Master's Thesis

Development of data analysis tools to study clusters of particles in turbulent flows and their time evolution

Tomás Gil, Álvaro
Villafañe Roca, Laura

Escuela Técnica Superior de Ingeniería del Diseño
Master's Degree in Aeronautical Engineering - 2019/20

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Contents

List of Figures

Chapter 1

Introduction

Turbulent flows carrying particles with larger inertia than that of the fluid element occur frequently within environments of both nature and engineering. Phenomena ranging from the formation of rain within clouds, the atmospheric transport of solid pollutants, or the deposition of sediments on river banks, to the flow in a combustion chamber or the volumetric absorption of solar radiation in a solar receiver all involve the interaction between inertial particles and turbulent flows. As a result, there has been great interest in the last decades to not only acquire an understanding of the underlying physical mechanisms playing a role in particle-laden turbulent flows, but also to develop quantitative tools to analyze the characteristics of these flows.

The main phenomena characterizing particle-laden turbulent flows are associated to the dynamic interaction between inertial particles and turbulent flow, where the response of a particle to the flow turbulence is directly dependent on its own inertia and on the scales of the flow. More precisely, the degree to which an inertial particle is capable of following the flow is described by the Stokes number, defined as the ratio of the particle response time to some representative time scale in the flow, the latter being usually the Kolmogorov time scale [1]. For instance, particles with very low Stokes numbers will simply follow the flow as a flow tracer, whereas particles associated with high values of such a number are not affected by any turbulent structure in their motion. Nevertheless, when the Stokes number describing a turbulent particle-laden flow is of the order of one, particles do respond to some degree to the turbulence of the flow, but are however not able to completely follow the curved streamlines of this turbulence. What results in these specific cases is that particles accumulate producing a notoriously in-homogeneous concentration field, dynamically forming particle structures in which the number density is significantly higher than the mean, as well as leaving regions of the domain completely absent of particles [2] [1] [3]. Based on this, the Stokes number can also be conceived as describing the degree to which turbulent eddies can modify the concentration field of the diluted particles [1]. This in-homogeneous concentration field is traditionally explained by the interaction of a particle that is heavier than the surrounding fluid with a turbulent structure, resulting in the outward acceleration of the particle due to centrifugal forces.

This phenomenon, by which the inertial particles diluted in the flow accumulate preferentially in regions of high strain rate within the flow is known as *preferential concentration*. Nevertheless, one can distinguish three aspects of this phenomenon: while as preferential concentration refers simply to the preferential movement of particles into particular regions, *clustering* is conceived to be the fact that particles tend to accumulate and segregate, and *clusters* are groups of particles remaining in proximity on time scales relatively larger than a particular turbulent time scale [4]. In opposition to the definition of a particle cluster as a region of the flow of high number density of particles, it is possible as well to define a *particle void* as a region of low number density of particles. In any case, preferential concentration refers to the mechanisms causing both clustering as well as the temporal evolution of particle clusters and voids. A review of turbulent dispersed multiphase flow is presented in Balachandar et al. (2010) [5].

Several different methods have been developed in the last decades to quantify the degree of preferential concentration within turbulent particle-laden flows, as well as to identify regions of particle clusters and voids within the flow, an overview of the most commonly employed of these being presented in the work by Monchaux et al. (2012) [4]. On the one hand, a large number of so-called box-counting methods exist, based on the separation of the domain into boxes of equal size and the later counting of the number of particles within each box. In the work by Fessler et al. (1994) [1], the level of preferential concentration of the flow is quantified by comparing the distribution of the number of particles per box in the preferentially concentrated case with that of the case of purely uniformly distributed particles, the latter being associated to a Poisson distribution. Given that in the preferentially concentrated case, by definition, regions of relatively high or low particle number density will occur more frequently, a clustering index can be extracted from the difference between both distributions, this quantifying the degree of preferential concentration [1]. Similarly, Aliseda et al. (2002) [2] analyzes the effect of the box size on the resulting preferential concentration, extracting that clustering is maximum at length scales of the order of ten times the Kolmogorov length scale. Moreover, in the alternative index based on box-counting proposed by Villafañe et al. (2016) [6], a qualification of preferential concentration is presented which unlike the index presented by Fessler et al. (1994) [1], is minimally influenced by the number of particles in the domain if a sufficient number of samples of the spatial particle distributions are available [6]. Furthermore, in order to identify particle clusters employing box-counting methods, one can define a concentration threshold such that boxes above such threshold are connected and considered as forming particle clusters [2]. Nevertheless, it is necessary to take into account that these methods require the extrinsic introduction of an arbitrary length scale (i.e a box size), as well as a user defined threshold [4]. In dilute conditions, the box size is dictated by the mean particle concentration and not necessarily by a flow length scale.

On the other hand, an alternative method for quantifying preferential concentration based on Voronoï tessellations has been introduced by Monchaux et al. (2010) [7]. These methods are based on the decomposition of the domain into independent cells associated to each particle, where each cell is the region of the domain closer to its associated particle than to any other particle [4]. Given that the volume of each Voronoï cell is inversely related with the concentration of particles at that region, a local concentration field can be obtained from the generated field of Voronoï volumes, and furthermore a quantification of preferential concentration can be obtained from the standard deviation of the probability density function of the resulting Voronoï volumes, without requiring the introduction of an arbitrary length scale [7]. Monchaux et al. (2010) [7] also delves into the identification of particle clusters by intersecting the probability density function of Voronoï volumes of the preferentially concentrated case with that of uniformly distributed particles. In this way, a maximum Voronoï volume which a cell can have to be classified as a cluster is determined, and multiple Voronoï cells fulfilling this criterion can be connected in order to form a single particle cluster. The main advantage of this method with respect to box-counting alternatives stems from its independence from any length scale or number density threshold in order to identify particle clusters, albeit a generally higher computational cost [4].

None of these methods of quantification of preferential concentration and identification of particle clusters in particle-laden turbulent flows actually employs clustering algorithms. Therefore, it is worth exploring whether a more efficient method for particle cluster identification can be developed from the application of these techniques. *Clustering algorithms* can be defined as techniques of organizing data samples within a dataset into sensible groupings, without employing labels which tag each object with prior identifiers [8]. Due to the fact that the objective of these algorithms is to sort unlabeled datasets into classes, and not to determine discriminative rules with which to classify new data samples into existing categories of the dataset, clustering algorithms form an important part of *unsupervised learning* within the area of machine learning [8]. In this sense, it is possible to redefine a *cluster* within this context as a group of data samples which are alike, and are at the same time different from any data sample belonging to another cluster [9]. For the purposes of the current study, the term *particle cluster* refers a the group of densely-placed particles caused by the preferential concentration of particles embedded in turbulent flow, whereas the term *cluster* represents a group of data samples resulting from the application of a clustering algorithm. For an introduction to the basic concepts underlying most clustering algorithms, the reader is referred to the work of Jain et al. (1988) [8].

With regards to these techniques of organizing a dataset into groups, a myriad of approaches have been developed in past years, a review of them being exposed in detail in the work by Xu et al. (2015) [9]. These algorithms can be sorted based on their operating principle. For instance, clustering algorithms based on partition are commonly used, of which K-means as introduced by MacQueen et al. (1967) [10] and PAM and CLARA as per Kaufman et al. (1990) [11] are notorious. Moreover, algorithms based on hierarchy, such as BIRCH as presented by Zhang. et al. (1996) [12], or based on distribution, of which Gaussian Mixture Models as introduced by Rasmussen et al. (1999) [13] is most famous, are also frequently employed. Futhermore, clustering algorithms based on density, of which DBSCAN, as developed by Ester et al. (1996) [14], and OPTICS, introduced by Ankerst et al. [15] are of particular relevance to the current study. For this latter category, a cluster is conceived as the group of data samples belonging to the same region of high density of data samples [9], a definition which is clearly reminiscing of that of the particle cluster. In Ester et al. (1996) [14], DBSCAN is presented as an efficient clustering algorithm capable of defining clusters of arbitrary shape with minimal knowledge of the domain. On the other hand, OPTICS, as introduced by Ankerst et al. (1999) [15], is a density-based clustering algorithm sharing many core principles of operation with DBSCAN, which however does not produce a clustering of the dataset explicitly, but instead allows for a representation of its clustering structure. In later work by Schubert et al. (2017) [16], the utility of DBSCAN is defended and heuristics for choosing adequate parameters are presented.

Based on the wide variety of clustering algorithms developed in the past decades, and the fact that these have not been applied in order to identify particle clusters within particle-laden turbulent flows, it seems appropriate to explore whether these techniques can be of any use in analyzing particle clusters. More specifically, it is worth questioning if considering a dataset of particle positions, one can group data samples representing particle positions into clusters, each of these in turn representing a particle cluster. Moreover, it is also necessary to investigate the limitations and requirements of such an application when compared to the existing methods of quantification of preferential concentration, in order to determine whether clustering algorithms are actually worth employing when identifying particle clusters. Additionally, it is worth exploring if based on a novel application of clustering algorithms to identification of particle clusters, one can develop a method for the tracking of a single particle cluster in time, in order to extract how particular characteristics of the particle structure evolve during its lifetime.

Therefore, a main objective of the current study is to apply density-based clustering algorithms to a dataset made up of positions of particles embedded in turbulent flow, in order to identify particles belonging to a particle cluster, and thus to group them defining three-dimensional particle structures. In order to evaluate the developed method, it is another objective of this study to compare the obtained results with that of an existing robust method for particle cluster identification, such as the application of Voronoï tessellations presented in Monchaux et al. (2010) [7]. This evaluation focuses on comparing the resulting particle cluster volumes and topologies, as well as the computational performance of both alternative methods.

With this objective, the current study simultaneously applies two density-based clustering algorithms to carry out the identification of particle clusters within a dataset of positions of particles embedded in turbulent flow: OPTICS and DBSCAN. Moreover, given the three-dimensional nature of the employed dataset, special additional steps have to be taken in order to more efficiently perform this analysis. On the one hand, the three-dimensional domain is simplified into several adjacent two-dimensional domains, and on the other hand, the information expressing the topology of a particle cluster is condensed into a reduced set of points representing each particle cluster. These additional steps reason the method's potential advantages with respect to an alternative method based on Voronoï tessellations. While on the one hand, the simplification of the three-dimensional into several two-dimensional domains allows for a milder computational cost, the condensation of each particle cluster topology into a reduced set of representative points permits further analyses concerning each cluster's morphology and evolution to be carried out with ease. Nevertheless, possible disadvantages of the current method with respect to a purely three-dimensional approach stem from the fact that the simplification of the three-dimensional domain into several adjacent two-dimensional domains implies a certain degree of loss of information.

In addition, the current study has the significant objective of developing a method for tracking particle structures over time. This novel technique is based on the previous simplification of a particle cluster topology into a set of representative points, and permits one to determine how certain characteristics of a particle cluster develop temporally.

This report is structured as follows. In Chapter 2, the implemented methods in each step of this analysis are presented in detail. After describing the carried out simplification of a three-dimensional domain and introducing the main principles of the employed clustering algorithms, each step of the identification of a three-dimensional particle cluster is displayed. The last part of this chapter exposes the implemented methods of temporal tracking of a particle cluster. Furthermore, in Chapter 3, the implemented technique of particle cluster identification is evaluated with an alternative method based on Voronoï tessellations, both by comparing the resulting cluster volumes as well as the characteristic topology of each cluster. The last part of this chapter carries out a comparison of the computational cost of both alternative methods. Lastly, the conclusions to the current study are presented.

Chapter 2

Methods of Analysis

2.1 Methodology for 3D Spatial Characterization

The dataset employed in this study consists on a list of particle positions in three-dimensional space for a given time instant. The particle positions were taken from the simulation of turbulent duct flow laden with small heavy inertial particles, performed in the framework of the PSAAPII program at Standford University. For details on the numerical framework and methodology see Esmaily et al. (2020) [17]. Whereas the fluid domain in this simulation has been computed based on an Eulerian approach, where all scales of the turbulent flow are resolved via direct numerical simulation (DNS), the dynamics of each individual particle have been determined with a Lagrangian approach, assuming that the only forces affecting particle motion are those of Stokes drag, gravity, and collisions as per the hard-sphere model. More precisely, the point-particle model has been utilized, where individual particles within the flow are represented by means of a force at a punctual location. This model is valid as long as the flow structures are much larger than the size of a particle, such that particle finite-size effects are neglected. Moreover, the interaction between the continuous fluid field and the discrete set of particles has been modeled with two-way coupling, where each particle exerts a force on the flow which is distributed among cells adjacent to the particle. The dimensions of the square duct utilized in this simulation as well as the characteristics of the flow and particles are presented in Table 2.1.

Given the three-dimensional nature of the turbulent structures in this flow, which introduce the same degree of complexity to the structures of particles within the flow, it is necessary to analyze these structures from a three-dimensional standpoint, at the same time taking into account the computational requirements that any calculation regarding three-dimensional turbulent flow entails. In this way, the current study has sought to decompose the complete three-dimensional analysis of the computational domain into several two-dimensional problems.

Duct Length [m]	0.27
Duct Width [m]	0.04
Friction Reynolds Number Re_τ	570
Kolmogorov Length Scale η	7e-5
Stokes Number St_η	12
Ratio of Particle Diameter to Kolmogorov Length Scale d_p/η	0.17

Table 2.1: Characteristic of Data-Generating Simulation - The friction Reynolds number Re_τ is calculated based on the friction velocity and the duct half-height, whereas η and St_η denote the Kolmogorov length scale based on the mean dissipation, and the Stokes number based on the Kolmogorov time scale respectively.

More precisely, the computational volume has been split into a series of planes parallel to the stream-wise direction X of the duct and with a constant value of Z, such that if these planes are distant enough from the duct walls, the parameters employed for the analysis of an individual two-dimensional snapshot are applicable to all of the snapshots defining the domain. Then, each of these two-dimensional computational domains will be generated by normally projecting all particle positions of particles within a certain distance of each plane, as if each plane simulated the readings of a laser sheet of a thickness of twice such distance. This procedure requires the adequate specification of three parameters: the minimum separation of each two-dimensional domain with respect to duct walls S, the separation between adjacent planes of constant Z, ΔZ, and the thickness of each simulated projection sheet t.

The first step of this processing of the original dataset involves a reduction of the domain under analysis. Since the introduction of phenomena relevant to the presence of the duct wall hinders the analysis of the isolated nature of the particle structures that occur in turbulent particle-laden flow, it is necessary to exclusively take into account a limited region of the domain which is far enough from any duct wall and allows for feasible computational requirements. In order to avoid the effects produced by wall proximity to be displayed in the extracted dataset, a minimum separation S from the wall is to be ensured in the Y and Z directions. Based on the distribution presented in Figure 2.1, and taking into account that the duct's section is square, it is reasonable to infer that a separation of 0.008 m from every wall is enough to avoid effects caused by wall proximity to be included in the extracted dataset, thereby focusing the study in the region of the domain of constant particle number density. As will be discussed later on, this is essential for the hyperparameters determined in Sections 2.2.3 and 2.2.4 to be applicable to the whole dataset, since these are particularly adapted to the concentration of particles at a single level of Z. This first step of processing is visualized in Figure 2.2, where a reduced three-dimensional region of the original duct domain is extracted.

The next step of this processing involves, within the previously extracted domain, the definition of a number of parallel planes of constant Z with a specific separation ΔZ between them. This separation must guarantee the continuity of topological properties of particle structures within adjacent planes while at the same time keeping an advantage with respect to the fully three-dimensional approach in terms of computational requirements. Each of

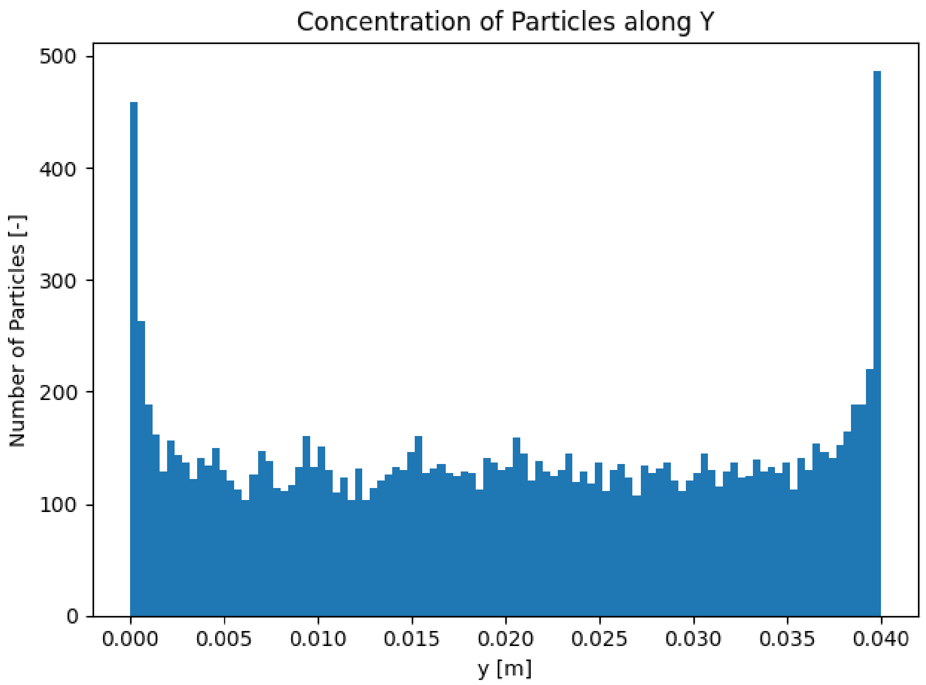

Figure 2.1: Concentration of Particles along Y in the unprocessed, original dataset

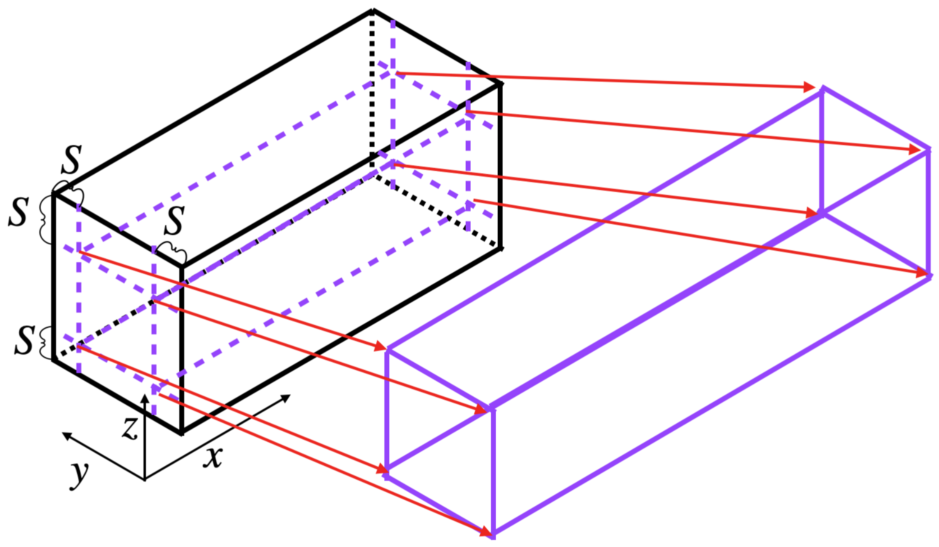

Figure 2.2: Extraction of Domain Separated S from all Duct Walls - In black: duct boundaries. In purple, extracted domain.

these parallel planes bisects an associated sheet of thickness t, such that all particles within such sheet are normally projected to the bisecting plane. Therefore, each two-dimensional domain resulting from this procedure is made up of the particle positions which are normally projected to each of these planes. As a result, from a large list of three-dimensional positions scattered about the computational volume, what is obtained from this process is an ordering of the positions into a discrete set of possible elevations normal to the stream-wise plane. This procedure is visualized in Figure 2.3.

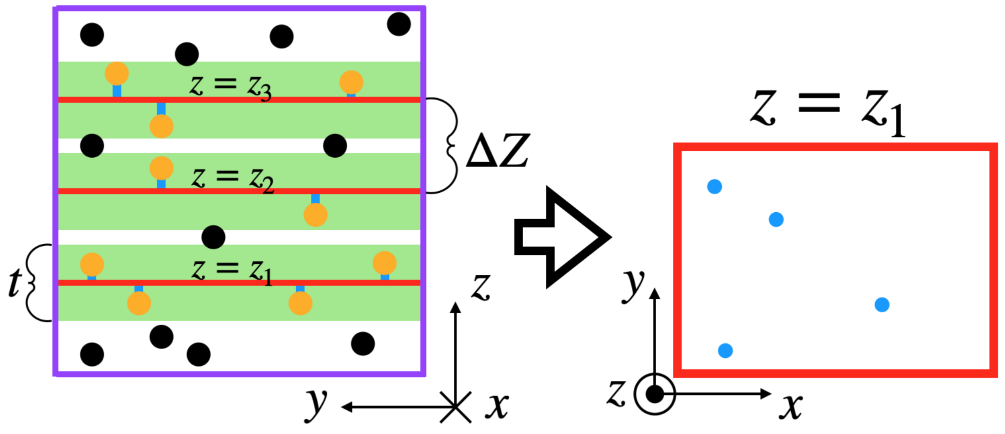

Figure 2.3: Projection of Particle Positions of Particles included within each Sheet of Thickness t whose Bisecting Plane is Separated ΔZ from Adjacent Bisecting Planes - In purple, previously extracted domain. In red: generated bisecting planes. In black dots, particle positions not captured by this simplification. In yellow dots, particle positions captured by this simplification. In blue dots, projected particle positions.

When specifying the apt separation ΔZ between adjacent planes of constant Z, it is necessary to bear in mind that such specification is closely tied with that of the thickness t associated to each projection sheet. In the extreme in which sheet thickness is close to null, what occurs is that the number of particles associated to each computational plane is extremely reduced, at the same time allowing that the processed Z value of the particle does not deviate much from its original Z value. On the contrary, the larger the sheet thickness t, the greater the amount of particles associated to a plane, and the greater the difference will be between their original Z coordinate and the Z value of the projection representing the particle. On the other hand, the effect of the separation between adjacent planes ΔZ has to do with the continuity of particle structure topologies in adjacent computational surfaces. If this separation is large, it makes sense to assume that the difference in particle structure appearance will be greater than if this separation is less significant.

Moreover, if the spacing between adjacent computational planes is large while the thickness of each sheet is small, a great amount of particles within sheets will not be associated to a plane, and their position would thus be lost. The opposite is to happen if the spacing between adjacent computational planes is minute when compared to the associated sheet thickness, where the information encapsulated in each Z level will actually describe much of what is happening at very different values of Z, and much of the particle positions within the original three-dimensional data set will be included in the processed results.

Thus, it is mainly sought to employ a sheet thickness capable of describing particle structures relevant to a single level of Z, and a separation between adjacent planes that allows for a certain continuity of structure topologies. In an ideal case, with an infinite number of particles, one would employ a sheet thickness equivalent to the Kolmogorov scale, as this would allow for a precise encapsulation of the particle structures caused by the turbulence in the flow. Nevertheless, the number area density resulting from this sheet thickness is too low. In the current study, a separation between adjacent planes equal to $11.42 \cdot \eta$ is utilized, η being the associated Kolmogorov length scale, and the sheet thickness t is imposed to be half of this separation. This combination of parameters is thought to adequately represent encapsulate the phenomena under analysis without incurring a computational cost close to the cost of a direct three-dimensional approach.

Table 2.2 presents the domain dimensions of the dataset based on which a decomposition into a set of two-dimensional planes is carried out, each including the normal projections of nearby particles, as well as the parameters employed in this decomposition.

Magnitude	[m]	length / H
Domain Size in X	[0, 0.27]	[0, 6.75]
Domain Size in Y	[0, 0.04]	[0, 1.00]
Domain Size in Z	[0, 0.04]	[0, 1.00]
Reduced Domain in X	[0.108, 0.162]	[2.7, 4.05]
Reduced Domain in Y	[0.008, 0.032]	[0.2, 0.8]
Reduced Domain in Z	[0.013, 0.028]	[0.325, 0.7]
Adjacent Plane Separation ΔZ	0.0008 ($11.42 \cdot \eta$)	0.02
Planar Projection Sheet Thickness t	0.0004 ($5.71 \cdot \eta$)	0.01
Average Number of Particles per computational Plane	29272 [-]	-
Average Planar Density	22,586,419 [$1/m^2$]	-

Table 2.2: Dimensions of the unprocessed dataset and dimensions resulting of the described decomposition into planes parallel to the stream-wise direction. H is the duct width, corresponding to 0.04 m, and η is the Kolmogorov Length Scale.

What results from the described processing of the original dataset is a simplification of the original three-dimensional domain into a set of two-dimensional domains which, for an adequate selection of defining parameters dependent on the original dataset's particle number density, can be further analyzed to group particle positions into particle clusters. Given the imposed separation with respect to all duct walls, a clustering algorithm with a single parameter configuration can be applied to all two-dimensional domains.

2.2 Clustering Analysis of a 2D Snapshot

2.2.1 Introduction to Clustering Algorithms

For every two-dimensional snapshot in Z representing projected particle positions within a specific computational plane, it is necessary to group particle positions into particle clusters, in order to define the clusters whose topology and temporal evolution are to be studied in later phases of this study. Treating the dataset containing particle positions as an arbitrary multidimensional dataset, in which every data sample describes a particle position within the computational plane, the grouping of particle positions can be performed making use of clustering algorithms. Thus, a description of the working principles, varieties, and applicability of these techniques is necessary within the context of this study.

Clustering methods are aimed at generating hypotheses, detecting anomalies, and identifying particular features within an unlabelled dataset [18]. This quintessential branch of unsupervised learning can be seen either as providing tools for a preliminary exploration or as allowing for the compression of the original dataset into a representative set of clusters [18]. Classically, a clustering algorithm can be described as a method for sorting data samples into groups such that [9]:

- Instances within the same cluster must be as similar as possible.
- Instances in different clusters must be as different as possible.

Based on these two principles, it is reasonable to infer that a great deal of importance is placed on the measures of similarity and dissimilarity employed in the clustering method, and on how this measure is processed in order to classify the dataset into an appropriate set of clusters. When dealing with quantitative data, it is common to employ distance functions when determining the similarity between a data sample and data samples in the same cluster or pertaining to a different cluster.

