of automatic control with the Campus Bio-Medico University of Rome, Italy, where he directs the Complex Systems and Security Laboratory (CoserityLab). His main research interests include distributed multi-agent systems, optimization, decision-making, and critical infrastructure protection. Since 2019, he has been serving as an Associate Editor for the Conference Editorial Board of the IEEE Control Systems Society. Since 2020, he has been serving as an Academic Editor for *PLOS One* on subject areas, such as systems science, optimization, and decision theory. Since 2022, he has been an Associate Editor of IEEE Control Systems Letters.

ARGYRIOS ZOLOTAS (Senior Member, IEEE) received the B.Eng. degree (Hons.) from the University of Leeds, the M.Sc. degree from the University of Leicester, and the Ph.D. degree from Loughborough University, U.K.

He was a Research Fellow with the Department of EEE, Imperial College London, and then held academic posts with Loughborough University, the University of Sussex, and the University of Lincoln. He was a Visiting Professor with the GIPSA-Laboratory, Grenoble INP, in 2018. Since 2019, he has been a Reader with SATM, Cranfield University, heading the Autonomous Systems Dynamics and Control Research Group, Centre for Autonomous and Cyber-Physical Systems. His research interests include advanced control, autonomous systems, and AI-based control applications.

Dr. Zolotas is a fellow of the U.K. HEA and the U.K. Institute of Measurement and Control. He serves on the editorial boards for a number of journals. He served as an Associate Editor for IEEE Transactions on Control Systems Technology and IEEE Control Systems Society Letters for several years.