The most commonly employed distance function when evaluating the similarity of two data points x_i and x_j is the Minkowski distance, which for a d-dimensional data space, depending on an additional parameter n, follows Eq. (2.1) [9]. In the case in which $n = 1$, the Minkowski distance represents the city block or Manhattan distance, basically adding the difference in each of the dimensional directions between both data samples. However, in the case in which $n = 2$, what one obtains out of this function is a classic Euclidean distance. Lastly, it is worth noting that in the case in which $n \to \infty$, the Chebysev distance results [9].

$$d(x_i, x_j, n) = \Big(\sum_{l=1}^{d} |x_{i,l} - x_{j,l}|^n \Big)^{(1/n)} \tag{2.1}$$

Another popular distance function mentioned in [9] is the standardized Euclidean distance, which weights the classic Euclidean distance based on the standard deviation s_l of the dataset in a dimensional direction l. This function is presented in Eq. (2.2).

$$d(x_i, x_j) = \Big(\sum_{l=1}^{d} |\frac{x_{i,l} - x_{j,l}}{s_l}|^2 \Big)^{(1/2)} \tag{2.2}$$

Moreover, in order to characterize the difference in orientation between two data samples as a parameter defining their distance, the Cosine distance can be employed, taking into account the scalar product of both data samples x_i and x_j as is visible in Eq. (2.3). This distance can also be seen as directly dependent on the angle θ between both data samples.

$$d(x_i, x_j) = 1 - \cos\theta = \frac{x_i \cdot x_j}{|x_i||x_j|} \tag{2.3}$$

Based on these commonly employed distance measures, a myriad of different techniques exist to group multi-dimensional data samples into representative clusters, depending on the strategy by which task is tackled. In any case, a set of relevant factors determine whether the application of a particular algorithm is convenient.

Firstly, one has to take into account whether the number of clusters in the grouped result is a preset parameter of the method or whether it is automatically determined based on the dataset's structure. In the case that the number of clusters is preset and is extremely different from the possible number of clusters which can appear due to the nature of the dataset, it may occur that visible groups within the dataset are split into multiple clusters, that a cluster is not associated to a large enough number of data samples to be considered representative, or that a single cluster is attached to multiple perceivable groups within the dataset. Thus, if a method in which the number of clusters is a preset parameter is employed, it is necessary to ensure that this number of clusters is coherent with the dataset's structure. Two common clustering algorithms in which the number of clusters is a preset parameter defining by the user are Gaussian Mixture Modeling and k-Means Clustering [9].

Secondly, it is necessary to contemplate whether the employed clustering technique is capable of defining clusters with an arbitrary shape or whether it is designed to detect data sample groups of a particular topology. For example, in the case of k-Means clustering, data samples are grouped based on their proximity to a punctual cluster center, such that data samples within the same actual group within the dataset may be assigned to different clusters if they have closer cluster centers nearby. Thus, it can be expected for the clusters resulting from applying k-Means to a dataset to have something similar to a circular shape, depending of course on the dataset structure and the number of specified cluster centers.

Lastly, a minor peculiarity of certain clustering algorithms which, as will be seen later on, is very useful for the current application, is whether the clustering method is capable of detecting and filtering out noise within the dataset or not. It may be the case, as in k-Means algorithm, that all particles must be assigned to their closest cluster center. Thus, what results is that isolated data samples may be associated to very dissimilar cluster centers, and the quality of the clustering is reduced [9]. Certain clustering algorithms, such as OPTICS and DBSCAN, are capable of classifying data samples as noise, and as a consequence do not attach a cluster label innecessarily.

2.2.2 DBSCAN and OPTICS

In the current study, in order to determine whether each particle within the two-dimensional snapshot in Z can be classified as belonging to a particle cluster or as belonging to a particle void, it is necessary to employ a density-based clustering algorithm capable of separating the relatively denser regions assigned to particle clusters from the relatively less dense regions assigned to particle voids. Being OPTICS and DBSCAN the main representatives of this family of clustering techniques, their employment within the context of this study is more than reasonable. As will be seen later on, both methods are particularly useful for the current application due to their ability of detecting clusters of arbitrary shapes without an *a priori* specification of the number of clusters to be defined, and due to the fact that not all data samples are grouped within a cluster, but may instead be labeled as noise samples. Moreover, the current study employs the description of the clustering structure of the dataset which OPTICS allows for in order to determine an adequate set of parameters, adapted to the average number density of the processed dataset, with which to perform the DBSCAN clustering analysis of each previously generated two-dimensional domain.

DBSCAN and OPTICS both belong to a same family of clustering algorithms based on density, where the main principle behind the clustering procedure is to observe the areas of high density within the dataset, and thus to group data samples belonging to a same area of high density within the same cluster [9]. In other words, these clustering techniques rely on the fact that there are regions of the dataset in which the density of data samples is relatively higher than in other regions. Based on this, it makes sense to connect all data samples within the same region of high density by associating them to the same cluster.

As is expected, a question arises which is similar as that appearing when defining a particle cluster or a particle void: how is a region of the dataset to be defined as dense? In brief, a density threshold must be set, and thus these algorithms require the definition of a maximum volume and a minimum number of data samples to indirectly fix this threshold. This requires the specification of a minimum number of samples $MinPts$, which when combined with a set neighborhood radius ϵ defines this number density threshold. Based on these parameters, for each data sample surpassing this density threshold, the number of neighbors within a known radius ϵ must be at least a specified value $MinPts$ [15].

In order to properly understand the basis on which OPTICS and DBSCAN are founded, it is necessary to introduce a series of concepts innately relevant to density-based clustering. The first of these terms to introduce is direct density-reachability [14].

Definition 2.2.1. Directly Density-Reachable Points: A data sample p is directly density-reachable from a sample q with respect to ϵ and $MinPts$ in a set of objects D if:

1. $p \in N_\epsilon(q)$, where $N_\epsilon(q)$ is the subset of D contained in the ϵ-neighborhood of q.
2. $Card(N_\epsilon(q)) \geq MinPts$, where $Card(N_\epsilon(q))$ is the number of elements in the set $N_\epsilon(q)$.

It is worth taking into account that for two data samples, this condition is not necessarily reciprocal, since it may be the case that both data samples are within the ϵ-neighborhood of each other but only one of the two fulfills the second condition. This second condition is called the core object condition, defining the data sample which satisfies it as a core object. In short, it is possible for a data sample to be directly density-reachable only from a core object, as is made visible in Figure 2.4, where p is directly density-reachable from q but q is not directly density-reachable from p.

On the other hand, this sort of connectivity within data samples can exist less directly, involving intermediate data samples connecting both considered points. In this way, the condition of density-reachability is defined [14].

Definition 2.2.2. Density-Reachable Points: A data sample p is directly density-reachable from a sample q with respect to ϵ and $MinPts$ in a set of objects D if there is a chain of objects $p_1, ..., p_n$ such that $p_1 = q$, $p_n = p$, $p_i \in D$, and p_{i+1} is directly density-reachable from p_i with respect to ϵ and $MinPts$.

It is worth noting that for the definition of density-reachable points, all except the last of the data sample in the chain $p_1, ..., p_n$ must necessarily be core objects, this causing this condition to be non-reciprocal. This condition is made visible in Figure 2.5.

Moreover, a core principle necessary to define a cluster within the context of these two density-based clustering method is density-connectivity [14].

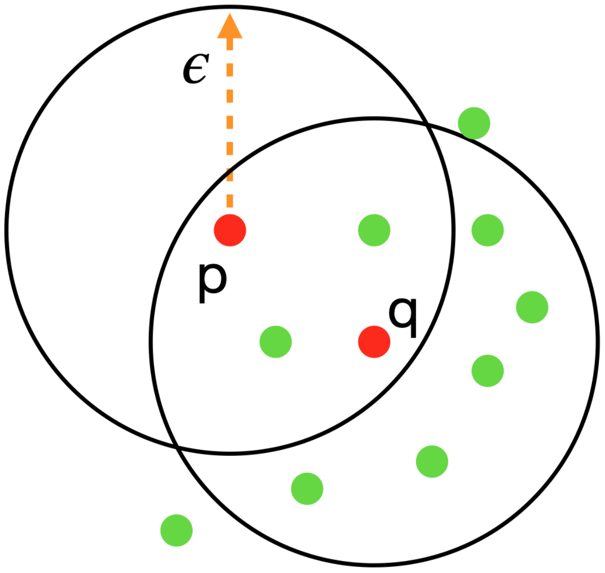

Figure 2.4: Directly Density-Reachable Points

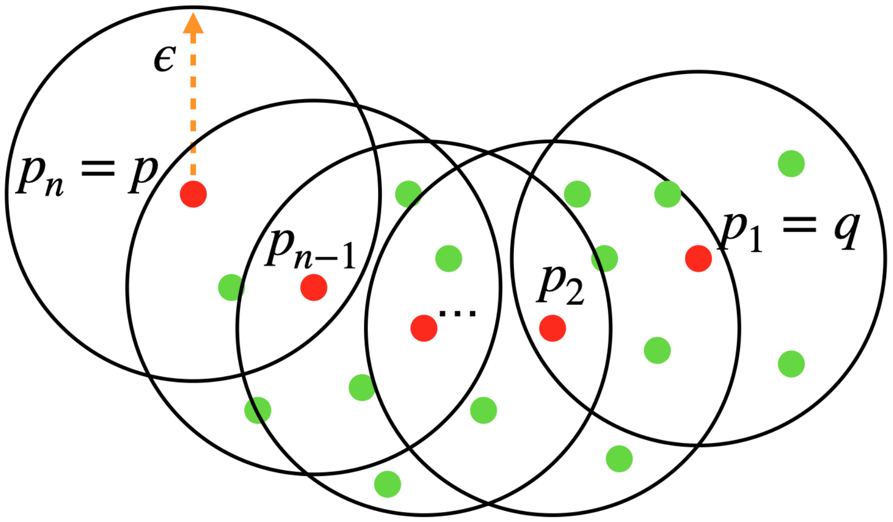

Figure 2.5: Density-Reachable Points

Definition 2.2.3. Density-Connected Points: A data sample p is density-connected to another sample q with respect to ϵ and $MinPts$ in a set of objects D if there is an object $o \in D$ such that both p and q are density-reachable from o with respect to ϵ and $MinPts$ in D.

Differently to the previous two, this condition is symmetric, since the definition only requires object o to be a core object, as well as all of the other samples connecting p and q with

o. As a result, it is now possible to define the conditions for a cluster to be assigned as well as for a data sample to be labeled as noise [14].

Definition 2.2.4. Cluster and Noise: Given a set of data samples D, a cluster C with respect to ϵ and $MinPts$ in D is a non-empty subset of D which satisfies:

1. $\forall p, q \in D$, if $p \in C$ and q is density-reachable from p with respect to ϵ and $MinPts$, then also $q \in C$
2. $\forall p, q \in C$, p is density-connected to q with respect to ϵ and $MinPts$ in D.

All data samples not included within a cluster are labeled as noise.

Based on this definition of a cluster, it is clear that since the fact that two points are density-reachable does not necessarily imply that both are core objects, not all data samples included within a cluster are necessarily core objects. Nevertheless, it is necessary for a cluster to contain at least one core object for density-reachable and density-connected points to occur, since a cluster can be seen as the set of all data samples in D which are density-reachable from an arbitrary core object in the cluster. On the other hand, it is directly reasonable that all noise data samples are non-core objects.

As the simpler algorithm of the two exposed here, DBSCAN (Density-Based Spatial Clustering of Applications with Noise) employs the definitions presented up to now. This algorithm begins by examining the ϵ-neighborhood of every data sample in the dataset. Then, if the ϵ-neighborhood $N_\epsilon(p)$ of a sample p contains more than $MinPts$ points, a new cluster C is created including p and all of its ϵ-neighbors. DBSCAN proceeds by analyzing the ϵ-neighborhood of all of the unprocessed points q in C. For each of these points q, if $N_\epsilon(q)$ contains more than $MinPts$ points, the ϵ-neighbors of q which are not in C are added to the cluster and their ϵ-neighborhood is analyzed in the following step. Once no new points can be added to the current cluster, the algorithm continues examining the ϵ-neighborhood of unprocessed data samples in the dataset [14].It is worth noting that the results of a DBSCAN procedure are deterministic, although they are expected to vary as soon as the dataset is permuted [16].

The simplicity of this algorithm relies on its direct dependency on the two user-defined parameters ϵ and $MinPts$ which explicitly state what the minimum density of data samples necessary to define a cluster is. Since both of these parameters are fixed, it is reasonable to expect that this algorithm will classify more data samples as noise the higher this threshold density is set to (by employing low values of ϵ or large values of $MinPts$), and will label more data samples within the same cluster as this threshold density is lowered.

On the other hand, OPTICS (Ordering Points To Identify the Clustering Structure) can be considered as an extension of DBSCAN, which does not take into account a single density threshold but rather examines the clustering structure of the dataset for a base ϵ and a fixed $MinPts$ [15]. This clustering structure is expressed by means of a particular sorting order of all data samples as well as the reachability and core distances of each of these data samples. In the current study, this expression of the clustering structure of the dataset is employed to determine a valid pair of ϵ and $MinPts$ with which to carry out the DBSCAN clustering analysis of each two-dimensional domain. Nevertheless, it is worth noting that is also possible to directly group the dataset into clusters by making use of this presentation of the dataset clustering structure.

In order to fully understand the basis on which OPTICS functions, it is important to introduce the concept of the core distance of a data sample. This distance, based on a particular number of points $MinPts$, defines the minimum neighborhood radius ϵ at which such data sample becomes a core object. More formally, the definition is as follows [15]:

Definition 2.2.5. Core Distance of a Data Sample p: Given an object p belonging to a dataset D, a distance ϵ, the ϵ-neighborhood of p $N_\epsilon(p)$, a number of points $MinPts$, and the distance to its n-th neighbor $d_{nth}(p, n, \epsilon)$, the core distance of p follows:

$$d_{core}(p, MinPts, \epsilon) = \begin{cases} undefined & Card(N_\epsilon(p)) < MinPts \\ d_{nth}(p, MinPts, \epsilon) & otherwise \end{cases}$$

Given that the definition of this distance takes into account whether the number of data samples in the ϵ-neighborhood of p is superior to $MinPts$, it is possible to add that the core distance of an object p with respect to ϵ and $MinPts$ will only be defined once p is a core object with respect to ϵ and $MinPts$. If this is fulfilled, then the core distance is actually the smallest neighborhood radius ϵ' which would allow for p to be a core object with respect to ϵ' and the specified $MinPts$. As a result, the core distance will always be smaller than the specified ϵ with which the number of data samples in $N_\epsilon(p)$ is examined, since otherwise it would imply that the object under analysis would require a larger neighborhood radius than ϵ to have $MinPts$ neighbors and as a result would not be a core object with respect to ϵ. These characteristics of the core distance are made visible in Figure 2.6, where $MinPts$ is set to 4.

Furthermore, it is necessary to describe the concept of the reachability distance in order to understand how OPTICS assigns data samples to clusters.

Definition 2.2.6. Reachability Distance of a Data Sample p with respect to o: Given objects p and o belonging to a dataset D, a distance ϵ, the ϵ-neighborhood of o $N_\epsilon(o)$, a number of points $MinPts$, and the distance between o and p $d(o, p)$, the reachability distance of p with respect to o is defined as follows:

$$d_{reachability}(p, o, MinPts, \epsilon) = \begin{cases} undefined & Card(N_\epsilon(o)) < MinPts \\ \max(d_{core}(o, MinPts, \epsilon), d(o, p)) & otherwise \end{cases}$$

In few words, the reachability distance of a data sample p with respect to o is the smallest neighborhood radius ϵ' with which p is directly density-reachable from o with respect to ϵ' and

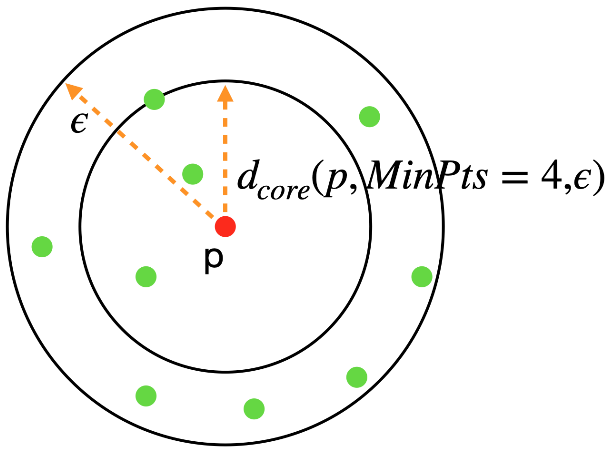

Figure 2.6: Core Distance of a Core Object with respect to ϵ and $MinPts = 4$

$MinPts$ at the same time allowing o to be a core object with respect to ϵ' and $MinPts$ [15]. Similarly to the definition of the core distance, this expression for the reachability distance requires o to be a core object with respect to ϵ and $MinPts$. However, once this condition is fulfilled, the reachability distance is the maximum between the core distance of o with respect to ϵ and $MinPts$ and the distance between samples o and p. The different cases for this definition, depending if p is outside or inside of the radius defined by the core distance of o are represented in Figure 2.7. As p_1 is closer to o than the $MinPts$-th neighbor of o, it is necessary for the neighborhood radius which the reachability distance defines to still include $MinPts$ samples, so that o is a core object with respect to such neighborhood radius and $MinPts$, which in this case is 4. Moreover, as p_2 is father from o than its $MinPts$-th neighbor, it is necessary to enlarge this neighborhood radius to allow for p_2 to be directly density-reachable from o. In any case, it is important to take into account that the reachability distance of a sample p depends directly on the core object o with respect to which it is defined.

Based on these definitions, the OPTICS algorithm fundamentally determines the clustering structure of the dataset by analyzing each data sample within the dataset and storing both its core distance as well as its reachability distance with respect to the closest core sample from which it is directly density reachable [15]. In order to do so, it requires the introduction of a fixed $MinPts$ and a base ϵ, although it does not directly employ them for cluster assignment as DBSCAN does.

To be more precise, for every unprocessed data point in the dataset, OPTICS retrieves its ϵ-neighborhood and determines its core distance with respect to ϵ and $MinPts$. If this object has an undefined core distance because it is not a core sample, the method goes on to the next unprocessed point in the dataset. If this is not the case as the data sample is a core object, the algorithm collects all directly density-reachable points from this point with respect to ϵ and $MinPts$ and stores these so-called seed points in a seed list, sorted by their reachability distance to the closest core object to which they have already been determined as directly density-

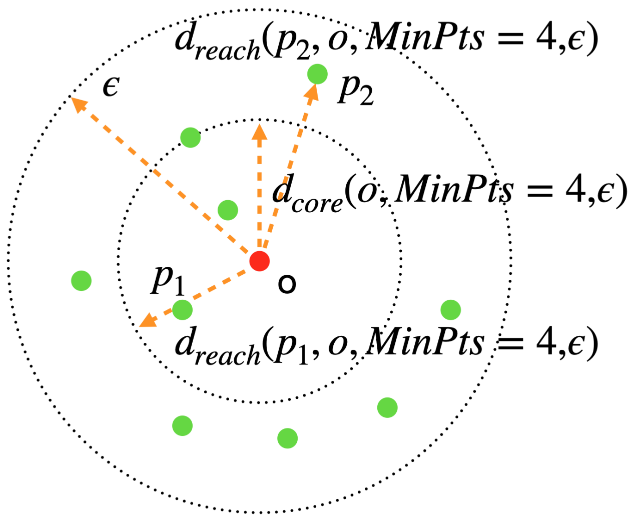
$d_{reach}(p_2, o, MinPts = 4, \epsilon)$
ϵ
p_2
$d_{core}(o, MinPts = 4, \epsilon)$
p_1
o
$d_{reach}(p_1, o, MinPts = 4, \epsilon)$

Figure 2.7: Reachability Distance of a Core Object with respect to ϵ and $MinPts = 4$

reachable. In other words, only if the new reachability of an already processed seed point is lower than the previously calculated one, this seed point ascends in the seed list accordingly. Then, OPTICS proceeds to process all of these stored seed points in the order in which they have been stored, by calculating their core distance and analyzing their ϵ-neighborhood. For each of these processed seed points, their reachability and core distances are stored and their neighboring samples are stored into the same seed list for further processing. Once no more seed samples are generated, the algorithm goes on to analyze another unprocessed sample in the dataset. As a result, the core and reachability distances of each of the samples in the dataset are determined. Moreover, the samples will be stored in the order in which they have been processed, such that the reachability distance that is stored for a sample will be determined with respect to a close predecessor in the ordered results.

As has been mentioned, a main utility of OPTICS is to analyze the clustering structure of the dataset as a whole by taking into account the ordering of this dataset which this algorithm generates as well as the set of reachability distances associated to each data sample. This is visually possible by means of a reachability plot displaying the reachability distances of the ordered data samples within the dataset. Take for instance the three simple clusters in Figure 2.8, generated by three distinct Normal distributions. Based on this image, it is easy to expect any adequate clustering procedure to assign three different clusters, one for each of the Normal distributions employed.

Once a reachability plot representing an overview of the clustering structure is computed, as presented in Figure 2.9, it is possible to assign each cluster to a dent within the reachability plot. Note that the coloring of this reachability plot is not assigned based on cluster labels assigned by OPTICS, but rather based on to which Normal distribution the data sample associated to such reachability distance belongs.

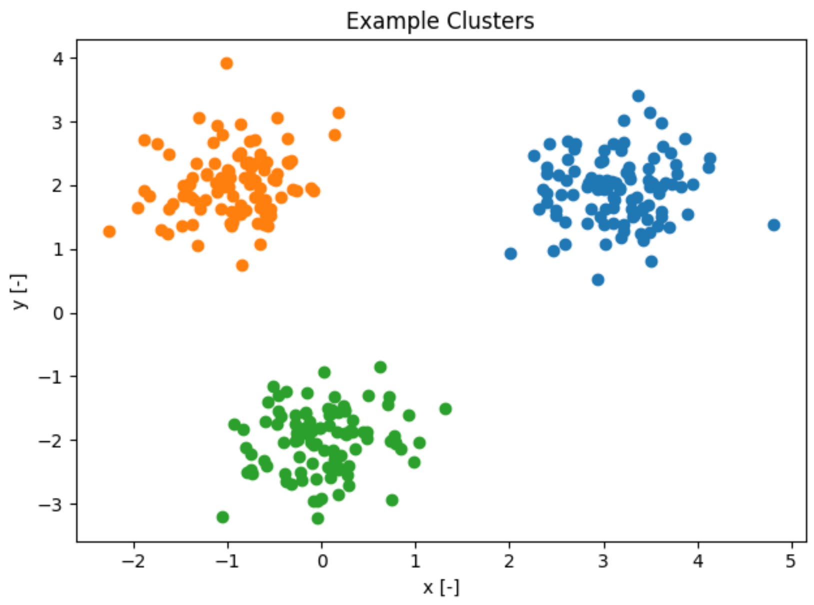

Figure 2.8: Three Artificially-Generated Simple Clusters

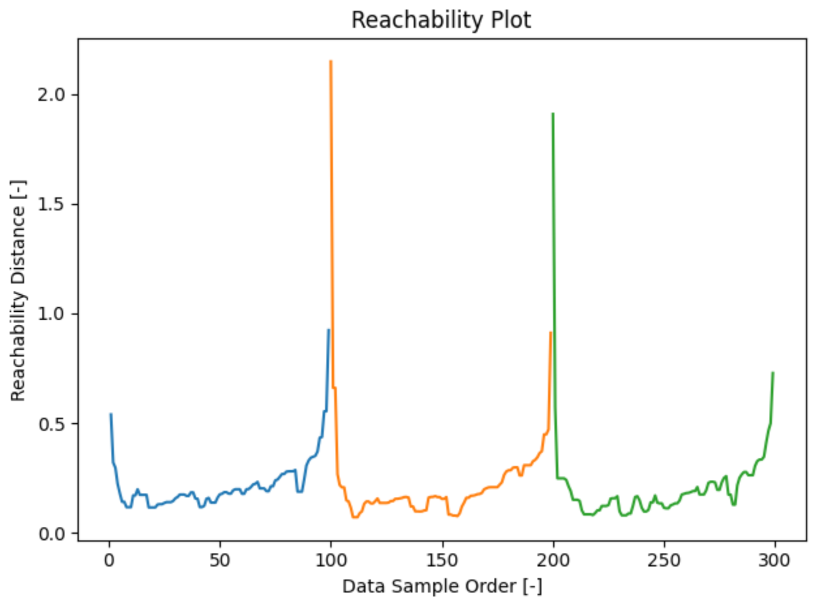

Figure 2.9: Reachability Plot of the Artificially-Generated Simple Clusters

It is necessary to understand that most clusters will have a typical shape associated to them in the reachability plot. For instance, the first reachability value of a cluster will be relatively large, since this first data sample is far away from any other sample which has previously been processed by the algorithm. If this initial reachability of the cluster were smaller, it might be the case that this would not be the beginning of a new cluster but rather the continuation of a previously defined one. Furthermore, as the algorithm proceeds to analyze all neighboring seed points at the same time sorting them based on their reachability distance with respect to the data sample from which they have been most recently directy density-reachable, the reachability distances of the points belonging to this cluster are smaller than the first reachability of this cluster, but also tend to increase due to the way in which these seed points are sorted. Thus, the last reachability distance associated to the cluster is relatively large, since this data sample has been placed last precisely due to its more significant reachability distance. To the skeptical eye, the smaller dents which appear within the greater cluster dent may contradict the general trend associated to the way the seed list is sorted. Nevertheless, one has to take into account that it may occur that an unprocessed cluster sample which is however stored in the seed list with a reachability distance with respect to an already processed sample in the cluster may have a lower reachability distance when computed with respect to the data point that is currently being processed. Then, this unprocessed sample will be placed at a higher position within the seed list, in some cases to the first position due to an abruptly lower reachability, thus reasoning these smaller dents within the reachability plot. Lastly, it is also necessary to remark that the reachability of the first data sample of the dataset to be processed is generally set to be undefined or infinite, since no other samples have been processed before.

Up to now, two density-based clustering methods have been presented. Being the simpler one of the two, DBSCAN requires the specification of a density threshold by means of a minimum number of samples $MinPts$ and a neighborhood radius ϵ, in order to group data samples into clusters. In this case, the number of parameters to be specified is the same as those of any box-counting methods of particle cluster identification, in order to set a fixed density threshold. On the other hand, OPTICS allows for a representation of the clustering structure of the dataset in terms of a reachability plot, for which a base ϵ (usually set to infinity) and a number of samples $MinPts$ have to be specified. It is worth noting that both of these methods require the introduction of user-defined parameters, where as the previously introduced methods of particle cluster identification based on Voronoï tessellations require none. Given that OPTICS is observed to require around 60% more time than DBSCAN to perform the same clustering task [15], it thus makes sense to employ OPTICS to obtain an overview of the clustering structure of the problem and thus select the adequate set of parameters to perform a DBSCAN clustering analysis aptly.

In the current application, these two algorithms are especially useful due to their capacity of working with clusters of arbitrary shapes, due to the fact that they only focus on data samples with a relatively higher data density. This, however, is associated to possible ”single link” effects, in which two clusters which could visually be considered as distinct are connected by a single line of data samples allowing them to be assigned to the same cluster by the algorithms presented here. These effects can be avoided by employing higher values of $MinPts$, this reducing the extent to which thin links of particles can communicate clusters [15].

Moreover, another useful peculiarity of these two clustering techniques is the fact that certain data samples within the dataset are labeled as noise if their data density is lower than the specified threshold. Thus, it is possible to, given a very populated data space, isolate data samples of relatively denser regions, and discard those data samples which are labeled as noise. In other words, it is not necessary to associate every particle within the dataset to a cluster, or equivalently, to generate non-representative clusters in regions of the dataset which are not as densely populated.

Lastly, it is especially convenient that both clustering methods are capable of autonomously deciding how many clusters should be assigned to the dataset, without any direct input from the user. It is reasonable to infer that the resulting number of clusters assigned to the dataset depends exclusively on how high the threshold density set by the user is, as the phenomena connecting different clusters will be largely affected by these parameters. In any case, it is possible to operate without requiring a direct estimation of the number of clusters within a dataset.

Taking into account the nature of these two algorithms, the current study employs OPTICS in order to determine a valid pair of $MinPts$ and ϵ to employ in a DBSCAN clustering analysis of the two-dimensional array of particle positions. More specifically, certain trends and peculiarities of the reachability plot of the position dataset are extracted in order to determine values for $MinPts$ and ϵ which are adapted with respect to the average density of particles in the two-dimensional computational plane which has been extracted. For instance, if the thickness of the sheet based on which the particles to be projected to a particular computational plane is varied, it is expected for the resulting two-dimensional set of particle positions to include similar shapes defined with a different average density. If this thickness is increased, more particles will be normally projected onto the computational plane, and although the shape of the presented structures will vary slightly, the density with which these structures are defined will be significantly greater. The opposite follows if the thickness of this sheet is reduced. With this in mind, it is necessary to employ the overview of the clustering structure which OPTICS allows for in order to determine a set of values for $MinPts$ and ϵ which suits the average density of the two-dimensional domain containing projected positions. This determination is presented in Sections 2.2.3 and 2.2.4. Then, based on these parameters, it follows to simply apply DBSCAN to properly group the computational plane into clusters and noise in a fast manner.

2.2.3 Towards Adapted Hyperparameters: Minimum Number of Points within the Cluster

Now that the two density-based clustering algorithms employed in this study have been thoroughly presented, it is necessary to more precisely describe how each of them is used. Given a single two-dimensional snapshot of particle positions for a single Z, these clustering techniques have to be applied in order to determine which particles belong to a relatively denser region to be denominated as a particle cluster, and which particles do not appear in a sufficiently dense region of the two-dimensional domain, and are thus to be labeled as noise by the clustering algorithm or within the context of this study, as belonging to a void.

As has been explained, a single combination of ϵ and $MinPts$ establishes a number density threshold with which DBSCAN groups data samples into clusters. Therefore, in order to obtain similar results with datasets of different average number density, a process by which these two parameters automatically adapt to the average number density of the dataset is necessary. This section deals with the determination of a value of $MinPts$ which is adapted to the number density of each two-dimensional domain to which DBSCAN is applied.

With the general overview of the clustering structure of the dataset which OPTICS allows for, a connection between the optimum parameter choice to be employed in the clustering routine and the average number density of the dataset can be obtained. As this overview of the clustering structure is mainly expressed in the reachability plot of the dataset, it is possible to extract a set of $MinPts$ and ϵ adapted to the peculiarities of the dataset based on it. For the purposes of the following explanation, a reduced dataset is used, where the spatial domain and thus the number of particles are decreased for greater ease of analysis. The particle positions exposed in Figure 2.10 correspond to a single snapshot in Z, whose domain is summarized in Table 2.3.

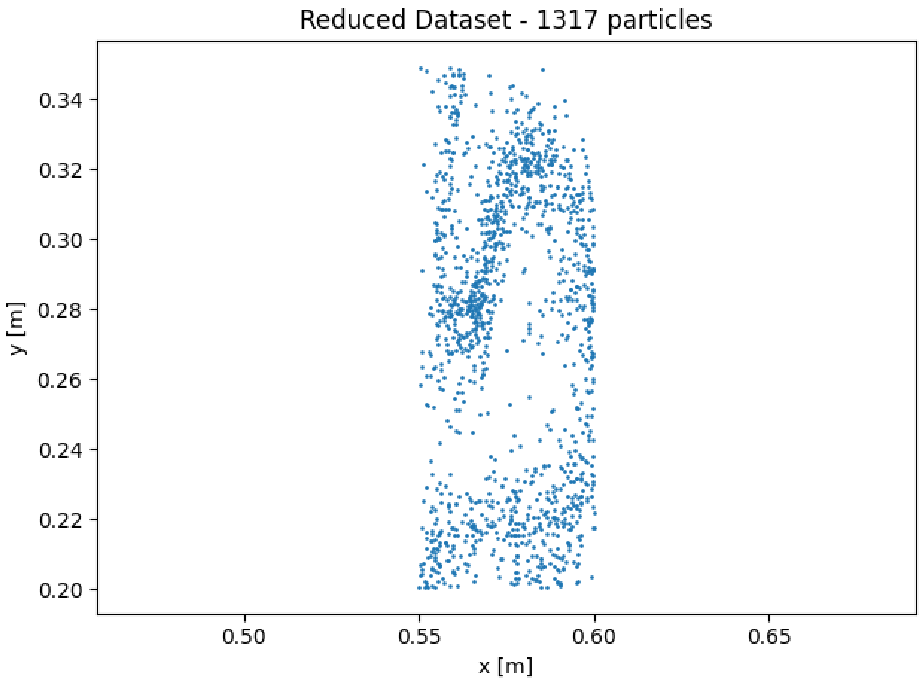

Figure 2.10: Particle Positions of Reduced Dataset

Based on this simple dataset, it is possible to analize how the reachability plot varies once the the minimum number of points within a cluster $MinPts$ is varied. In the first place,

Z [m]	0.02
Thickness of Sheet [m]	0.00084
Planar Range in X [m]	[0.1485, 0.162]
Planar Range in Y [m]	[0.08, 0.14]

Table 2.3: Spatial Domain of Reduced Dataset

a couple of relevant definitions must be recalled. On the one hand, the core distance of a data sample is the neighborhood radius that such sample requires in order to include $MinPts$ neighbors within this neighborhood. On the other hand, the reachability distance of a data sample p with respect to an object o includes the maximum between the distance between both samples and the core distance of o. Now, by observing Figure 2.11, it is clear that the shape of the reachability curve does vary significantly with $MinPts$. In general terms, it can be said the curve is smoothed out and increased in value. If the previous definitions are taken into account, it makes sense that the curve rises in value as $MinPts$ grows, since this means that the core distance of the data sample with respect to which the reachability is computed also increases.

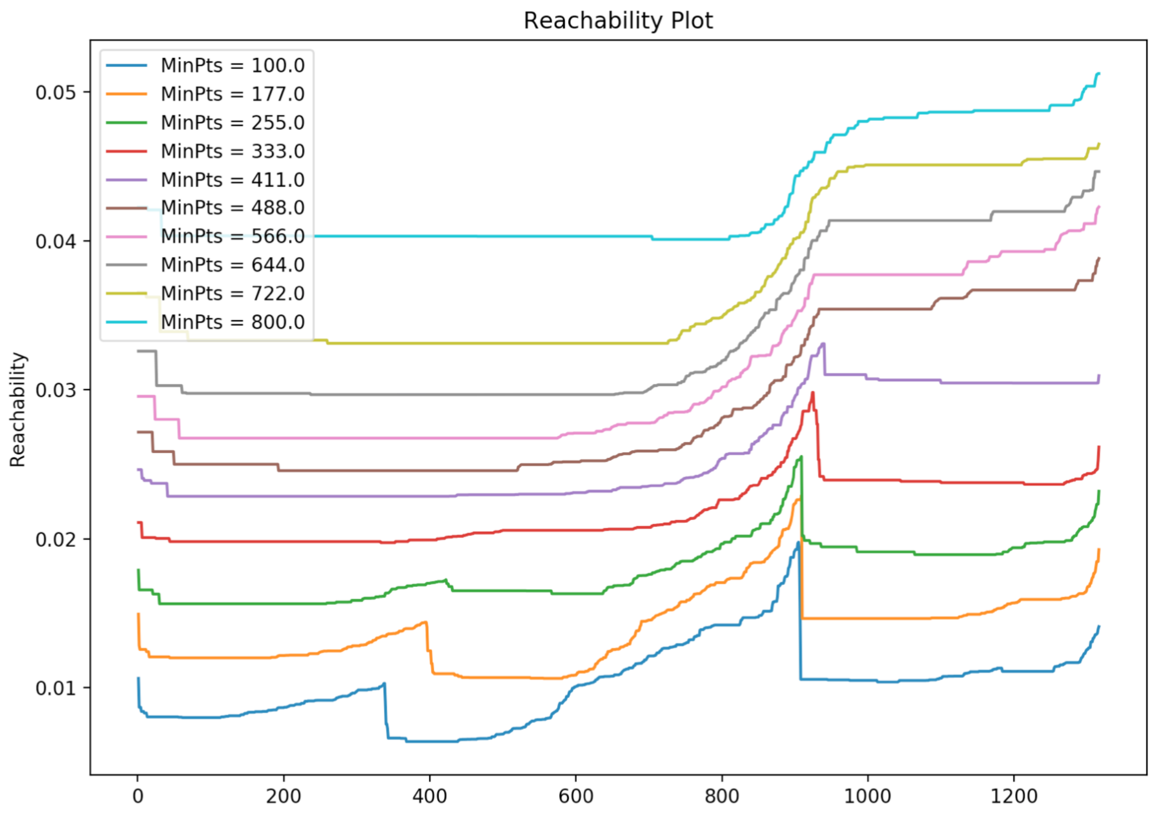

Figure 2.11: Reachability Plots for Different $MinPts$ of the Reduced Dataset - Note that several OPTICS analyses have been carried out for the same reduced dataset, all with a base ϵ of infinity.

This same phenomenon explains the fact that the reachability curve of the reduced dataset is smoothed out with an increasing value of $MinPts$. As has been explained previously, the sudden drop in reachability at the beginning of a cluster dent is caused because the first particle within the cluster is very far away from the previously processed data samples, but the second particle within the cluster is much closer to this first particle, thus causing this immediate fall in the reachability plot. In fact, it is clear due to the flat shape corresponding to the first particles after this drop in reachability that these reachability values correspond to the core distance of the first particles in the cluster. As the particles in the aforementioned sorted seed list are read, the reachability values commence to represent instead the distance between the current particle and the closest core object particle, in this way explaining the continuous growth in the curve. This occurs until a sudden drop occurs again. In most cases, it can be said that this drop in the reachability with respect to the closest core object occurs because such measure passes from being the distance to a core object of the previous cluster to being the core distance of another core object in the current cluster.

Taking this dynamic into account, as well as the fact that the core distance of every particle is to increase once $MinPts$ grows, it is also reasonable to expect for these aforementioned drops in reachability to be less significant, since the the distance to a core object in the previous cluster will not be as large when compared to the core distance of a core object in the current cluster, the latter having increased in value with $MinPts$. In fact, given a sufficiently large $MinPts$, these sudden drops disappear altogether, since the core distance of the core object of the current cluster is not anymore larger than the distance to a core object in the previous cluster.

This trend is exaggerated once one analyzes the plot of reachability distances normalized with the minimum reachability distance resulting of the OPTICS analysis of a particular $MinPts$ value. In Figure 2.12, it possible to observe how the spikes in reachability are smoothed out once $MinPts$ is increased, such that the variation with respect to the minimum reachability of the OPTICS analysis is smaller.

One can also infer that as $MinPts$ increases, there is a sort of convergence in the number of dents, or concave regions $N_{concave}$, within the reachability plot. A single dent or concave region can be formally defined as a region of the reachability plot in which the curve decreases from a previous local maximum and then ascends after such a descent. In Figure 2.12, it is possible to observe how smaller dents in the plot start to disappear once $MinPts$ is increased, such that the number of concave regions in the curves tends to a particular value. This converged number of concave regions $N_{concave}$ can be seen to describe an innate clustering structure in the dataset which does not vary with an increase of $MinPts$. In other words, if one were to employ the steepness of the reachability curve in order to determine the clusters in the dataset, as is exposed in [15], one would obtain the same number of clusters for increasing values of $MinPts$ once this reachability curve is considered to converge. Although the shape and size of these clusters would vary modestly, it is reasonable to expect the location of these clusters to correspond to the same regions of the data space as $MinPts$ varies.

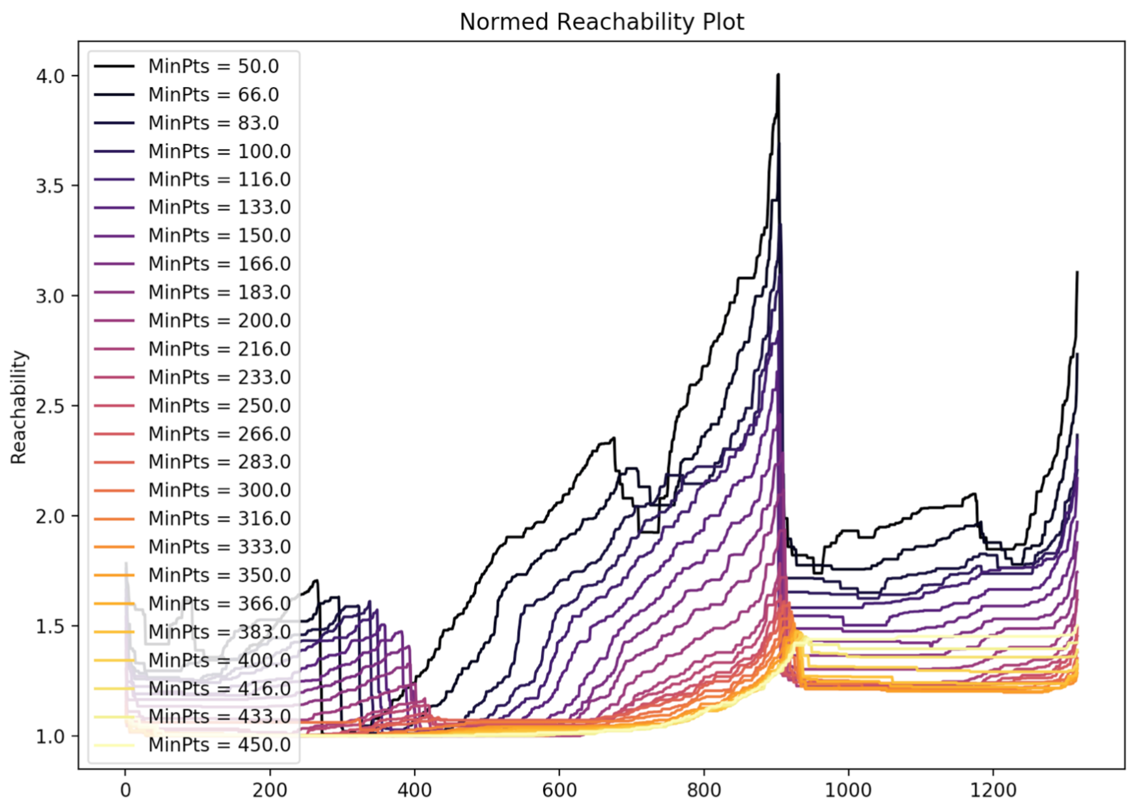

Figure 2.12: Normalized Reachability Plots for Different $MinPts$ of the Reduced Dataset

Consequentially, if one seeks to determine a value of $MinPts$ which considers the innate clustering structure of the dataset, a structure which is not expected to vary significantly with variations in the average number density resulting from modifications in the thickness of the computational sheet, it makes sense to employ the value for $MinPts$ at which the number of concave regions in the reachability curve $N_{concave}$ is seen to converge.

This last condition is formalized in the implemented code by analyzing the derivative of the number of concave regions $N_{concave}$ with $MinPts$, as is expressed in Equation (2.4), where $MinPts^*$ corresponds to this optimum number of minimum points within a cluster and χ expresses the severity of this criterion.

$$\left.\frac{\partial N_{concave}}{\partial MinPts}\right|_{MinPts^*} \leq \chi \tag{2.4}$$

Bearing this criterion in mind, for the reduced dataset employed in this section, this convergence is displayed in Figure 2.13, where χ is set to 0.1. This choice of χ is reasoned by the fact that $N_{concave}$ close to the converged state of the reachability plot adopts small, discrete values, such that variations of $N_{concave}$ below 10% are rare. Nevertheless, a series of considerations have to be taken into account with regards to the computational implementation of this criterion. On the one hand, this derivative is computed numerically by means of the command *gradient* which the Python library *numpy* provides. On the other hand, this criterion has to be fulfilled for a given number of consecutive values of $MinPts$, in order for $MinPts^*$ to be extracted. In short, in order to obtain $MinPts^*$, it is necessary to obtain the reachability plot of a single two-dimensional domain containing projected particle positions for several values of $MinPts$, and after counting the number of concave-up regions in each reachability plot, observe for which $MinPts^*$ is Equation (2.4) fulfilled.

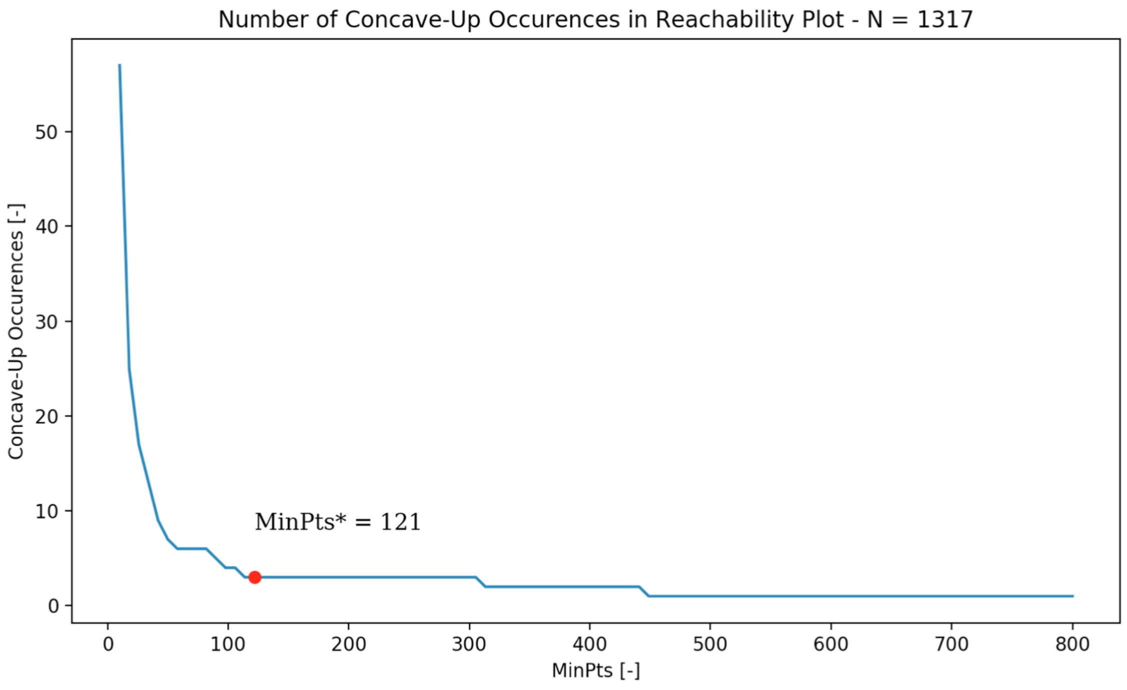

Figure 2.13: Determination of $MinPts^*$ for the Reduced Dataset by Computing $N_{concave}$ for different $MinPts$ - 1317 particles in domain

With the objective of analyzing how this optimum $MinPts^*$ varies with the thickness of the computational sheet based on which the dataset is generated, the similar procedure has been carried out with different thicknesses but the same planar range in X and Y. Therefore, in Figure 2.14, the sheet thickness is increased from 0.00084 m to 0.00164 m, thus bringing about an increase in the average number density of the computational plane, from $1,625,925$ $1/m^2$ to $3,219,753$ $1/m^2$. As a result, the optimum minimum number of points defining a cluster is increased, from 121 to 203. On the other hand, Figure 2.15 displays the optimization of $MinPts$ with a sheet thickness of 0.00052 m, where the average number density is decreased to $1,002,469$ $1/m^2$ and with it the value of $MinPts^*$.

Outside of the reduced dataset, one can perform this same procedure for a two-dimensional

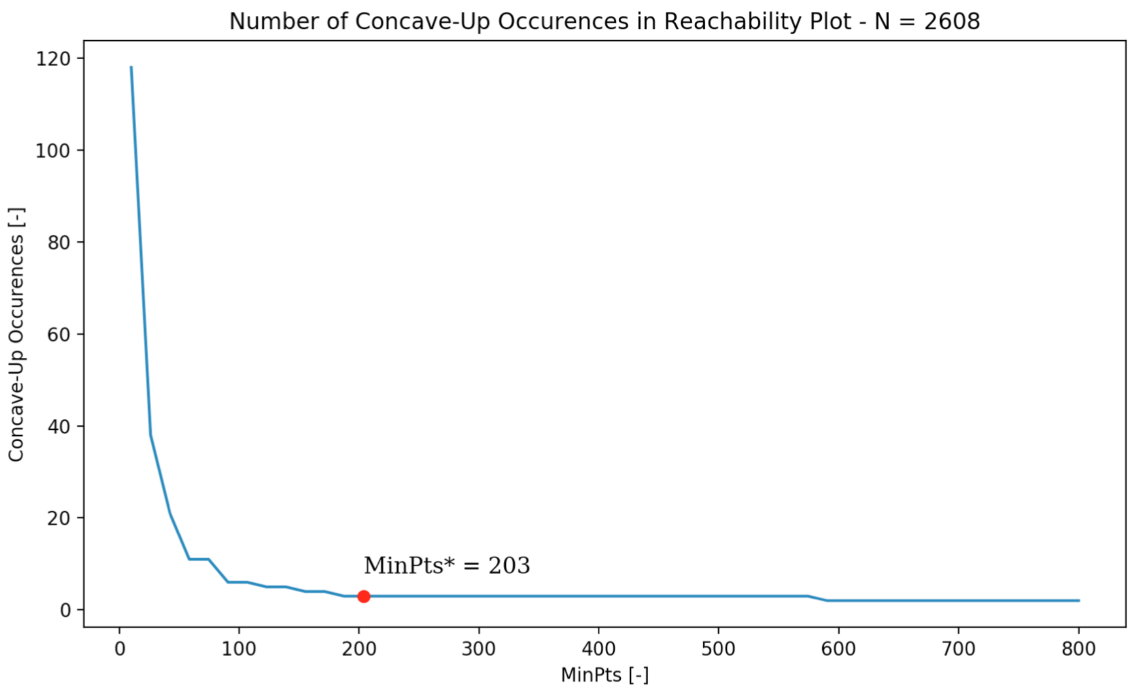

Figure 2.14: Determination of $MinPts^*$ for the Reduced Dataset by Computing $N_{concave}$ for different $MinPts$ - 2608 particles in domain with Sheet Thickness of 0.00164 m

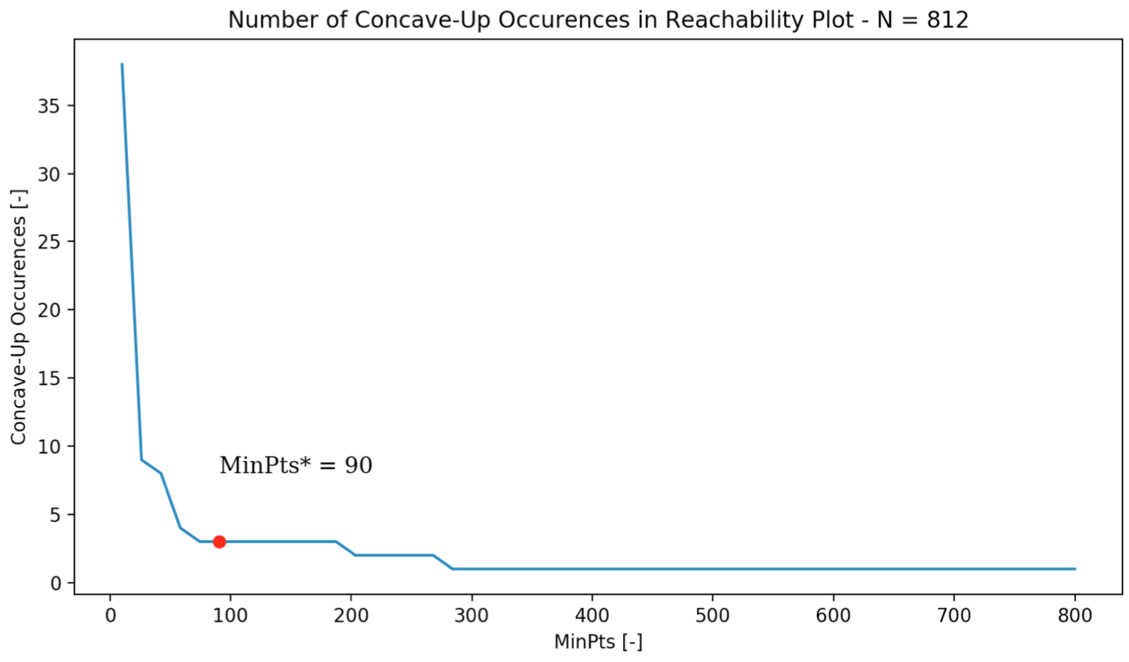

Figure 2.15: Determination of $MinPts^*$ for the Reduced Dataset by Computing $N_{concave}$ for different $MinPts$ - 812 particles in domain with Sheet Thickness of 0.00052 m

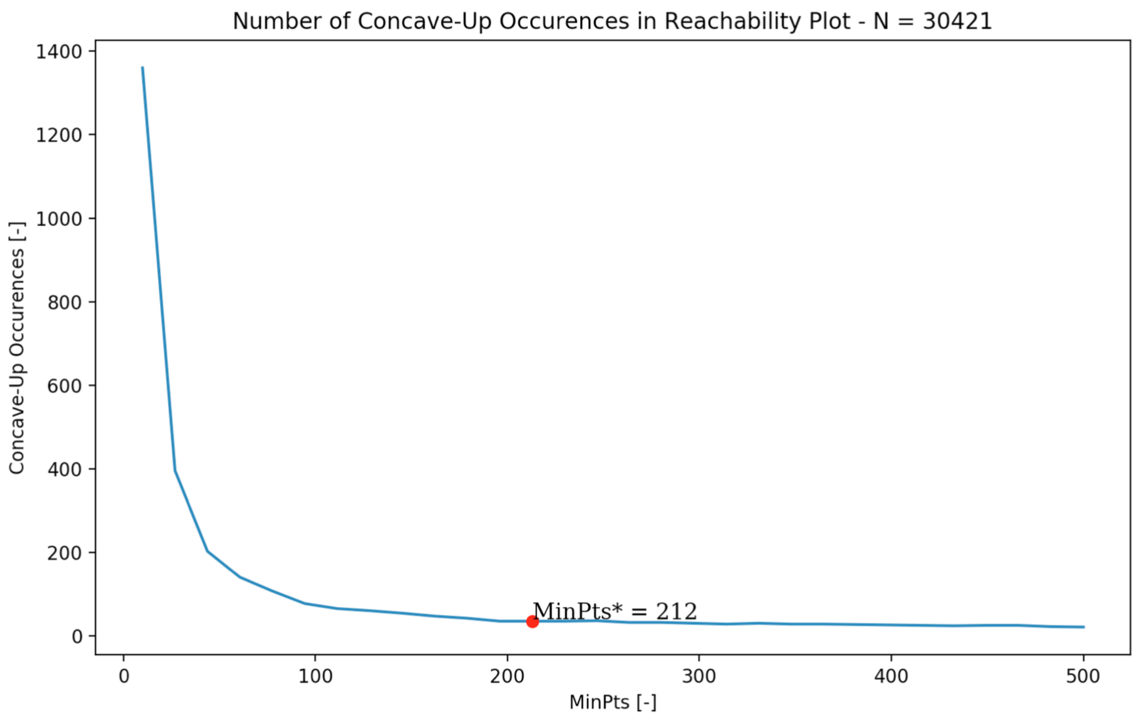

Figure 2.16: Determination of $MinPts^*$ for the Original Dataset - 30421 particles in domain

snapshot in Z of the original dataset, and obtain a parameter based on which a clustering routine can be performed without an influence of the average number density of the dataset. The results of performing this optimization of $MinPts$ in a single snapshot in Z of thickness 0.0004 of the original dataset are displayed in Figure 2.16.

What results from this optimization is a value for $MinPts$ which is adapted to the number density of the employed dataset, which within the context of this study is directly related to the sheet thickness with which the three-dimensional particle positions are projected into a set of parallel planes. With this $MinPts^*$ one obtains a particular reachability plot which is thought to represent the clustering structure of the dataset in a way which will not vary significantly once $MinPts$ is increased. Thus, a straight-forward DBSCAN clustering analysis can be applied to the dataset as soon as the remaining parameter defining the minimum data density of a cluster is obtained in a way which is also adapted to the particularities of the given dataset. In this way, it is necessary to develop a method of determining an adapted value for ϵ.

2.2.4 Towards Adapted Hyperparameters: ϵ

Once that an optimum value of $MinPts$ has been determined in a way which is properly adapted to the average density of the dataset under analysis, it is necessary to obtain a value for ϵ, the other parameter required in order to define a minimum cluster density, which is also capable of interpreting the dataset's idiosyncrasies. It is worth noting that once $MinPts$ is fixed, the reachability plot representing the clustering structure of the dataset is also fixed, and thus studies of its variation are not anymore possible.

In order to determine an optimum value of ϵ for the clustering analysis which ensues, it is necessary to obtain a length scale which takes into account the average number density associated with the dataset. If the average number density of the dataset was very high, it would be necessary for the density threshold based on which a cluster is defined to be elevated as well, such that the regions of the dataset belonging to a cluster are not unnecessarily enlarged with respect to the case in which the average number density of the dataset is lower. The way to adapt this density threshold to an increased average number density is, for a known $MinPts$, to employ a lower neighborhood radius ϵ. On the other hand, if the dataset consisted of more sparsely populated data samples, it would make sense to relax the minimum density requirements for a cluster, in order to ensure that clusters still appear where they appear with greater average number densities of the dataset. In this case, for a given $MinPts$, it follows to increase ϵ, thus reducing the density threshold of a cluster.

In order to connect the average number density of particles within a single snapshot in Z with a particular length scale for ϵ, the current study has generated a random distribution of particles on the planar domain. This random distribution is easily created once one imposes a uniform probability along the whole domain. Moreover, in order to ensure that this random distribution of particle positions is representative of the dataset under analysis, these particle positions are distributed maintaining the same planar number density as such dataset. In order to preserve the simplicity of analysis which this reduction allowed for, the same reduced dataset as the one presented in Section 2.2.3 is employed in order to study a method determining an optimum value for ϵ. Consequently, Figure 2.17 presents the random distribution of particle positions resulting from randomly scattering the same number of particles as in the reduced dataset throughout the same planar domain.

Once this random distribution of particles has been generated, one has properly expressed the average nature of the dataset's number density in a new dataset of particle positions. From here, it makes sense to extract a reference length scale which takes into account the ability of adjacent or neighboring data samples of being directly density-reachable with respect to each other, for the previously specified $MinPts$. Therefore, the mean reachability of the randomly distributed set of particle positions is extracted, accurately representing the average neighborhood radius ϵ with which adjacent particles are directly density-reachable from each other. In Figure 2.18, it is possible to observe that the reachability plot of the equivalent random distribution of particle positions is close to constant, and has an average value that is

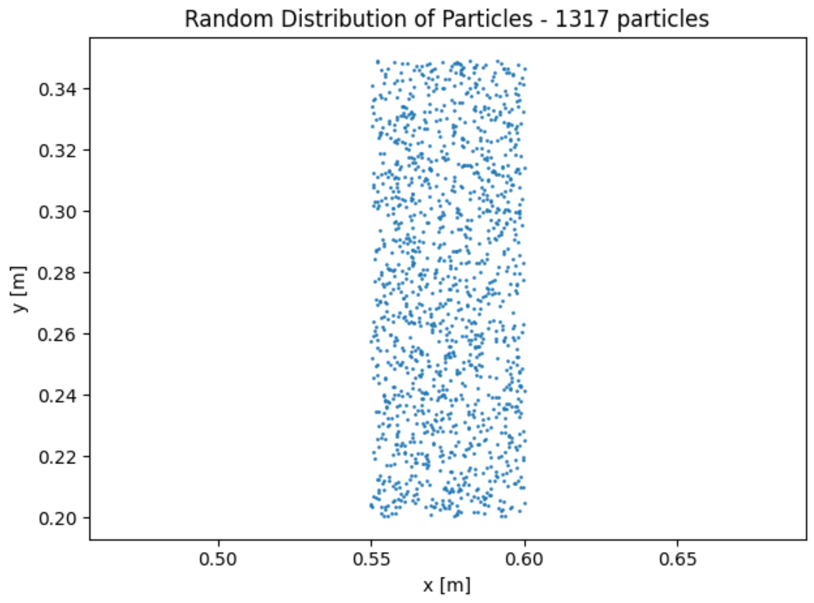

Figure 2.17: Random Distribution of Particle Positions for the Employed Reduced Dataset

within the range of possible rechability distances of the reduced dataset.

In the current study, after close examination, it was determined that an ϵ^* value of 95% of the mean reachability of the randomly distributed equivalent dataset results in an adequate DBSCAN clustering analysis. This choice is reasoned by the fact that if ϵ^* is slightly lower than this mean reachability, in the case of perfectly uniformly distributed particle positions (where the local number density is constant and equal to the planar number density), no particle is associated to a cluster by DBSCAN.

In Figure 2.19, one can see the reachability plot resulting from the OPTICS analysis of a single snapshot in Z or the original dataset compared with the reachability plot of an equivalent random distribution of particle positions. As is visible in this comparision of reachability curves, regions of the domain of lower number density than the analogous randomly distributed case present higher reachability distances.

In this way, an optimum ϵ^* can be determined which takes into account not only the previously obtained $MinPts^*$ but also the inherent connection between the average number density of the dataset and the capability of adjacent particles of being directly-density reachable with respect to each other, expressed by means of a randomly distributed set of particle positions. Now that the two parameters defining the minimum cluster density can be obtained in a manner that is adapted to the average number density of the dataset under analysis, one can perform a simple DBSCAN clustering analysis for each of the snapshots in Z in the datasets, given that the average number density of all of these two-dimensional domains is

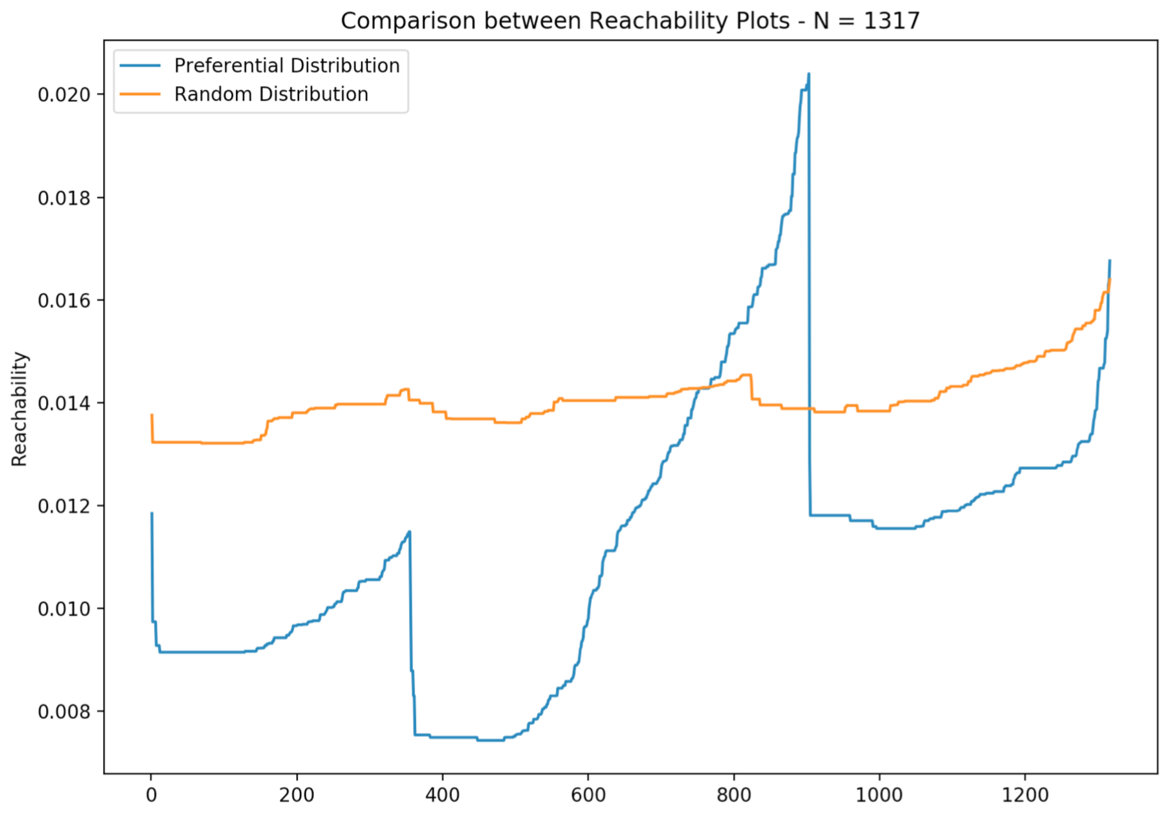

Figure 2.18: Reachability Plots of the Reduced Dataset and of its Equivalent Random Distribution - 1317 particles in domain

the same. In other words, if the thickness of the sheets defining which particle positions are projected normally to which plane of constant Z is the same for all the sheets, the parameters obtained by means of these optimizations are applicable to all of the two-dimensional subsets of positions within the dataset.

2.2.5 Results

Once that these two optimizations for $MinPts^*$ and ϵ^* have been carried out, what remains is to perform the DBSCAN clustering analysis of each of the two-dimensional snapshots in Z of the dataset. It is worth noting that given the capacity of DBSCAN and OPTICS to classify data samples as noise, it is possible to sort particles within the dataset as belonging to a particle void, given that their number density is not as significant as in those regions of the dataset which the algorithm labels as a cluster. Moreover, DBSCAN is also capable of separating cluster data samples into different cluster groups depending on the connectivity of their member samples. In short, given a set of particle positions within a single Z level,

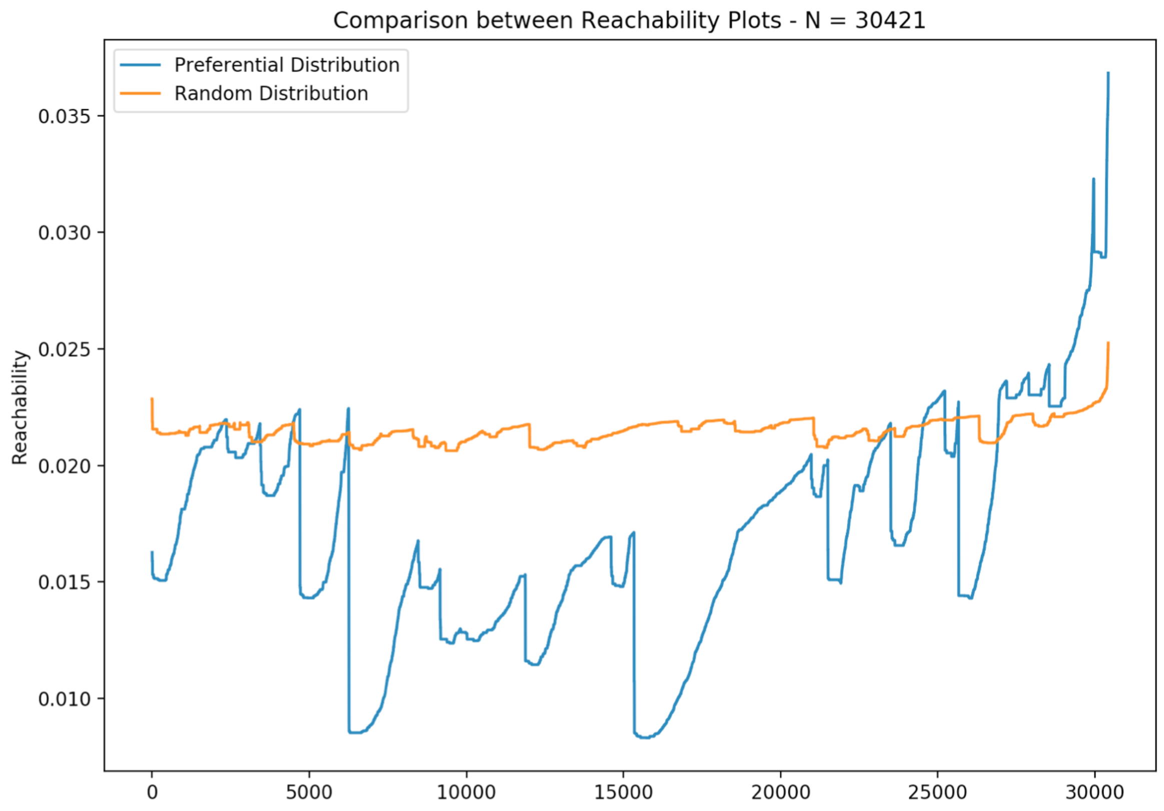

Figure 2.19: Reachability Plots of the Employed Dataset and of its Equivalent Random Distribution - 30421 particles in domain

this application of DBSCAN separates cluster particles from void particles, and also separates cluster particles from particles in different clusters.

It is moreover worth noting that after introducing the optimized set of parameters, it was found that the clustering technique increased in accuracy once non-core objects within clusters defined by DBSCAN are later labeled as noise, such that the resulting clusters are exclusively made up of core objects. For more information with regards to the determination of the implementation's accuracy, see Section 3.2.

With regards to the separation of particles into cluster and void particles, the implemented routine resulted in what is presented in Figures 2.20 and 2.21. While as Figure 2.20 presents the particle positions and classifications for $Z = 0.0184$ m, Figure 2.21 displays the same information for $Z = 0.0192$ m. Note that given the employed separation between adjacent projection sheets, both figures present adjacent snapshots in Z, such that the continuity in the topology of the particle structures present is ensured. As is possible from a qualitative analysis, relatively denser regions of the domain are labeled as clusters, and less densely populated regions are classified as voids. For a detailed validation of this classification, see Section 3.2.

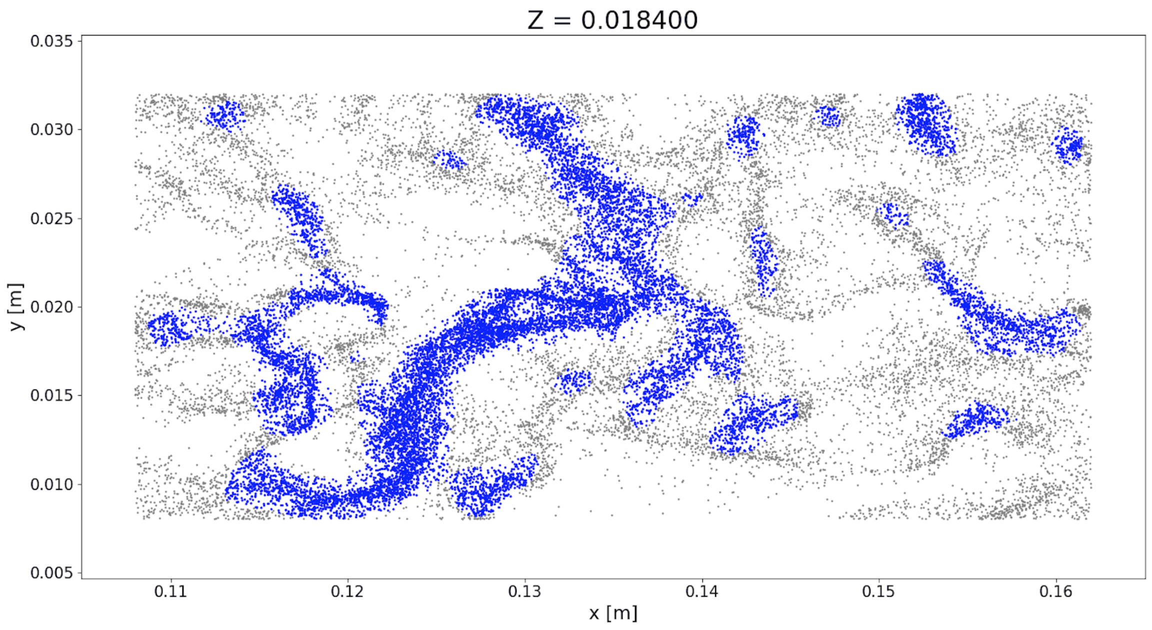

Figure 2.20: Classification of Particles into Cluster and Void Particles for $Z = 0.0184$ m. In blue: cluster particles, in gray: void particles.

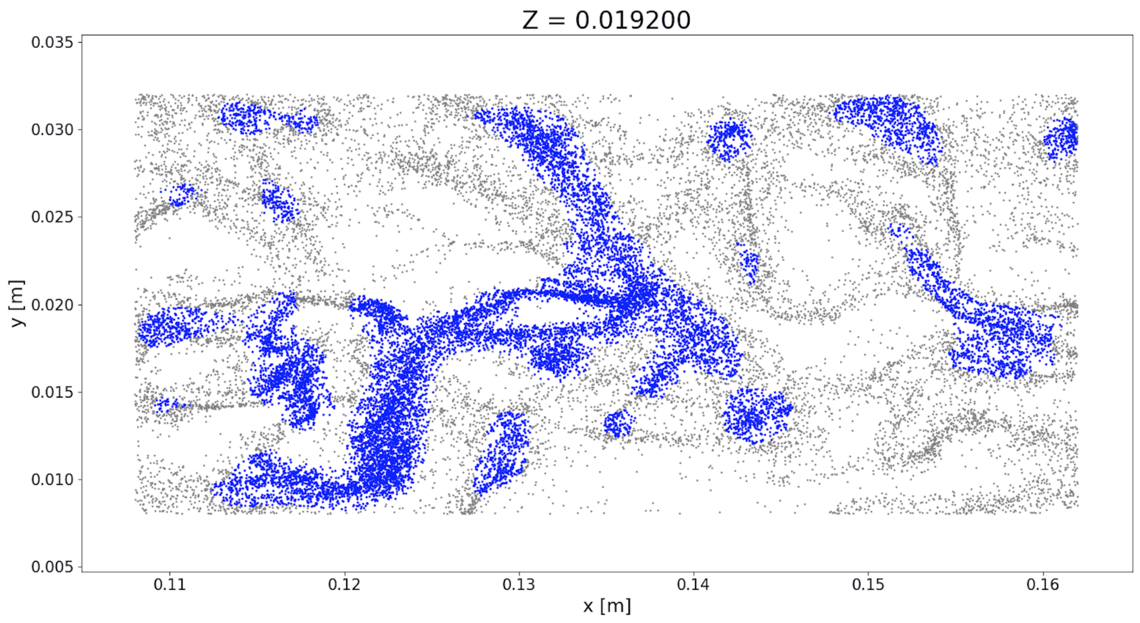

Figure 2.21: Classification of Particles into Cluster and Void Particles for $Z = 0.0192$ m. In blue: cluster particles, in gray: void particles.

Furthermore, the result of applying DBSCAN to separate cluster particles into different clusters is exposed in Figures 2.22 and 2.23, the former referencing $Z = 0.0184$ m and the latter $Z = 0.0192$ m. It is important to bear in mind that the labels which DBSCAN applies within the two-dimensional snapshot in Z do not correspond to the actual particle cluster labels which this study intends to obtain. In order to separate different particle clusters, it is necessary to take into account their three-dimensional nature by examining the connectivity of cluster particles between different two-dimensional planes. On the other hand, the cluster labels which DBSCAN provides correspond exclusively to the connectivity of cluster particles within the same snapshot in Z. In any case, this information is particularily useful in later stages of analysis.

Based on the results presented in Figures 2.22 and 2.23, it is worth discussing why, for the cluster particles associated with a dark green cluster label, visibly separated groups of cluster particles appear. This is because what is presented in these figures is not the clustering arrangement resulting from DBSCAN, but rather the result of excluding all non-core data samples from such a clustering arrangement. Therefore, when these non-core data samples are set aside, previously existing connections between groups of particle positions disappear, leaving visibly separated groups with the same cluster label.

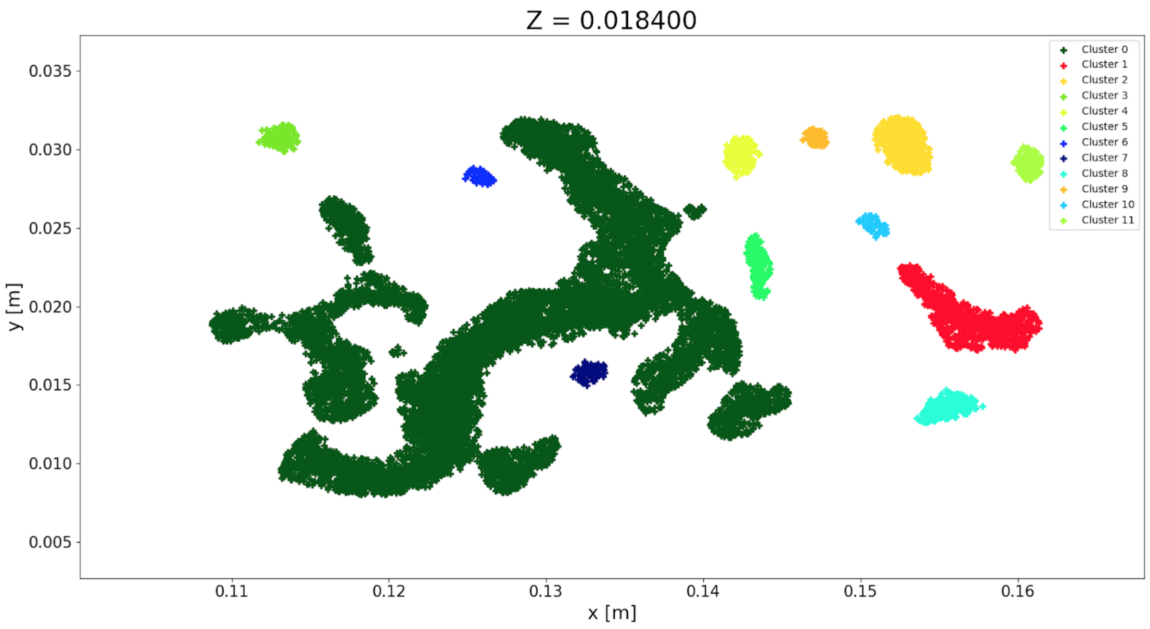

Figure 2.22: Classification of Cluster Particles into Different Clusters for $Z = 0.0184$ m

It is important to note that the current study has employed the implemented OPTICS and DBSCAN commands in the Python library *scikit-learn*, these allowing for a very straightforward use of these clustering procedures.

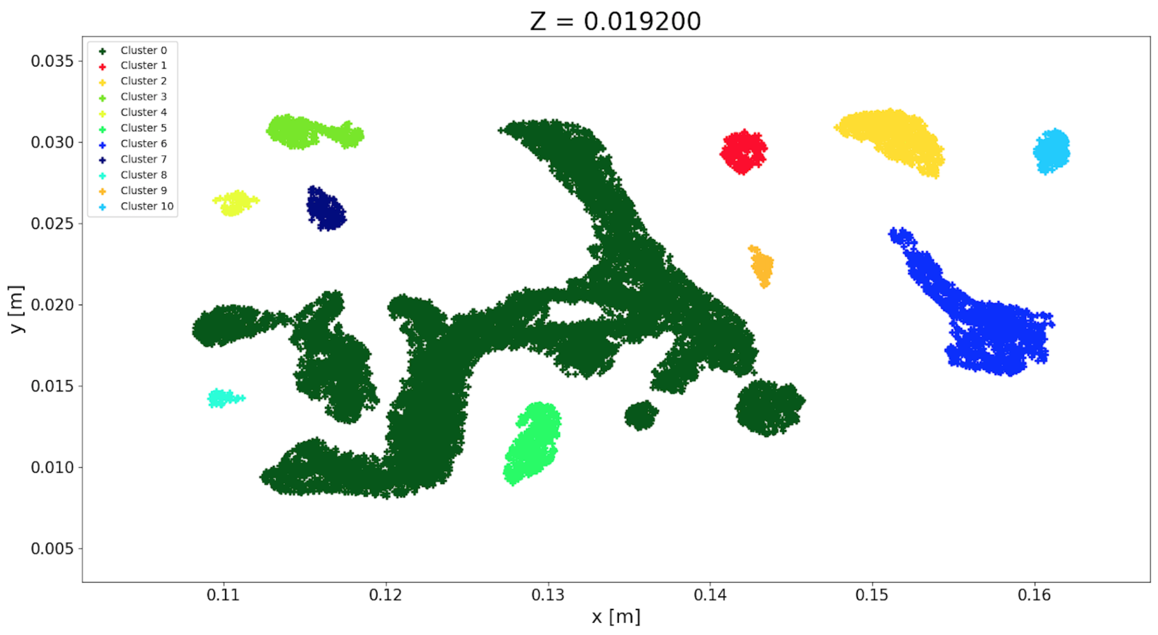

Figure 2.23: Classification of Cluster Particles into Different Clusters for $Z = 0.0192$ m

2.3 Determination of Cluster Boundaries within the 2D Snapshot

2.3.1 What Sort of Cluster Boundary is Desirable?

Further steps of analysis in the current study require the condensation of a three-dimensional cluster topology into a point surface representing its boundary and a set of interior representative points. This condensation parts from the two-dimensional clusters which the implemented application of DBSCAN at each two-dimensional domain results in, by first determining a set of connected points describing each cluster contour. While as for a single snapshot in Z these cluster boundaries will be represented by a set of closed curves, once the connectivity of cluster particles within different Z levels is examined, what will result is that the cluster boundary is described by a cloud of points in three dimensions representing cluster boundary surfaces.

The main utility behind the aforementioned simplification of a three-dimensional topology into a boundary and a set of interior points is the fact that in order to ascertain which two-dimensional clusters defined by DBSCAN describe the same three-dimensional cluster, it is necessary to determine connectivities between such interior points at different two-dimensional domains. These connectivities are traced precisely by examining whether a straight line between interior points collides with a cluster boundary. Therefore, it is reasonable to expect that the characteristics of the generated cluster boundaries will greatly affect the outcome of the later stages of this analysis.

Another argument in favor of this condensation of a three-dimensional cluster is a reduction in computational cost, when compared with the case in which all of the particles indicated by DBSCAN as belonging to a cluster in every two-dimensional domain are processed in order to determine their three-dimensional cluster memberships. However, with this latter method one would obtain not only detailed information with regards to the shape of the cluster, but also particular insight with regards to the concentration field existing within the cluster. Nevertheless, once several DBSCAN clusters within different snapshots in Z are connected to form a three-dimensional cluster according to the connectivity of their cluster particles, the number of points which represent a single three-dimensional cluster turns out to be significantly high.

In any case, it is clear that this procedure must be carried out by capturing as much information as is possible with regards to the original cluster, at the same time incurring the least possible computational cost. These requirements directly affect the design of the implemented method. To begin, a method for determining the cluster's boundary points within a single two-dimensional snapshot in Z is necessary. Moreover, this method should preserve as much of the information regarding the cluster's shape as possible, since the whole objective of this decomposition of the cluster is to maintain the maximum possible amount of detail in describing its topology whilst ensuring computational efficiency.

In this way, if one were to maximize the detail of description of the cluster boundaries, the maximum resolution which one can achieve is defined by the average separation between particles within the cluster. In other words, by defining a boundary curve made up of cluster particles, where adjacent particles in the curve are also adjacent neighbors in the cluster, one obtains the most possibly dense description of the boundary.

On the other hand, different methods exist by which the separation between particles in the cluster boundary is larger than the separation between adjacent particles in the cluster. For instance, if the two-dimensional domain were to be divided into rectangular cells, such that the cluster boundary is to be defined by those cell centers of cells which contain cluster particles and are adjacent to cells which contain no cluster particle, depending on the selected cell size, one could obtain a cluster boundary defined with less resolution than the one allowed for by the whole set of cluster particles itself.

In this way, the ambitious mind would seek to increase the resolution of the boundary as much as possible, later realizing that there is a inconvenience to such ambition. Although within the context of DBSCAN, a cluster is defined by a region of number density higher than a specified threshold, several different number densities may appear within a single cluster. Therefore, if one is to define a cluster boundary based directly on the particles defining the cluster, one will encounter that the spacing between adjacent particles defining the boundary curve varies, such that the cluster boundary in areas of the cluster of high number density is defined by particles closer to one another than in areas of the cluster associated to a lower number density. This can result in problems when examining the connectivity of clusters, since it may occur that some less densely sampled parts of the contour may result in the existing boundary to remain undetected. However, the homogenization of the spacing between adjacent boundary particles would require the introduction of new boundary particles in areas of the cluster of low number density or the elimination of boundary particles in areas of higher number density, the former signifying an introduction of interpolated information to define the cluster boundary and the latter implying a loss of detail in the description of the cluster topology.

On the other hand, if one were to resort methods in which the obtained cluster boundary presents a resolution which is lower than the one presented by the set of cluster particles itself, the main resulting problem is that the boundary will not be as descriptive as the cluster itself, and information describing the cluster's topology would be lost. Moreover, this method would require the non-trivial decision of which resolution is appropriate for the current application. For instance, if the previously presented cell-based analysis were to be followed, it would be necessary to select a cell size which adequately extracts a closed curve of boundary particles and does not nullify the idiosyncrasies of the cluster's boundary in excess.

Therefore, the current study can be said to follow the concept of *wu wei* in the definition of cluster boundaries, since it seeks to conserve the complete extent of detail which a boundary made up of adjacent cluster particles allows for, at the same time refusing to homogenize the separation between boundary particles in face of the appearance of different number densities within the cluster. As a result, the topology of the cluster is not simplified at all during this compression of cluster information into a boundary and a set of interior cluster particles, and its resolution is only limited by the average number density associated to the dataset. Moreover, the current implementation is capable of detecting holes within the cluster, including these details within the obtained contours.

Once the motivation and particularities associated to the determination of a set of points defining a boundary contour for a single DBSCAN cluster within an individual snapshot in Z have been presented, it follows to describe the method employed in the current study.

2.3.2 Description of Employed Method

Given that the purpose of the implemented method is to obtain a cluster boundary which maintains as much information as possible regarding the cluster's topology, it follows for the contour which this method defines to be made up of a selection of particle positions included within the same cluster. More generally, a cluster particle must fulfill a particular criterion for it to be classified as a boundary particle, and therefore the implemented method must proceed to examine whether each of the cluster particles within the level of Z under analysis satisfy such criterion.

At its core, a cluster particle which can be defined as a boundary to such cluster is a particle which is not surrounded by neighboring cluster particles in all directions. More simply, this particle must be adjacent to a "gap" in which no cluster particles appear. Therefore, it follows to examine the neighborhood of the cluster particle, and within this neighborhood, determine if there exists a region in which cluster particles are absent. In the implemented method, these empty regions within the neighborhood of the cluster particle are detected by measuring the angle between adjacent particle-neighbor particle vectors. More visually, if the cluster particle is imagined to trace vectors towards all of its neighboring particles, as soon as the angle between two of these adjacent vectors is larger than a specified threshold, a region absent of cluster particles is said to exist within the vicinity of this particle, and thus the particle is part of the cluster's boundary. This criterion is formalized in Definition 2.3.1 and visualized in Figure 2.24.

Definition 2.3.1. Boundary Cluster Particle: Given a cluster particle position p, a neighborhood radius r, the set of cluster particle positions $n_i \in N_r(p)$ within the r-neighborhood of p, sorted based on the orientation of vector $n_i - p$ with the X direction, an angle function $\theta(p, n_i)$ defining the angle between $n_i - p$ and the X direction, and a threshold angle δ, p corresponds to a boundary particle if, for any $n_i \in N_r(p)$:

$$\theta(p, n_{i+1}) - \theta(p, n_i) \geq \delta$$

Bearing into account this criterion, this method, for each isolated snapshot in Z, examines whether this condition is satisfied for each of the particles classified as cluster particles by the previous application of DBSCAN, and labels as boundary particles all those cluster particles which fulfill the criterion. In this way, a method capable of defining cluster contours which express an unprocessed version of the cluster's topology is developed, based on examining the fulfillment of a particular criterion for every cluster particle. Nevertheless, the nontrivial choice of the parameters defining the boundary particle criterion, in particular the threshold angle δ and the neighborhood radius r must be carried out in an adequate way.

2.3.3 Adequate Selection of Parameters

In order to appropriately select a suitable pair of parameters δ and r, it is necessary to understand what these parameters represent. In simple terms, the combination of these two represents the ease with which the boundary particle criterion is satisfied. Just as previously, density-based clustering algorithms were conceived as assigning a cluster to a region of the domain with data sample density larger than a specified threshold, this method can be interpreted as assigning a particular label to cluster particles with a local number density below

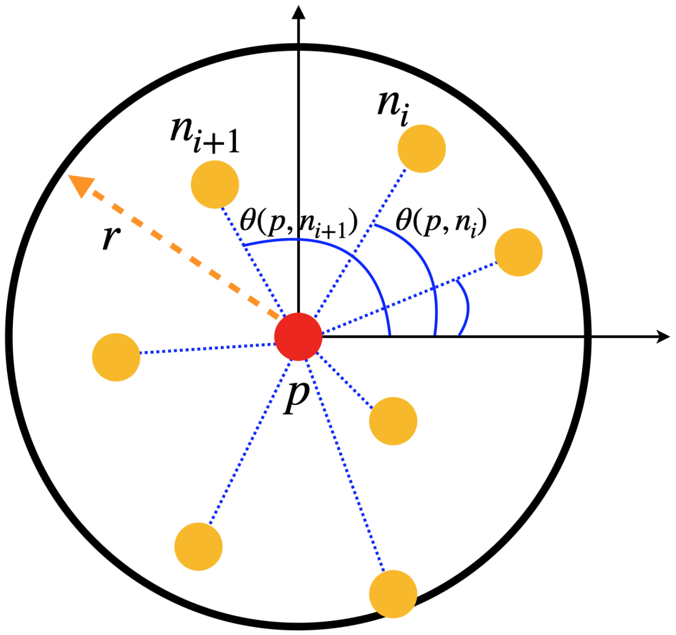

Figure 2.24: A cluster particle p, its r-neighboring particles n_i, and their associated angle functions $\theta(p, n_i)$

a specified threshold. Hence, the optimization of these two parameters has to be carried out at the same time, since an optimum threshold angle δ will apply exclusively to an optimum radius r, and viceversa.

In the current implementation, a neighborhood radius r is defined making sure that its value is adapted to the average number density of the dataset. Then, if the sheet thickness employed to project particle positions into a single level of Z is varied, this neighborhood radius is able to take such variation into account. With this purpose, r is set to be proportional to the value of ϵ utilized in the previous DBSCAN analysis, which itself is proportional to the average reachability distance which would result in the case that the same number of particle positions were distributed according to a random Poisson process. Given that ϵ is capable of adapting to variations in the average number density of the dataset, so will r respond to such variations. Therefore, when speaking of modifying r in this section, one is in reality changing the constant k with which ϵ is multiplied in order to define r.

Then, the specification of k must take into account an intrinsic characteristic of particle clusters, thus being applicable to a dataset of any number density. Since the number of particles defining a cluster is finite, there are necessarily voids between adjacent particles. In areas of a cluster of lower number density, these voids caused by the separation between adjacent particles are more significant than in areas within the cluster of high number density. Thus, the neighborhood radius r must allow for a boundary criterion which is sensitive enough to be fulfilled at cluster boundaries of all types, but is not triggered by these voids internal to a cluster that appear due to the separation between adjacent particles. It is worth noting that while as these voids are mainly caused by the limited number of particles defining a cluster, it may occur that actual holes exist within a cluster, which would also be perceivable if the number of particles describing the cluster were infinite. Therefore, the implemented method will only be able to distinguish voids of the latter category from those of the former by means of their associated length scale.

More precisely, for a fixed δ, an increase in r will result in a significant decrease in the number of particles defined as boundary particles until a particular r from which all smaller internal voids cease triggering cluster particles to be classified as boundary particles, and the decrease in the number of boundary particles with the growth of r is milder and due to a decrease in detail in defining cluster contours.

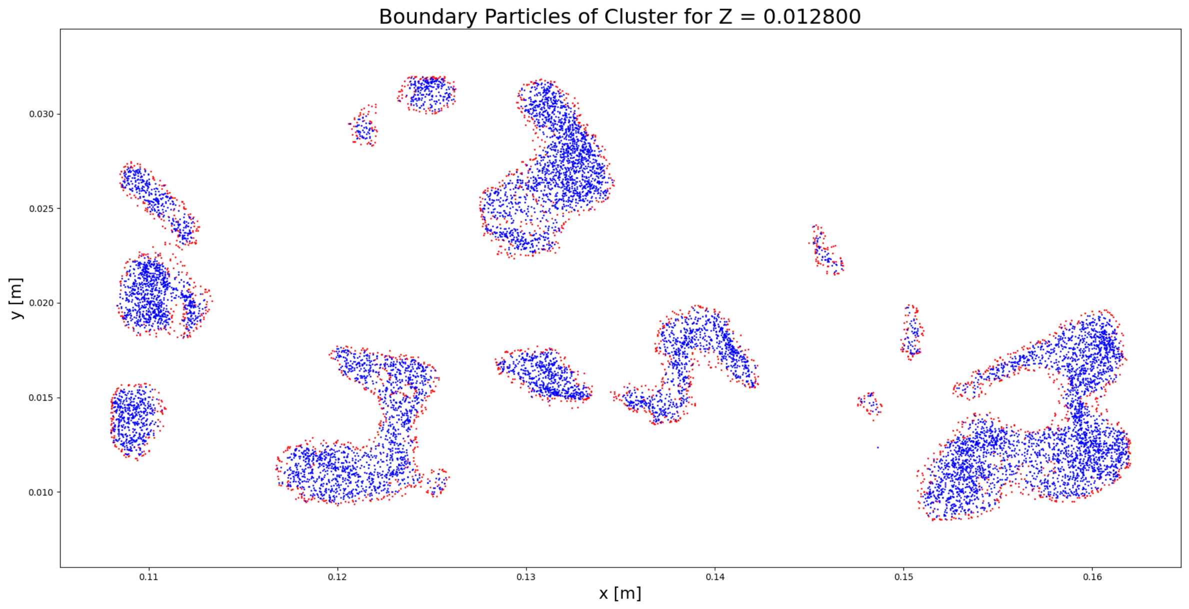

Figure 2.25: Resulting Cluster Boundaries for $k = 0.35$ and $\delta = \pi/3$ rad - In blue: cluster particles, in red: boundary particles.

This trend is visible in Figures 2.25, 2.26, and 2.27, for a fixed $\delta = \pi/3$. Once k (and thus $r = k \cdot \epsilon$) is increased from 0.35 to 0.5 the internal voids triggering the labeling of interior cluster particles as cluster contours cease having this effect, such that the number of cluster particles defined as boundary particles is significantly reduced. Nevertheless, when k is further increased from 0.5 to 0.7, the variation in the definition of cluster boundaries is very slight, only due to some particles close to the boundary which cease being classified as boundary particles. The evolution of the proportion of cluster particles fulfilling the boundary criterion with k is more precisely presented in Figure 2.28. Here, it is possible to discern that although the presented curve is somewhat noisy, its slope does decrease significantly after 0.5. As a result, the current implementation has employed $k = 0.5$.

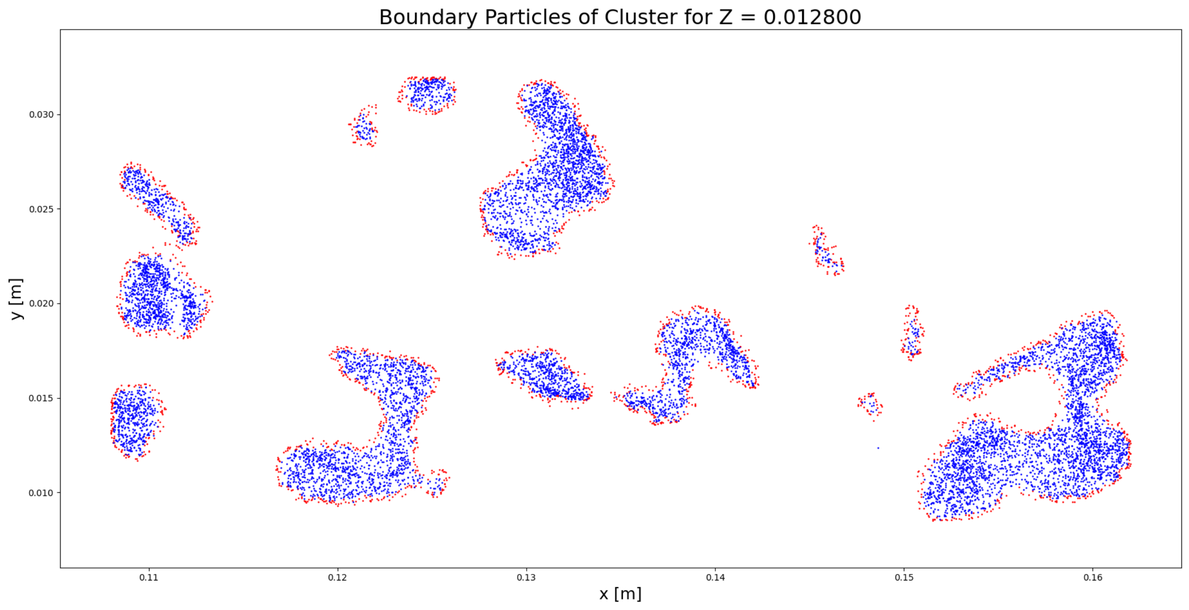

Figure 2.26: Resulting Cluster Boundaries for $k = 0.5$ and $\delta = \pi/3$ rad - In blue: cluster particles, in red: boundary particles.

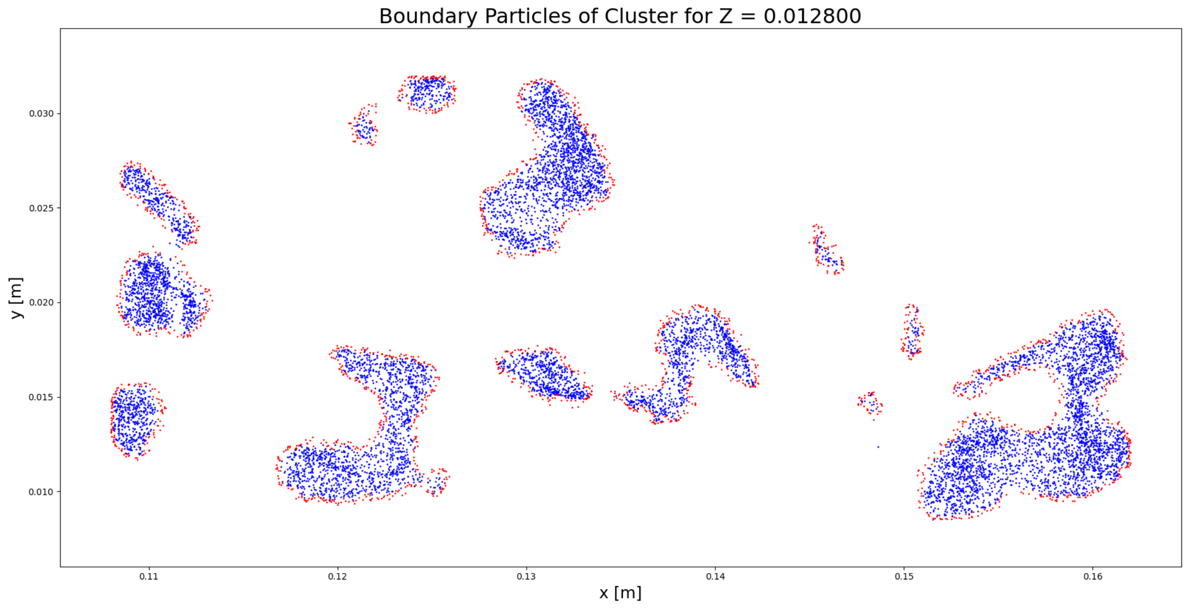

Figure 2.27: Resulting Cluster Boundaries for $k = 0.7$ and $\delta = \pi/3$ rad - In blue: cluster particles, in red: boundary particles.

On the other hand, for a fixed value of k it is expected for increases of δ to cause the boundary particle criterion to become more restrictive, since a greater separation between adjacent particle-neighboring particle vectors will be required for such particle to be labeled as a boundary particle. In any case, as long as this threshold angle is large enough to allow for a

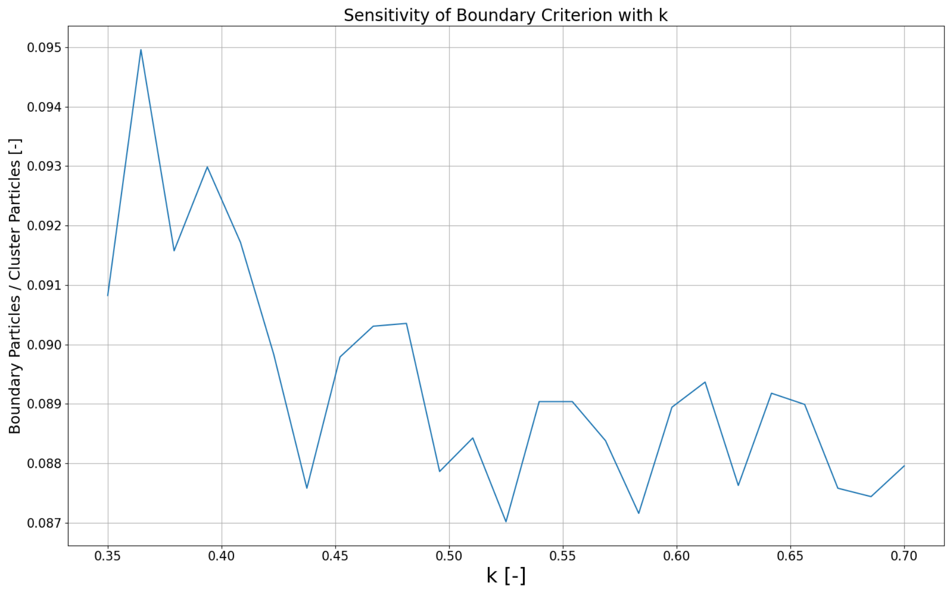

Figure 2.28: Sensitivity of Boundary Criterion with k - The evolution of the proportion of cluster particles fulfilling the boundary criterion with k is here presented.

minimum restriction within the cluster boundary criterion, this variation is negligible, as the proportion of cluster particles fulfilling the boundary criterion remains constant. In Figures 2.30, 2.26, and 2.29, one can see how the decrease in δ causes a very slight relaxation in the boundary particle criterion. In this way, δ has been set to $\pi/6$.

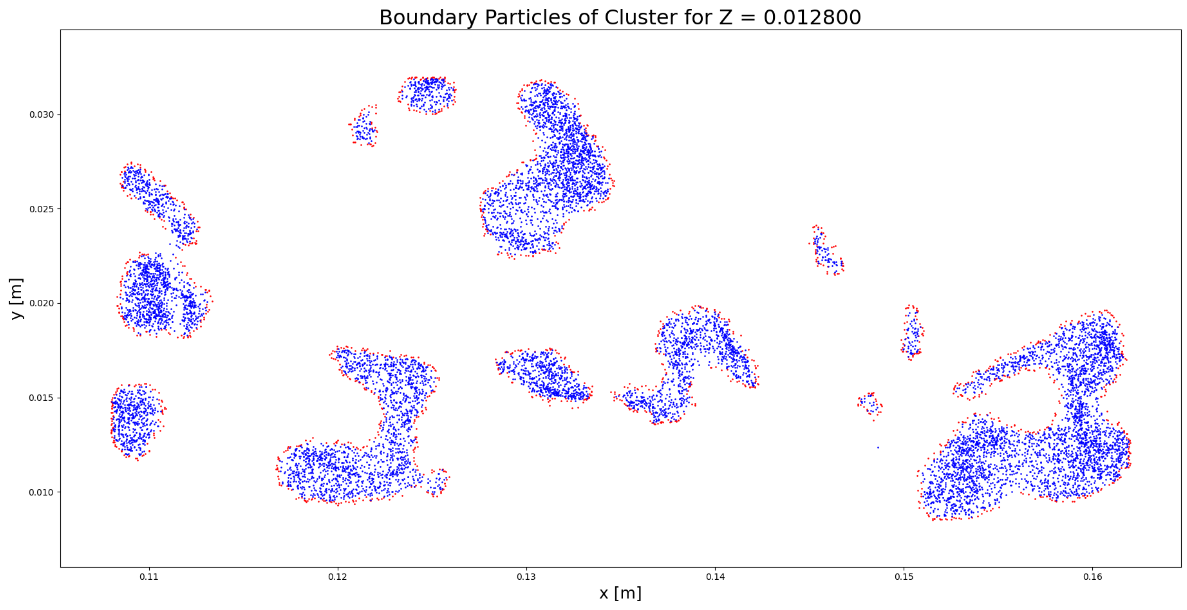

Figure 2.29: Resulting Cluster Boundaries for $k = 0.5$ and $\delta = \pi/2$ rad - In blue: cluster particles, in red: boundary particles.

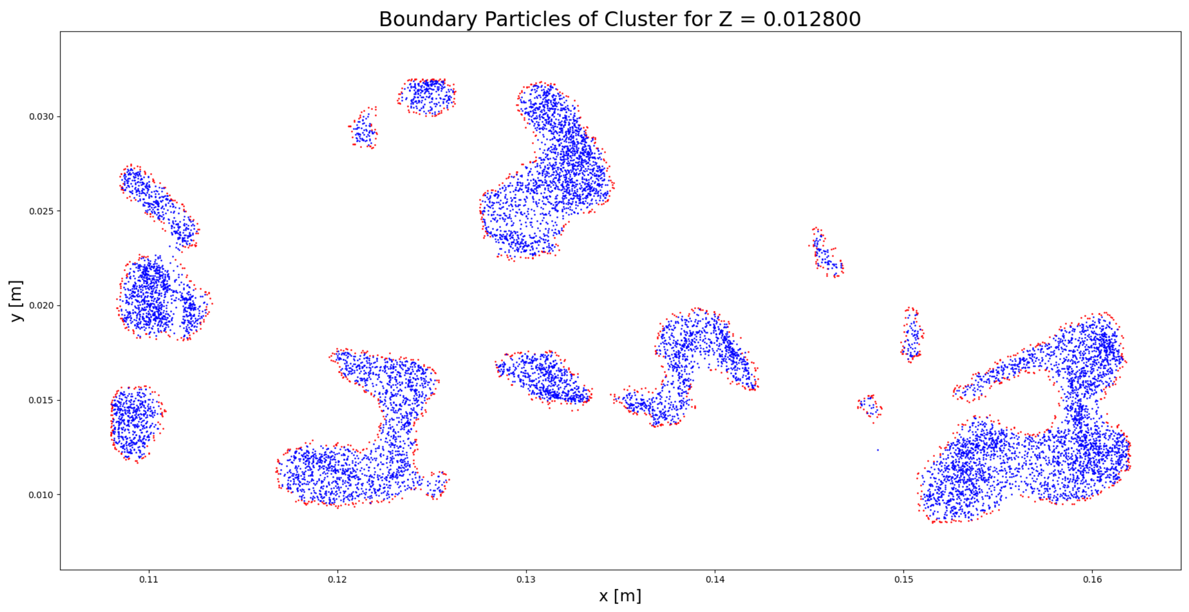

Figure 2.30: Resulting Cluster Boundaries for $k = 0.5$ and $\delta = \pi/4$ rad - In blue: cluster particles, in red: boundary particles.

2.3.4 Results

What results from the current implementation is an appropriate determination of cluster boundaries, defined by a set of cluster particles where adjacent boundary particles are also adjacent within the cluster itself. As the homogeneity of this contour is not imposed, areas of high number density within the cluster are bounded by more closely spaced particles than areas within the cluster where particles are not as densely packed. Thus, the current method does not circumvent the presented problem of inhomogeneous number densities within the cluster. However, this does not obstaculize later steps of analysis.

The selected combination of parameters yields a cluster boundary which is generally densely populated, in order to ensure that in the later stages of analysis within this study, cluster connectivity is determined ensuring that all cluster boundaries are respected. Figures 2.31, 2.32, 2.33 and 2.34 display the obtained cluster boundaries for Z levels of 0.136 m, 0.144 m, 0.152 m, and 0.160 m, respectively.

2.3.5 Attempted Alternatives

Given that the developed methods for later steps of the analysis carried out within the study require the separation of the two-dimensional domain within a single snapshot in Z into cells

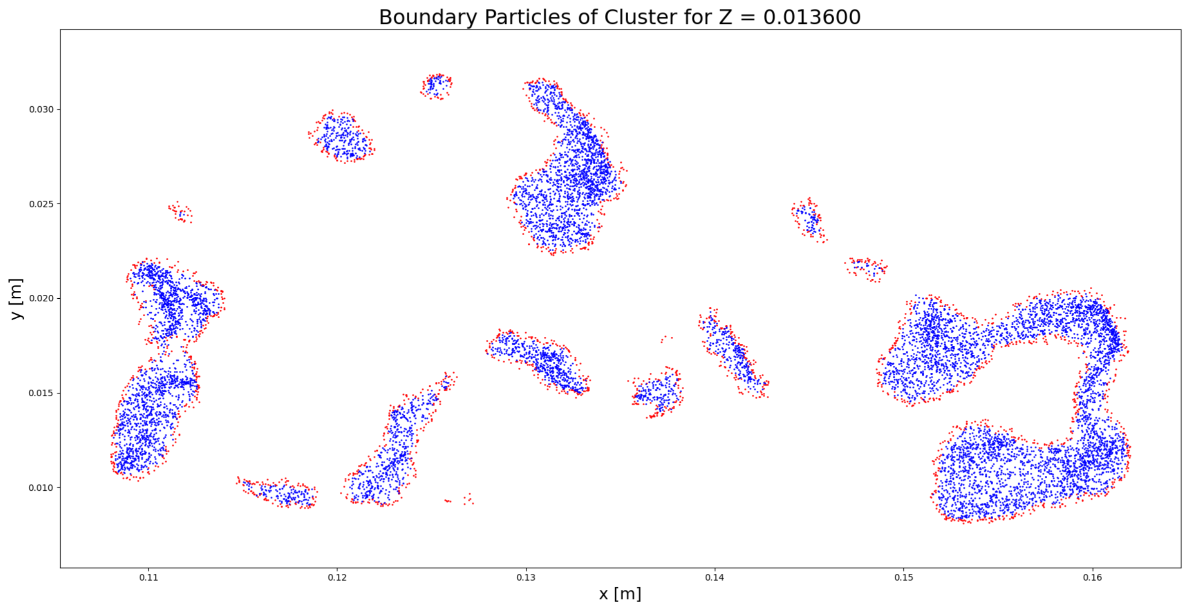

Figure 2.31: Resulting Cluster Boundaries for $k = 0.5$ and $\delta = \pi/3$ rad - $Z = 0.136$ m - In blue: cluster particles, in red: boundary particles.

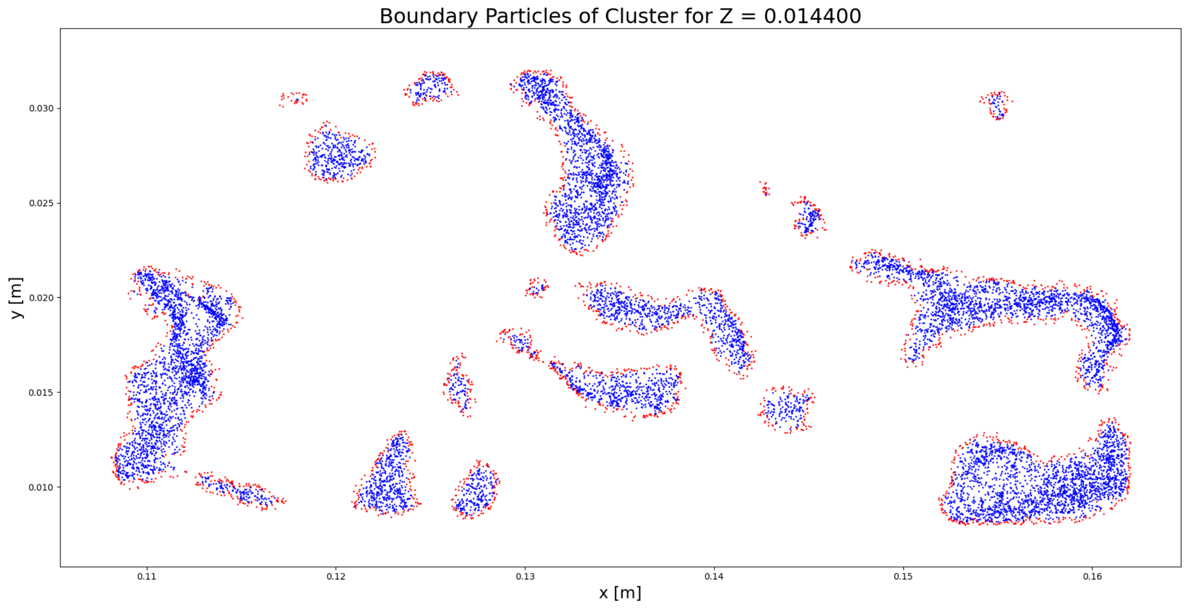

Figure 2.32: Resulting Cluster Boundaries for $k = 0.5$ and $\delta = \pi/3$ rad - $Z = 0.144$ m - In blue: cluster particles, in red: boundary particles.

of uniform size, this decomposition can also be taken advantage of in order to define cluster boundaries in a simple and computationally efficient way.

More precisely, for a known cell size, one can define a uniform grid occupying the whole

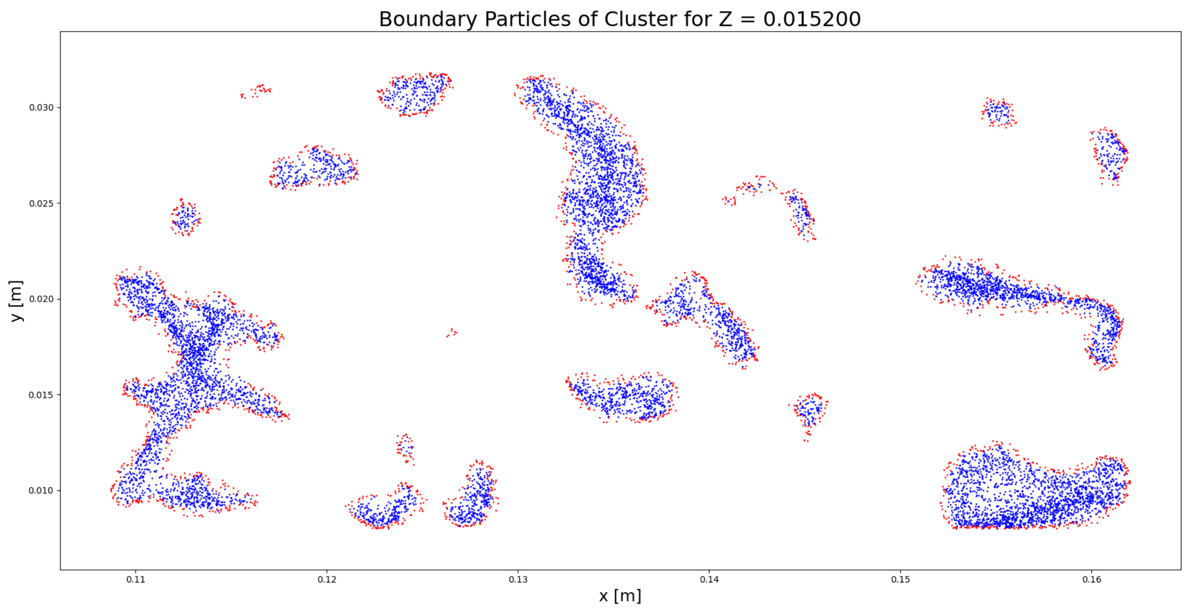

Figure 2.33: Resulting Cluster Boundaries for $k = 0.5$ and $\delta = \pi/3$ rad - $Z = 0.152$ m - In blue: cluster particles, in red: boundary particles.

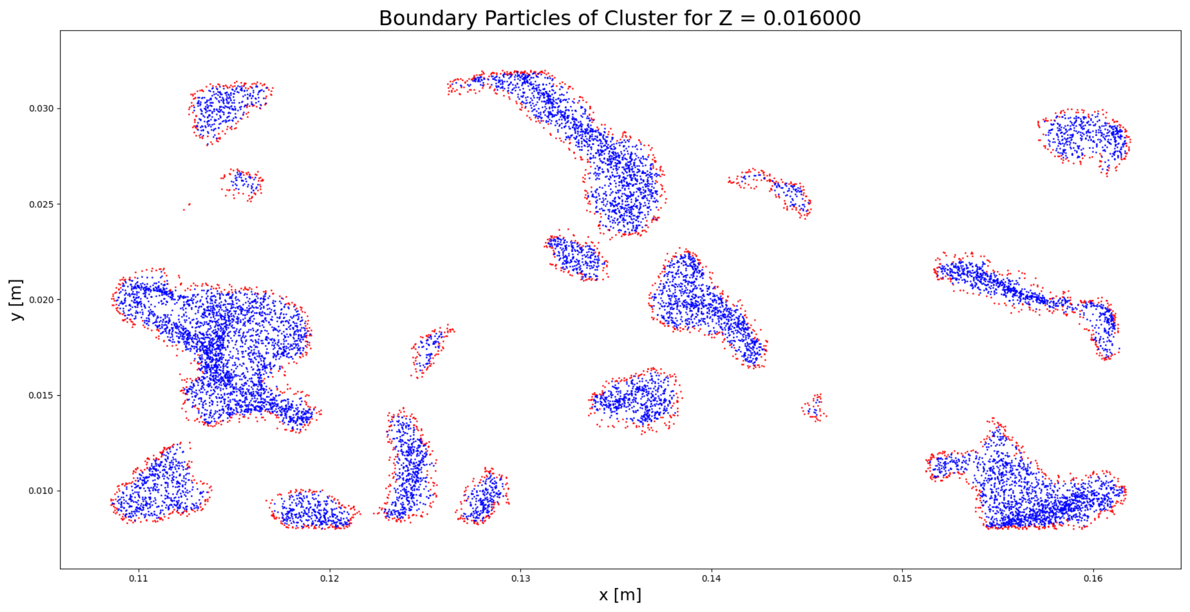

Figure 2.34: Resulting Cluster Boundaries for $k = 0.5$ and $\delta = \pi/3$ rad - $Z = 0.160$ m - In blue: cluster particles, in red: boundary particles.

domain, where each cell is associated to a cell center of known coordinates. For each of these cells, one is capable of examining if there exist cluster particles within it, or if otherwise the cell is empty. Once the population of each cell is determined, it is possible, for each of the cells containing cluster particles, to observe whether any of the neighboring cells are empty. If this is so, one can label this cell as a cluster boundary cell, and later construct a cluster boundary based on the cell centers of all cluster boundary cells.

What results from this method is a cluster contour which simplifies the cluster's topology insofar as the cell size employed is much larger than the characteristic length scale of the smallest details defining such topology. Moreover, if one is to employ too small cells, what results is that empty cells exist even within the cluster, and the cluster boundary ceases to be realistic. Therefore, a nontrivial decision of cell size has to be carried out, in order to select a cell size which does not excessively simplify the cluster's shape but also allows for a reasonable boundary. Although in the current study, this alternative allowed for relatively fast computations of the cluster boundary, it was preferred to maintain the more detailed description of the cluster topology which the previously described method allows for, albeit a higher computational cost.

It is necessary to note that the cell size for this method is specified as proportional to the neighborhood radius ϵ employed in the previously carried out DBSCAN clustering analysis in order to allow for independence of the average number density within the dataset, such that the cell size c follows $c = k \cdot \epsilon$. Therefore, in Figure 2.35, one can see that for $k = 0.132$, the cell size is not large enough to result in reasonable cluster contours, whereas in Figure 2.36, the cell size is duplicated and the cluster's shape is greatly simplified.

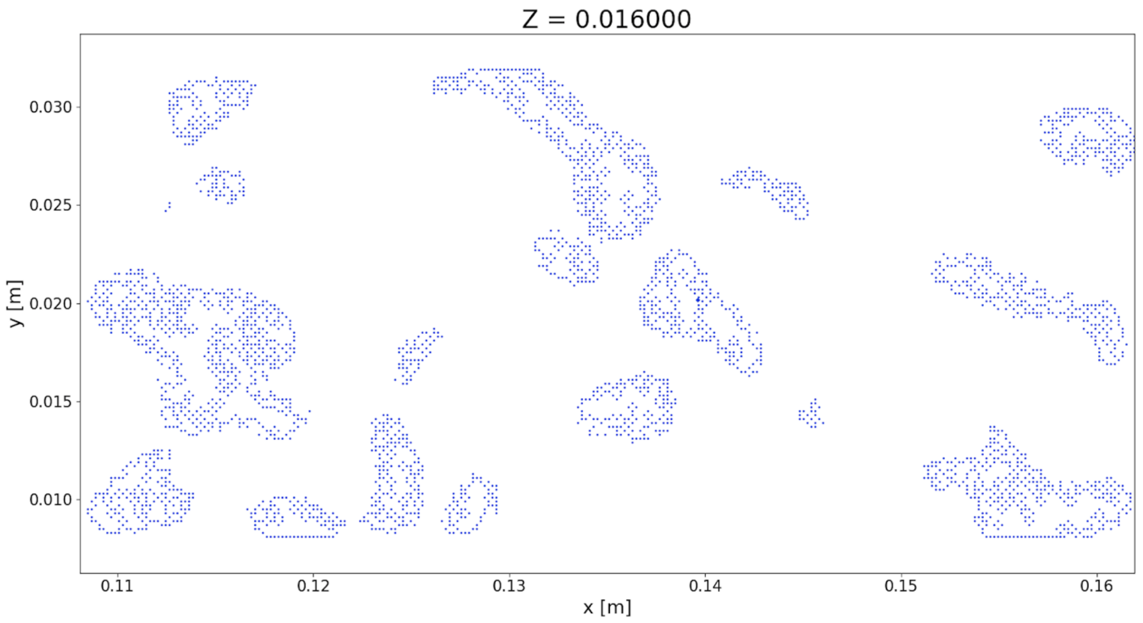

Figure 2.35: Resulting Cluster Boundaries with Grid Decomposition and $k = 0.132$

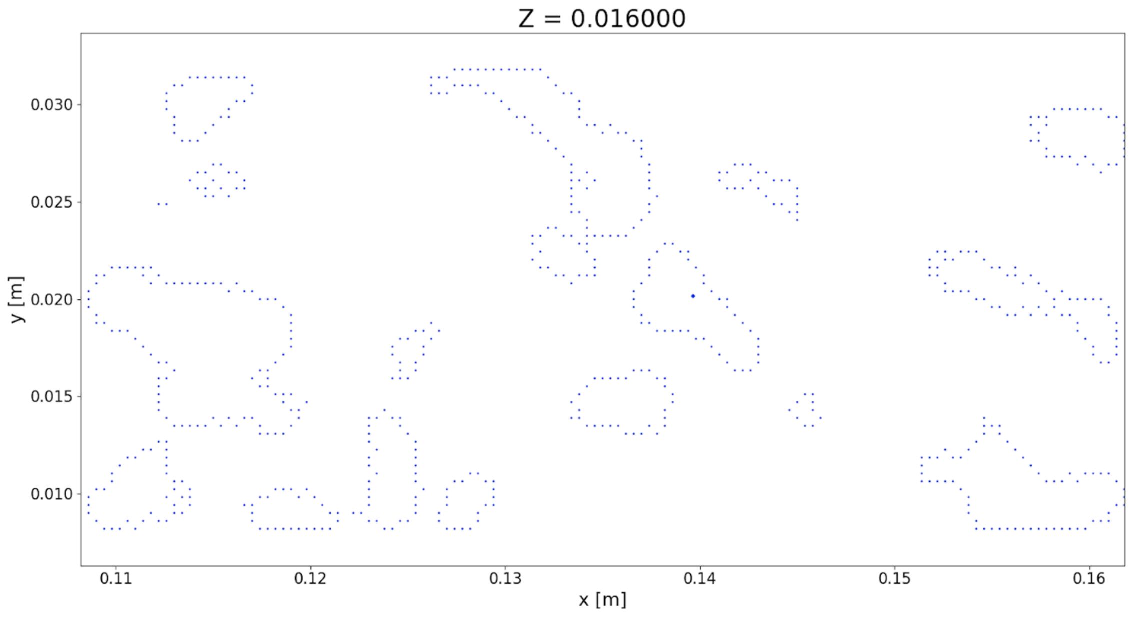

Figure 2.36: Resulting Cluster Boundaries with Grid Decomposition and $k = 0.264$

2.4 Definition of a *Brute Skeleton* Describing 2D Cluster Topology

2.4.1 What is a *Brute Skeleton* and why is it worth defining?

Up to now, the current study has dealt with how to project a set of three-dimensional particle positions into a discrete number of parallel planes, in order to analyze the two-dimensional domain which each snapshot in Z results in. Moreover, it has been discussed how to separate, for each of these planes containing normal projections of particle positions, densely populated regions by classifying them as clusters by means of DBSCAN. The next step of analysis to be presented has been the determination of cluster boundaries based on these two-dimensional clusters, in order to express as much of the cluster's topology in the most efficient way possible. What follows now is the determination of an appropriate set of interior points within each two-dimensional cluster, these points defining what is called within the context of this study as the cluster's *brute skeleton*.

As has been stated previously, the topological and temporal analysis of a particle cluster requires the simplification of the information defining the cluster, in order to carry out such exploration in a more efficient way. It has already been described how for every two-dimensional snapshot in Z of a cluster, one can obtain a cluster boundary curve, and how once the connectivity of cluster particles along different levels of Z is examined, one can unite the boundary curves of the same cluster to form a boundary surface made up of a cloud of points from different levels of Z. Nevertheless, to further condense the description of a cluster's topology, a compression necessary both to examine the connectivity of different two-dimensional clusters along different Z as well as to track the evolution of a particular cluster with time, it is convenient to define a constellation of points interior to the two-dimensional cluster which adequately carry out this condensation.

More precisely, as will be seen in later sections of this study, it is specifically this interior constellation of a single two-dimensional cluster which will be connected to other constellations of clusters at different levels of Z. Moreover, it is this constellation which will be tracked in different steps of time in order to analyze how a cluster progresses with time. However, from a more general point of view, just as Rigel, Betelgeuse and Bellatrix, among others, define Orion, it is possible to consider this brute skeleton as an ultimate simplification of the cluster's topology. As a result, each of the points within a constellation is associated to a large group of cluster particles, such that the properties of a constellation point are directly attributable to the cluster particles it represents.

To the experienced reader, it may seem that this step of analysis intends to re-invent the wheel, overlooking the existence of a topological skeletonization in order to condense the description of complex geometries such as the one of a cluster. This topological skeletonization can produce a geometrical representation of the three-dimensional shape of the cluster by means of a connection of arcs [19]. Such a representation of the three-dimensional cluster would be interesting to obtain, since the combination of brute skeleton points of different levels of Z results in something similar. In any case, the attempts for such a development within this study have resulted in overly complex and expensive methods which were not as advantageous as the straight-forward determination of a so-called brute skeleton exposed in this section.

Nevertheless, a set of guidelines can be established in order to generate a constellation of points which defines the topology of the two-dimensional cluster in a way which is convenient for the purposes of this study. With the objective of examining cluster connectivities along different levels of Z, it is required for the brute skeleton to populate as much of the cluster as possible. On the other hand, given the key role of cluster boundaries in the description of cluster shapes, it is necessary for the developed constellation to be based on points which are as far as possible from any cluster boundary. In fact, given the notion of a skeleton curve as a curve equidistant to the boundaries of the figure it represents [19], the more a brute skeleton point is farthest possible from all boundaries, the more it will resemble the actual topological skeleton of the cluster. Thus, there needs to be a balance between explicitly filling up the space within a cluster with a representative interior points and simplifying this representation by obtaining points as equidistant as possible from all cluster boundaries. Given these guidelines, a description of the procedure followed in order to generate such a constellation is exposed.

2.4.2 Description of Employed Method

Once the concept and usage of the brute skeleton of a two-dimensional DBSCAN cluster have been introduced, and the set of requirements which such a constellation must fulfill in order to be useful in later stages of analysis has been presented, it follows to present the methods with which such a set of points is defined. The algorithm with this objective can be seen a maximization constrained by a set of restrictions. More precisely, the objective of this procedure is to determine the set of highest ranking positions within the cluster, where these points are ranked according to the proximity of cluster boundaries. Constraining this determination is a minimum separation between brute skeleton points and between brute skeleton and cluster boundaries.

Given the key role of cluster boundaries in the definition of this internal constellation of points defining the cluster's topology, and the fact that these internal points are determined based on a scoring system which ranks them, it is natural that this employed scoring system is intrinsically related with the proximity of cluster boundaries. As a result, the developed method ranks all points interior to the cluster based on their distance to the closest cluster boundary. The scores assigned to interior cluster points within a two-dimensional snapshot in Z are visible in Figure 2.37.

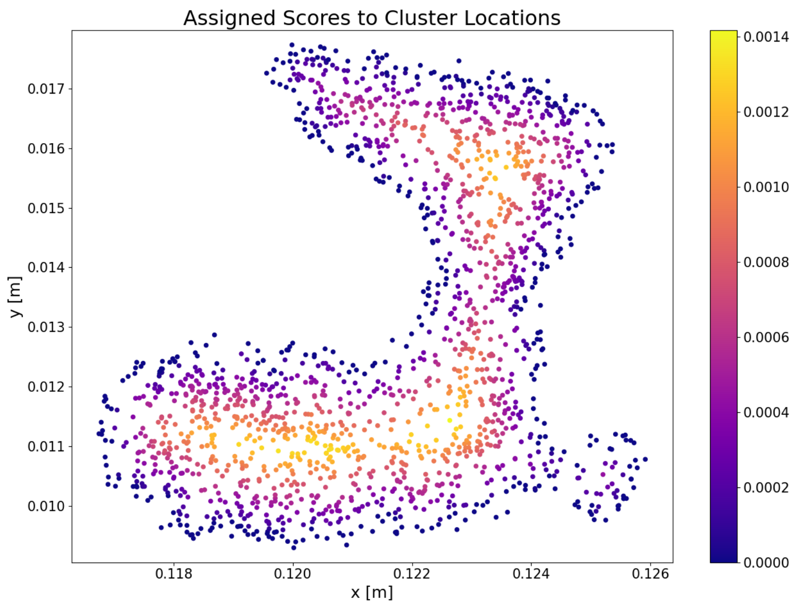

Figure 2.37: Contours of Assigned Scores to Cluster Locations

However, if one were to base the position of the cluster's brute skeleton only on those regions within the cluster in which this score is relatively elevated, one would obtain a series of very close points in very isolated locations of the cluster. Instead, it is necessary for this interior constellation to be distributed along the cluster. To implement this, a minimum separation between brute skeleton points is enforced, such that for every iteration of this algorithm, the highest ranking position internal to the cluster which satisfies the specified minimum separation to all previously defined brute skeleton positions is determined. While as for the first iteration, the highest ranking cluster position is directly selected, the next iterations have to consider only the highest ranking positions outside of a specified neighborhood radius of all previously defined brute skeleton positions. Also, in order to facilitate the determination of cluster connectivities along different levels in Z, a minimum distance between brute skeleton particles and cluster boundaries is also enforced. As a result of these two enforced minimum separations, between positions within the constellation and between brute skeleton positions and boundary particles, the internal set of points defining the cluster's shape spreads out throughout the two-dimensional cluster.

In any case, the application of this procedure requires the possible positions within the cluster to take into consideration at every iteration of the algorithm to be finite. In other words, if for every iteration, the highest scoring position which is also sufficiently separated from all previously selected positions is to be selected, it is convenient for the number of candidate positions to consider to be limited, to avoid performing an maximization over a continuous space within the two-dimensional cluster. Therefore, one could take into account the positions of all cluster particles within the same cluster, since these positions are known and finite. However, this would result in a considerable computational cost of the method, since for the employed dataset, the average number of particles within a two-dimensional particle cluster is 1575.86. Alternatively, one can divide the space within the cluster by means of a uniform grid of known size, such that every cell within the grid has an associated cell center. Based on each cell center position, one can assign a score to each cell, and thus carry out the same procedure based on a set of points which is finite, known, and uniformly distributed. More importantly, for an adequate selection of the implemented cell size (presented in the following section), the computational cost of the procedure can be reduced. Therefore, the current implementation employs a two-dimensional discretization of the area within every cluster, and considers as candidate positions for the brute skeleton the highest ranking cell centers which are also sufficiently separated from all previously selected cell centers.

Based on the exposed criteria, the procedure of determining the brute skeleton positions is summarized as follows. For a single DBSCAN cluster within a two-dimensional snapshot in Z, the area within the cluster is discretized by means of a uniform grid, and each of the cell centers in the grid is assigned a score equal to their distance to their closest cluster boundary. Then, for each iteration, the algorithm proceeds by disabling all cell centers which are too close to previously defined brute skeleton cell centers or to cluster boundaries, and by selecting the highest scoring cell center which is not disabled and labeling it as part of the cluster's representative inner constellation of points. This procedure continues until no more cell centers can be selected for a given two-dimensional cluster.

2.4.3 Adequate Selection of Parameters

Once the procedure based on which this determination of an interior constellation of points defining the two-dimensional cluster's topology is carried out, it is necessary to present the criteria behind the selection of its defining parameters. Based on the previous description of the employed procedure, one can expect the parameters playing a role in this method to be the cell size cs for the discretization of the area within a cluster, the minimum separation between brute skeleton particles s, and the minimum separation between brute skeleton and cluster boundaries c. Note that since all of these magnitudes are distances, it is convenient to make these parameters proportional to a known reference length scale that is relevant. In this case, s and c are defined as proportional to the Kolmogorov length scale η of the flow, since they are thought to be related to the characteristic size of the turbulent structures in the flow, whereas the cell size cs employs the radius ϵ employed in previous steps of this analysis, in order to associate with the average number density of the dataset. In this particular method, it is necessary to bear in mind that a selection of parameters is adequate as long as it allows for later steps of analysis within this study to be carried out without issues and fulfills the aforementioned guidelines, such that a complete optimization is not necessary.

With respect to the cell size with which to carry out the two-dimensional discretization of the area within every DBSCAN label, it is necessary to take into account that this parameter will greatly affect the computational cost of the implemented method. The smaller the cell size, the more candidate cell center positions have to be taken into account for every iteration of the method, and as a result, the greater the time associated to each iteration. Moreover, above a certain number of cell centers within a cluster, it is expected for the result of this algorithm to vary mildly, since the highest ranking points will not vary significantly in position. However, the way in which the two-dimensional cluster is discretized puts a limit on how large this cell size can be. In order to discretize the area within the cluster, a uniform grid is defined along the whole two-dimensional domain of the current snapshot in Z, and the grid cells containing cluster particles from that cluster are extracted. If these grid cells are too large, it may occur that the cell center of a cell associated to such cluster is actually outside what the previously defined cluster boundary curves define as the cluster. This is visible for values of k_{cell} around 0.4 (where the cell size is defined by $k_{cell} \cdot \epsilon$).

Bearing in mind this undesirable phenomenon and the evolution of execution times with k_{cell} as presented in Table 2.4, the current implementation employs $k_{cell} = 0.2$. Furthermore, the resulting discretization of a single snapshot in Z is presented in Figure 2.38, where the cell center of each cell is classified according to the particle cluster which the cell contains, where a cluster label of -1 corresponds to cells void of cluster particles. This discretization results in an average of 1.18 cluster particles occupying each cell, and for those cells containing cluster particles, the average number of particles is 4.64.

On the other hand, the minimum separation between skeleton points determines both the number of brute skeleton points which will be generated in order to describe a cluster's topology

k_{cell} [-], where Cell Size $cs = k_{cell} \cdot \epsilon$	Execution Time [s]
0.1	14.359
0.2	2.879
0.3	1.142

Table 2.4: Execution Times for a Single Snapshot in Z for different k_{cell} - $k_s = 0.4$

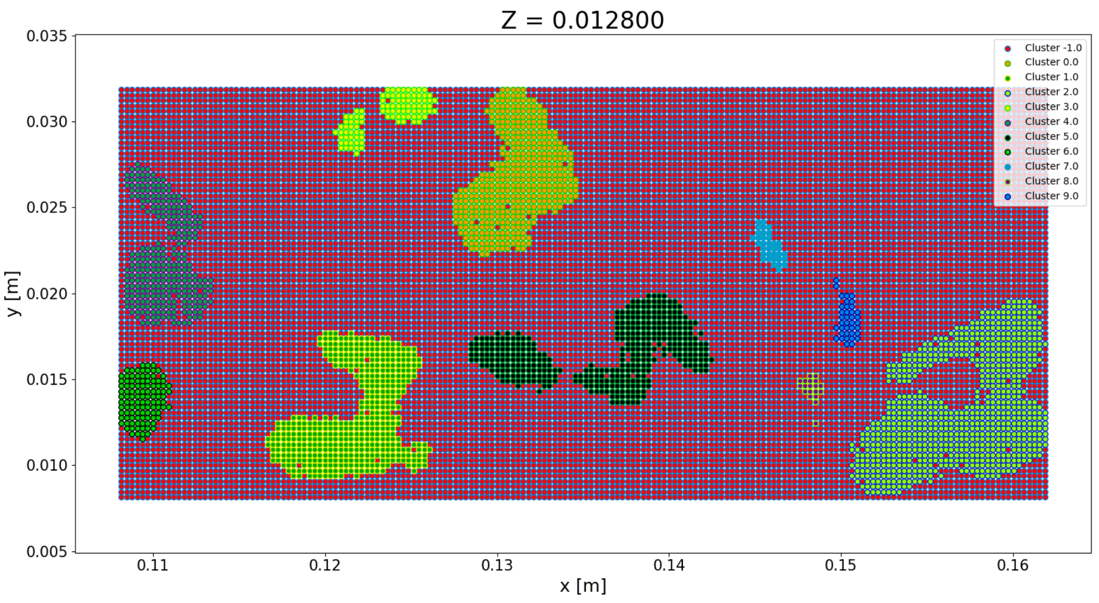

Figure 2.38: Classified Cell Centers of a Discretized Snapshot in Z - $k_{cell} = 0.2$

as well as to which extent the area within the cluster is populated by these points. The greater this minimum separation, the smaller amount of skeleton points that are to be employed to describe the cluster's topology. As a consequence, the condensation of the cluster's shape is carried out with less points, but the examination of the cluster's connectivity along different levels of Z is more difficult. On the other hand, the less significant this k_s, (where Separation $s = k_s \cdot \eta$), the more the inner area within the cluster is filled up by brute skeleton points, facilitating connectivity determinations but causing this constellation of points to diverge from the topological skeleton. This trend is visible in Figures 2.39, 2.40 and 2.41, where k_s grows from 8.5, to 17.0, to 34, thereby reducing the number of brute skeleton points within the cluster, describing the cluster's topology with less detail. Given that it results in a moderate distribution of internal points, allowing for an appropriate determination of cluster connections as well as for certain simplicity in the description of the cluster's shape, this minimum separation between brute skeleton positions is enforced with $k_s = 17.0$. Note that as long as the number of skeleton points within the skeleton is sufficient to trace connections within its extremes, this separation will work with later steps of this analysis.

Lastly, the minimum separation c between internal constellation position and cluster bound-

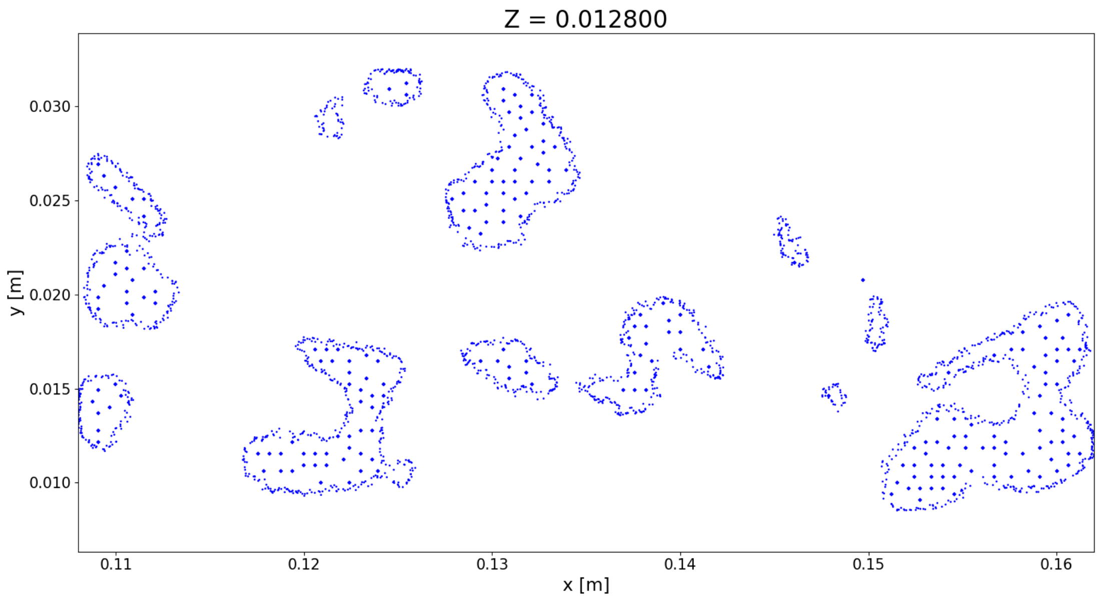

Figure 2.39: Generated Brute Skeleton for a Single Snapshot in Z - $k_s = 0.4$

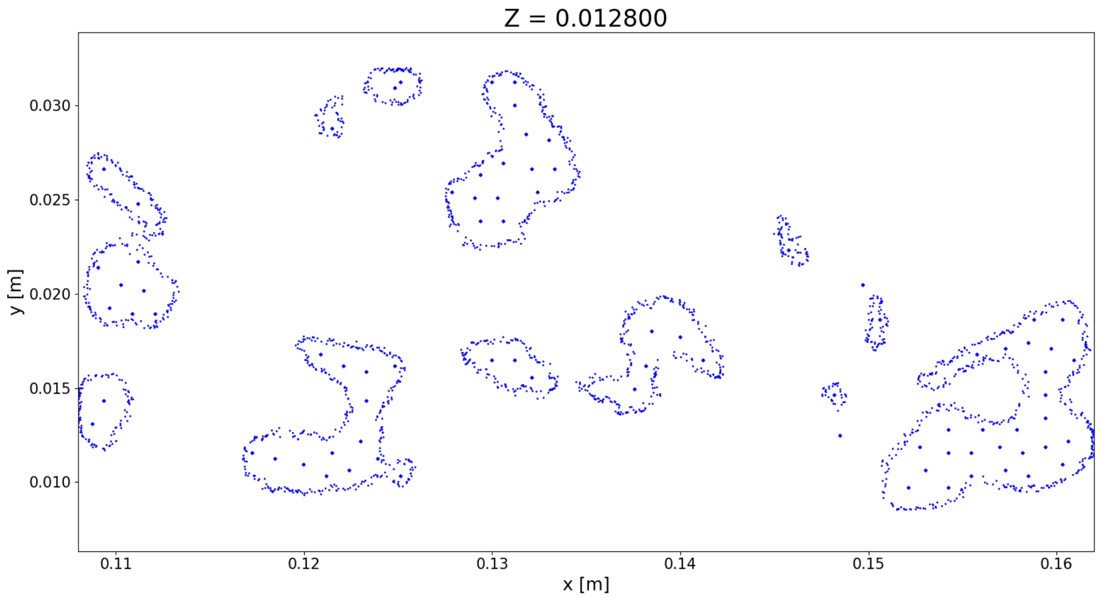

Figure 2.40: Generated Brute Skeleton for a Single Snapshot in Z - $k_s = 0.8$

aries has to be fixed based on the criteria applied in the posterior determination of cluster connectivities. As will be seen, these connectivities are examined by tracing straight trajectories between skeleton points, and observing if the trajectory is at any point closer to a boundary

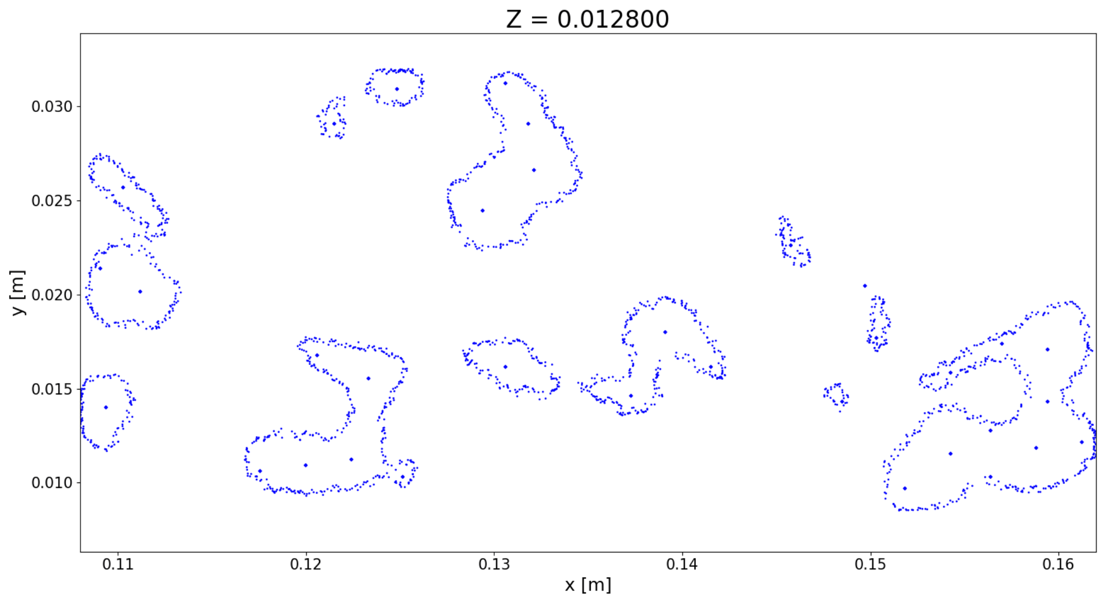

Figure 2.41: Generated Brute Skeleton for a Single Snapshot in Z - $k_s = 1.6$

particle than a specified collision distance. Therefore, it is necessary for this minimum separation c to be larger than such collision distance. As will be argued in further steps of analysis within this study, if this separation is equivalent to $k_c \cdot \eta$, $k_c = 3.2$.

2.4.4 Results

Based on these selected values for the cell size of the discretization of the area within each cluster, the minimum separation between brute skeleton points, and the minimum separation between brute skeleton points and cluster boundaries, what results is a simplified description of the two-dimensional cluster in a single snapshot in Z, based on its boundary curve and on an internal constellation of points, the latter being useful to track each cluster's temporal evolution as well as to determine whether different two-dimensional clusters within different snapshots in Z describe the same three-dimensional particle cluster.

As was mentioned in Section 2.2.5, after performing a DBSCAN analysis to classify cluster particles, all non-core data samples with a cluster label are excluded from their cluster. This results in small groups of particle positions appearing visibly separated from another body of particle positions, both groups sharing the same DBSCAN cluster label. Since the developed method tackles each DBSCAN label individually, it may occur that if this separated group of points is small, the score assigned to positions within it is not enough to guarantee the appearance of a brute skeleton point, or all of these positions are excessively close to already generated brute skeleton positions. As a result, some groups of projected particle positions do not have a representative skeleton point associated to them.

As is visible in Figures 2.42, 2.43, and 2.44, every cluster can be adequately represented by a constellation of interior points capable of defining its topology from a simplified standpoint, and which can be further employed in the steps of analysis of this study.

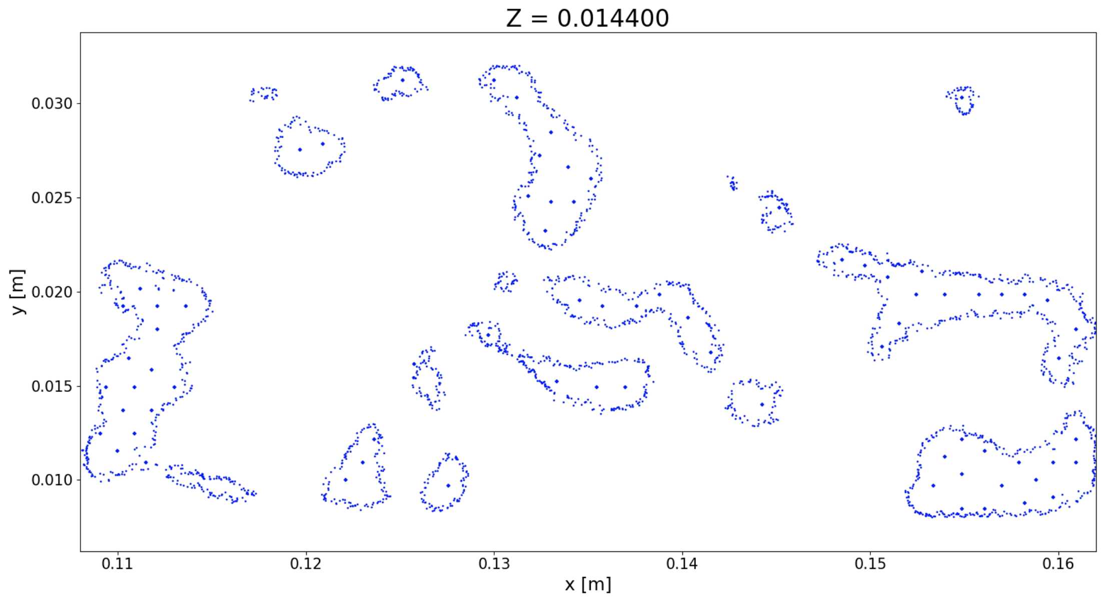

Figure 2.42: Generated Brute Skeleton for a Single Snapshot in $Z = 0.0144$

2.4.5 Attempted Alternatives

Before the implemented method was developed, two other alternative procedures to determine a set of points internal to the cluster describing its topology were attempted. While as one of them works with the resulting cloud of points representing the cluster boundaries for different values of Z, the other employs the boundary curve describing the cluster's topology for a single snapshot in Z. While as the first of these could not output adequate results, the latter incurred a higher computational cost than the finally selected method.

The first of these alternatives intended to, given the three-dimensional boundary surface of a cluster, define a trajectory from a starting to an ending point which was as equidistant as possible from all cluster boundaries. The result of this technique would have been something similar to the topolgical skeleton curve of a three-dimensional figure, as described in Sharf et al.

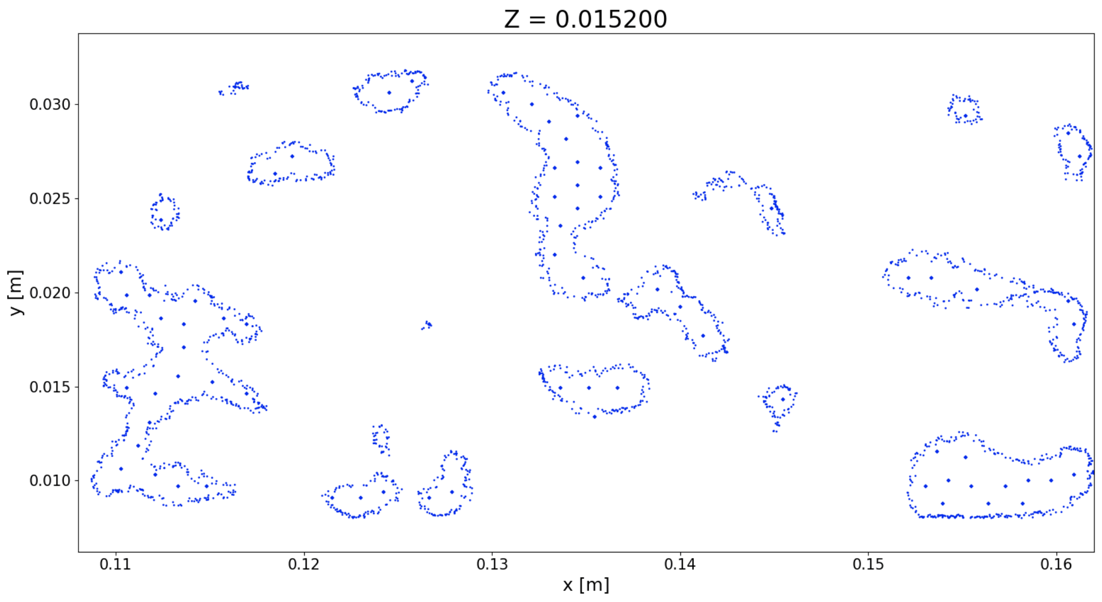

Figure 2.43: Generated Brute Skeleton for a Single Snapshot in $Z = 0.0152$

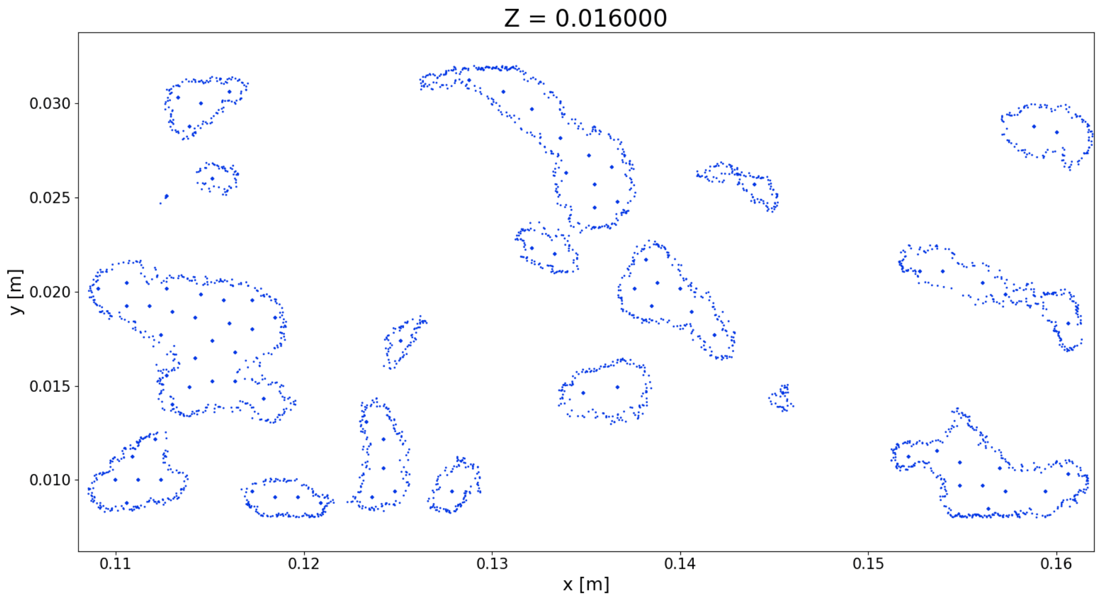

Figure 2.44: Generated Brute Skeleton for a Single Snapshot in $Z = 0.0160$

(2007) [19]. Moreover, this method required the training of a *runner* agent in charge of tracing an appropriate trajectory between a given pair of points. Every time the agent was executed, for every time step of the simulation, the runner had the task of measuring the proximity to the closest cluster boundary in every direction. Based on these measures, the agent was programmed to determine the optimum direction with which to trace the trajectory in that time step. In this study, a convolutional neural network (CNN) was employed to implement the agent's decision-making procedure, where the inputs were the proximity measures to cluster boundaries as well as the direction of a straight line to the target point, and the CNN's output consisted of the direction to be followed by the *runner* agent in that particular time step.

Nevertheless, the true core of this method is the training procedure by which an appropriate *runner* agent is generated. In this attempt, a genetic algorithm was employed, which accomplishes the optimization of a population of agents based on the following steps [20]:

1. The population is initialized. In the current application, the CNN of every *runner* agent in the population was initialized.
2. Each agent within the population is executed, and evaluated accordingly. In the current application, this consisted in simulating the tracing of a trajectory for each agent in the population, and assigning a score to such agent based on the adequacy of the trajectory.
3. The population is reproduced, in order to form a new generation based on the highest scoring agents. In this attempt, the best *runner* agents were extracted, their corresponding CNN slightly altered and combined, in order to define a new generation of agents. Based on this new generation, the previous and the current steps are repeated, until a minimum performance is achieved by the *runner* agents.

Although this procedure would have resulted in a three-dimensional curve which could also connect two-dimensional DBSCAN clusters in different snapshots in Z, and which further simplifies the description of the cluster's topology, this implementation yielded no adequate results. Moreover, since training implied the simulation of a large number of *runner* agents, the associated computational costs were significant, thus hindering the applicability of this method to an efficient routine of particle cluster analysis.

The other attempted alternative method is similar to the selected method in the sense that it employs the cluster's boundary curve within a single snapshot in Z, and based on this two-dimensional curve determines a set of points internal to the cluster within the same Z level. In simple terms, this method also tackles every two-dimensional DBSCAN cluster individually, and based on its boundary curve generates a set of interior positions defining the cluster's brute skeleton. However, this procedure works by, given an initial point within the cluster, iteratively displacing it such that the distance to the closest cluster boundary is the same for all directions. As a result, an iterative procedure has to be carried out for every point in the brute skeleton, by which its position converges to be equidistant to all cluster boundaries. Nevertheless, this procedure proved to be much more computationally costly than the selected method, and the degree to which these internal points filled the area within the cluster was not as controllable.

2.5 Re-determination of Cluster Labels based on 3D Skeleton Point Connectivity

2.5.1 Extrapolation to the 3D Case

In the previous steps of analysis, individual snapshots of two-dimensional particle positions at different levels of Z have been treated separately, as if they pertained to different scenarios. Nevertheless, it is worth recalling that this whole study has dealt with a particular three-dimensional volume of space, within which a set of parallel sheets, of a known thickness, have been generated. For each of these sheets, each at a different value of Z, the particles contained therein have been normally projected to the mid-plane of the sheet, thus transforming a large three-dimensional domain into a number of adjacent two-dimensional domains. Therefore, although each of these sheets can be examined individually, it is necessary to connect the results occurring at different values of Z, to obtain an analysis which contemplates the whole three-dimensional domain.

As has been explained in Section 2.1, the way in which these parallel sheets for different levels of Z are generated greatly affects the adequacy with which the results of examination of a single two-dimensional domain can be extrapolated to the complete three-dimensional domain. On the one hand, one needs to employ a sheet thickness which is neither excessive as to allow for phenomena from disparate levels of Z to be depicted within a same Z value nor insufficient, thus resulting in a description of the particle structures under analysis which lacks detail. On the other hand, the separation between adjacent sheets has to be significant enough as to take advantage of the resulting computational efficiency with respect to a direct examination of the three-dimensional domain and at the same time low enough to permit a certain continuity between what adjacent two-dimensional domains represent.

Therefore, assuming that the selected parameters defining the generated parallel sheets are adequate, it is possible to make use of the existing continuity between adjacent two-dimensional domains in order to make connections which give the results of this analysis an additional dimension. More precisely, once particle clusters in different two-dimensional domains have been identified, it is possible to connect these according to their location and their similarity, thus recognizing that the two-dimensional clusters that appear at different values of Z are merely the intersection areas of the same three-dimensional particle structure with several planes of constant Z.

This introduction of an additional dimension to the cluster analysis within this study brings about the determination of multiple other parameters characterizing the cluster. Firstly, a new set of cluster labels can be defined based on this three-dimensional connectivity. Just as DBSCAN resulted in the grouping of particle positions into different particle clusters, each of these being associated to a particular label, it is now possible to assign a label to three-dimensional particle clusters based on whether connections between different planes of constant Z exist. Secondly, the boundary curves of different two-dimensional domains can be merged as long as they belong to the same three-dimensional particle cluster, thus creating a surface which describes the three-dimensional topology of the cluster. Lastly, given the area occupied by each particle cluster in a single two-dimensional domain, it is possible to approximately calculate the volume associated to each three-dimensional cluster.

Nevertheless, this step can be further separated into two separate procedures. The first one of these employs the previously developed condensation of cluster information into a cluster boundary and a constellation of inner points. Based on this simplification, the connectivities of brute skeleton points at different values of Z are examined, simply checking the intersection of a trajectory with cluster boundaries. Once the connections of each skeleton point have been determined, it is necessary to translate these results to the whole of the cluster particles. Therefore, the second procedure in this step deals with assigning the labels of three-dimensional clusters to the corresponding cluster particles in a coherent and efficient way, in order to express the information obtained in the previous procedure in the level of detail that employing the whole of cluster particles allows for.

2.5.2 Description of Employed Method

Connection of Neighboring Skeleton Points

Based on the condensation of a two-dimensional particle cluster into a set of interior points and a closed boundary curve, it is necessary to try to connect skeleton points in order to determine if they belong to the same three-dimensional particle structure. On the one hand, it is necessary to carry out the straight-forward grouping of brute skeleton points interior to the same two-dimensional cluster as belonging to the same three-dimensional cluster label. Although this procedure is performed with ease by the human eye, its computational implementation is not as simple, and relies greatly on how densely sampled the constellation of interior points is. On the other hand, it is necessary to determine if skeleton points in adjacent two-dimensional domains pertain to the same three-dimensional particle structure, this too depending greatly on the abundance of skeleton points within a cluster.

With the aim of introducing the core concepts based on which this procedure is carried out, a definition for directly connected points is presented.

Definition 2.5.1. *Directly Connected Points*: Given a pair of points r_1 and r_2 in $\mathbb{R}^2$, a set of two-dimensional points defining a boundary $\mathcal{B}$ in $\mathbb{R}^2$, a collision distance d_c, a step size d_s, and the points $t \in \mathcal{T}(r_1, r_2, d_s)$ defining the straight-line trajectory between r_1 and r_2 with step size d_s, r_1 and r_2 are directly connected with respect to d_c and d_s if none of the points $t \in \mathcal{T}(r_1, r_2, d_s)$ are within d_c of any boundary point $b \in \mathcal{B}$.

Based on the fact that this definition deals exclusively with two-dimensional points, it is expected for this condition to be examined within a single two-dimensional domain. More importantly, two brute skeleton points interior to the same two-dimensional cluster are said to belong to the same three-dimensional cluster as long as they are directly connected with respect to a collision distance d_c and a sampling distance d_s. In simple terms, one has to imagine a straight trajectory between two skeleton points, made up of uniformly separated trajectory points. Based on this trajectory, the proximity of each trajectory point with nearby boundary points is examined, such that if the trajectory is at any point too close to a cluster boundary, it is considered to collide with the boundary, thus resulting in the two points not being directly connected. This simple condition is the one determining whether two skeleton points within the same plane of constant Z belong to the same three-dimensional cluster.

Moreover, in order to determine whether brute skeleton points in adjacent two-dimensional domains belong to the same cluster, it is necessary to define the concept of projection-connectivity.

Definition 2.5.2. *Projection-Connectivity*: Given a pair of levels of Z, z_1 and z_2, each associated to a point r_1 and r_2 and a set of cluster boundaries $\mathcal{B}_1$ and $\mathcal{B}_2$, a collision distance d_c, and a step size d_s, r_1 is said to be projection-connected to r_2 with respect to d_c and d_s if its normal projection on the plane defined by $Z = z_2$, r_1', is directly connected with r_2 with respect to d_c and d_s.

The definition of projection-connected points deals with the fact that the two points are not included within the same two-dimensional domain, and as a result one of the two points is normally projected onto the plane of constant Z of the other point, in order to examine whether a straight trajectory from the projected point to the other point collides with any cluster boundary. Thus, one has to essentially imagine the same straight trajectory between two points as before, but this time one of the two points to be connected corresponds instead to the projection of a point originally existing in an adjacent level of Z.

It is important to bear in mind that the condition of projection-connectivity between two points is not symmetric, since the cluster boundaries at both two-dimensional domains are not the same. In other words, r_1 may not be projection-connected to r_2 while r_2 is projection-connected to r_1. This is due to the fact that since the cluster boundaries at different levels of Z are different, the projection of r_2 can be directly connected with r_1 while the trajectory between the projection of r_1 and r_2 collides with a cluster boundary. This is presented in Figure 2.45, where due to the different cluster boundaries for different values of Z, the projection of r_2 is directly connected with r_1, and thus r_2 is projection-connected to r_1 but r_1 is not projection-connected to r_2.

Bearing this in mind, the implemented method establishes that if for two brute skeleton points at adjacent planes of constant Z, one is projection-connected with respect to the other, both can be considered to form part of the same three-dimensional cluster, and can thus be associated to the same cluster label. Thus, for each of these skeleton points, its connections

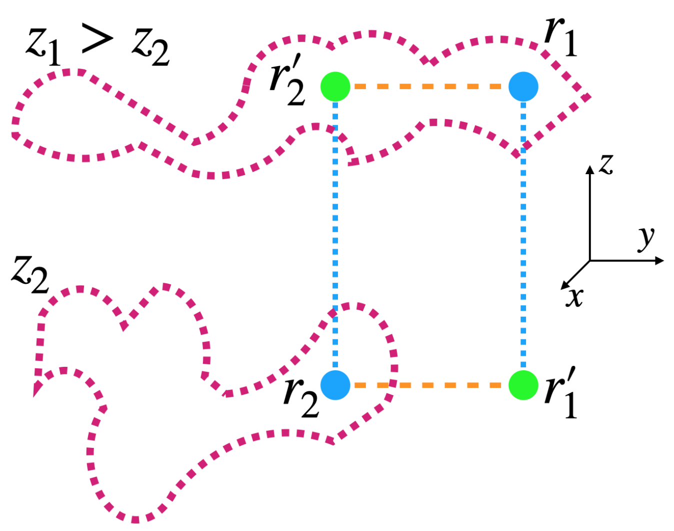

Figure 2.45: Projection-Connectivity is not Symmetric - In pink: cluster boundaries, in blue: original points, in green: projected points, in orange: straight trajectories.

are simply determined by observing whether a straight trajectory connecting either the point or its projections onto adjacent Z levels with other brute skeleton points collides with cluster boundaries or not.

As a result, the current study determines connectivities between skeleton points as follows. For each previously generated skeleton point, its three-dimensional position as well as those of its projections into the adjacent levels of Z are stored. For each of these (at most) three stored positions, all brute skeleton points within a certain radius of relevance and in the same level of Z are extracted. Then, straight trajectories of a known step size between each stored position and the relevant skeleton points are generated, and it is observed whether any of these trajectories gets closer than a specified collision distance to any cluster boundary within the same two-dimensional domain. Of these trajectories, those that connect the original brute skeleton point or its projection with another nearby skeleton point without colliding with a cluster boundary are detected, and the connections from the original brute skeleton point to other skeleton points which they allow for are stored. After this, the algorithm goes on to process the next skeleton point. Once all brute skeleton points have been analyzed, it is ensured that the connections between different skeleton points are made mutual, since it may occur that while point s_j is included in the connections of s_i, since projection-connectivity is not a symmetric condition, point s_i is not included in the connections of s_j. Therefore, an additional sweep of the list of connections is required to avoid this.

Once this initial procedure has been carried out, it follows to assign a cluster label to each of the processed skeleton points, in order to ensure that brute skeleton points deemed as connected are associated to the same cluster label, as they belong to the same three-dimensional particle structure. To do so, a sweep over the list of skeleton point connections is performed, propagating a cluster label to connected points. Nevertheless, since it is sought for cluster labels defined in previous iterations of this labeling procedure to be conserved, a priority is given to cluster labels according to their antiquity. Therefore, if when propagating a label among connected skeleton points, one of the connected points has already been assigned a label with priority over the current label, all of the skeleton points over which the current label has been propagated are now associated to the label with priority. In this way, it is ensured that the performed cluster labeling is coherent.

Based on the described procedure, the number of necessary parameters defining the performance of the method is easily derivable. On the one hand, as is visible just by the definition of directly connected points, it is necessary to define both a collision distance d_c and a step size d_s. Moreover, since only a neighborhood of relevant skeleton points is taken into account for each skeleton point, it is also necessary to introduce a radius d_b delimiting this neighborhood.

With regards to the neighborhood radius with which to select the nearby relevant skeleton points, it is worth noting that since the objective of this determination of connections between skeleton points is the assignment of cluster memberships, it is not necessary to connect every brute skeleton point with as many other skeleton points as possible. As long as these points within the cluster are abundant enough, it will be possible to unite significantly separated skeleton points under the same cluster label by means of other brute skeleton points in between. Thus, the selection of this distance will not have a great effect on the resulting cluster labels assigned to each skeleton point, although it will affect the number of brute skeleton points with which each skeleton point is connected, as well as the processing time associated to the algorithm. In the current implementation, it has been sought to define this parameter as proportional to the Kolmogorov length scale η, in order to relate such distance with a characteristic length scale of the flow turbulence. Thus, if this neighborhood distance is defined as $d_b = k_b \cdot \eta$, the current implementation has found $k_b = 105$ to be a valid selection.

Moreover, the collision distance d_c and the step size defining the trajectory between points d_s can be seen as closely related distance parameters, both defining the severity with which the collision criterion of a trajectory is imposed. On the one hand, larger values of d_s allow for a milder computational cost of the method, while as smaller step sizes ensure a more exact examination of trajectory collisions. On the other hand, the greater the value of d_c the stricter the collision criterion is, since a trajectory will have to cross farther by a cluster boundary in order to consider that it has collided. Moreover, the ratio between both is also relevant, since it may occur that if the step size d_s is more than twice the collision distance d_c, the trajectory actually jumps over a cluster boundary particle without triggering a collision. In any case, the current study has implemented this method with $d_s = 0.4 \cdot \eta$ and $d_c = 2 \cdot \eta$ with relative success.

As a final note, it is worth noting that a main limitation of this method in determining connectivities between skeleton points is the fact that the generated trajectories are straight. As a result, the method depends greatly on the abundance of brute skeleton points within the cluster to properly connect these points, since while as very disparate points may not be directly connected due to the abrupt topology of the cluster, a set of intermediate skeleton points can allow the method to relate them. In fact, the greater the number of skeleton points within a cluster, the more severe the imposed collision condition can be, since the task of connecting brute skeleton points in separated areas of the cluster is eased.

Assignment of Three-Dimensional Cluster Labels to Cluster Particles

Once the connectivities between skeleton points have been determined, and a set of three-dimensional cluster labels have been assigned, what follows is to apply this labeling scheme to the cluster particles based on which each brute skeleton was generated. This is performed with great ease once it is known that skeleton points within the same two-dimensional cluster have the same cluster label and that cluster particles within the same DBSCAN cluster have the same label.

As a result of these two parting assumptions, the core principle of this procedure is to substitute the DBSCAN label of all cluster particles within the same two-dimensional cluster by the three-dimensional cluster label of a skeleton point within such cluster. More precisely, for each skeleton point, its closest cluster particle is extracted. Then, all of the cluster particles within the same level of Z with the DBSCAN label of such closest cluster particle have the label of the skeleton points assigned to them. This procedure is repeated for all skeleton points.

Moreover, if it is not satisfied that skeleton points within the same DBSCAN cluster are associated to the same three-dimensional cluster label due to a deficient examination of connectivities, this method is prepared to define separate cluster labels within the same two-dimensional cluster. In simple terms, each cluster particle will be associated to the cluster label of the brute skeleton point which is closest to it. Nevertheless, it is assumed that the abundance of skeleton points within every cluster and the collision criteria employed within this implementation stop this from happening in the first place.

2.5.3 Results

Connection of Neighboring Skeleton Points

What results from this procedure, part of it visible in Figures 2.46, 2.47 and 2.48, is that most of the two-dimensional clusters in the whole domain are connected with the same three-dimensional cluster label, such that according to the implemented method the connection of most cluster particles into a single three-dimensional cluster is adequate. On the other hand, several small clusters are identified, intersecting a small number of two-dimensional domains in Z.

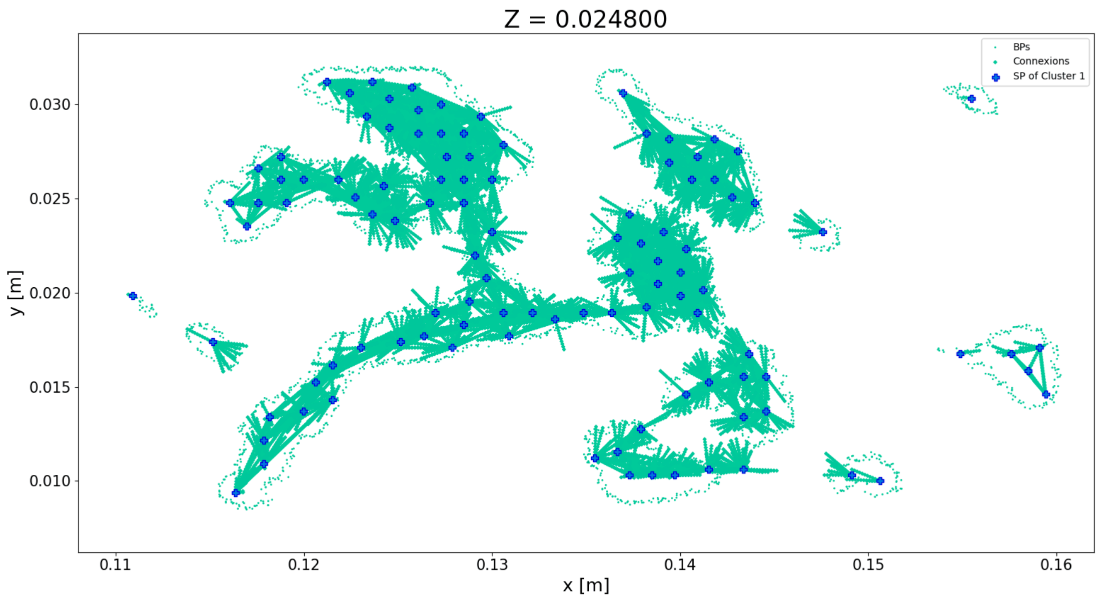

Figure 2.46: Connectivity and Labeling of Skeleton Points - $Z = 0.0248$ m

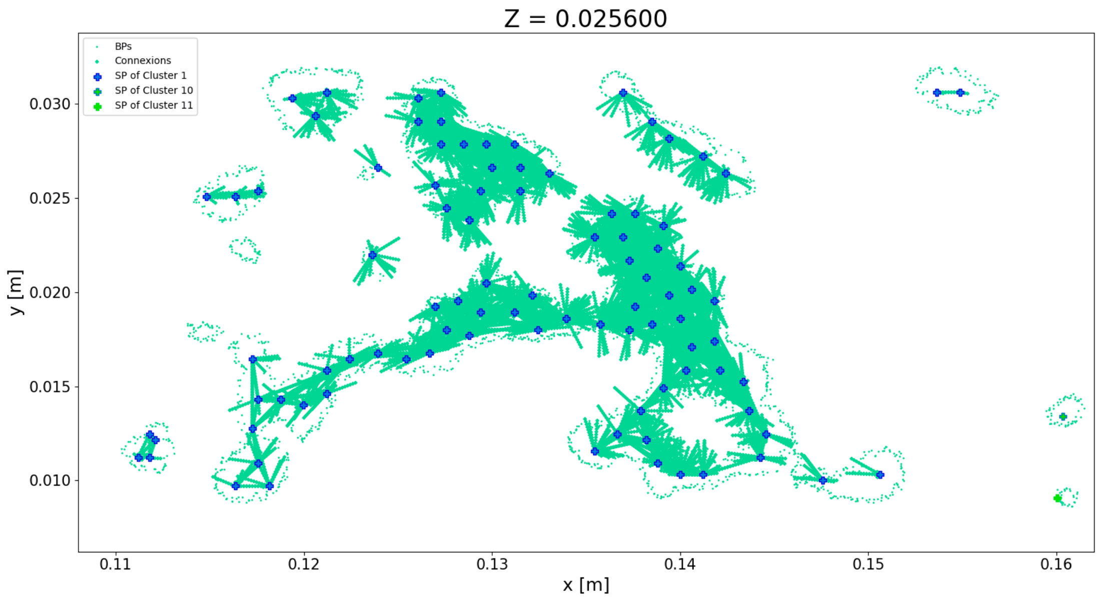

Figure 2.47: Connectivity and Labeling of Skeleton Points - $Z = 0.0256$ m

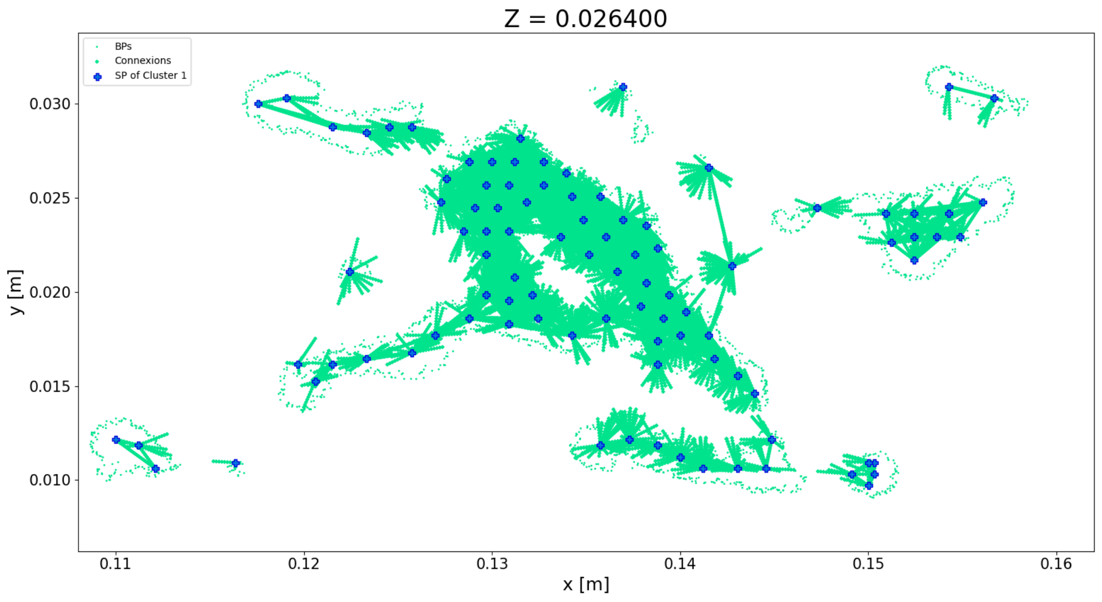

Figure 2.48: Connectivity and Labeling of Skeleton Points - $Z = 0.0264$ m

2.5.4 Attempted Alternatives

Given that once the implemented method relied on straight trajectories between skeleton points in order to connect them, a strong dependency of the method's performance with the abundance of skeleton points existed, an alternative method was attempted in order to avoid this dependency, by developing instead trajectories between skeleton points which instead of being systematically straight, curve themselves in order to avoid cluster boundaries. This way, the cluster's topology can more adequately be taken into account when connecting brute skeleton points within a cluster, since the generated trajectory is adapted to the obstaculizing cluster boundaries.

Similarly to attempted alternatives in previous steps of analysis within this study, it was sought to train an agent capable of tracing a trajectory between points within the cluster, but in this case, the agent was to function in a two-dimensional circuit between two skeleton points and enclosed by the corresponding cluster boundary curves. Moreover, there was still a simulation of the agent's operation within the circuit to be carried out, at every time step of which the agent deciding upon an action based on its current state. This state, as in the similar method presented in Section 2.4.5, was the input of a neural network in charge of the selection of the action to select in the current time step, and it was precisely this neural network which was trained during its execution.

The main difference in this method with respect to the attempted alternative for describing a skeleton curve of the three-dimensional cluster lies in the way in which the agent's decision making was trained. Instead of a costly genetic algorithm, reinforcement learning was applied by assuming the agent's simulation to constitute a Markov decision process. In this process, the set of possible actions to take by the agent is defined as a set of known angular deviations from its current course, and the state of the agent is defined by the distances to the closest cluster boundaries in each direction, its current course, the course of the straight trajectory towards its target, and the distance to its target. Moreover, a reward function is defined, which assigns a score to each possible state within the circuit. Then, the agent's training has the objective of defining a policy that determines the action which, based on the agent's current state, maximizes its expected future reward.

More precisely, Deep Q-Learning was applied, such that the decision-making core of the agent was made up of a neural network which defines $Q(s, a)$, a function describing, for a current state s and an action at such state a, the associated expected future reward. In this way, with an already trained neural network, it is expected that introducing the current state into such function will allow the agent to determine which action is to maximize its expected future reward.

Furthermore, special care had to be taken with the definition of a viable reward function which ensured that a maximization of the expected future reward meant taking the agent to the circuit's target. This reward function was made up of three components: one in charge of driving the agent towards the target at all points of the domain, one in charge of attracting the agent towards the target when particularly close to it, and a last term repelling the agent from areas of the circuit close to cluster boundaries. The first of these components follows Equation (2.5), where $\vec{r}$ represents the agent's current position, $\vec{t}$ is the location of the circuit's target, and $\vec{o}$ the location of the circuit's origin. The reward contour along the domain which results from this component of the reward function is presented in Figure 2.49, basically driving the agent away from the origin of the circuit and towards the target.

$$R_1(\vec{r}, \vec{t}, \vec{o}) = \min(\frac{(\vec{r} - \vec{o}) \cdot (\vec{t} - \vec{o})}{|\vec{t} - \vec{o}|^2}, 1) + \min(\frac{(\vec{t} - \vec{r}) \cdot (\vec{t} - \vec{o})}{|\vec{t} - \vec{o}|^2}, 0) \tag{2.5}$$

Moreover, a second reward function centered at the circuit's target was defined, as exposed in Equation (2.6), where $\vec{r}$ corresponds to the agent's current location and $\vec{t}$ is the location of the circuit's target.

$$R_2(\vec{r}, \vec{t}) = \frac{1}{(|\vec{r} - \vec{t}|)^2} \tag{2.6}$$

Furthermore, the third component in this reward function can be contemplated more pre-

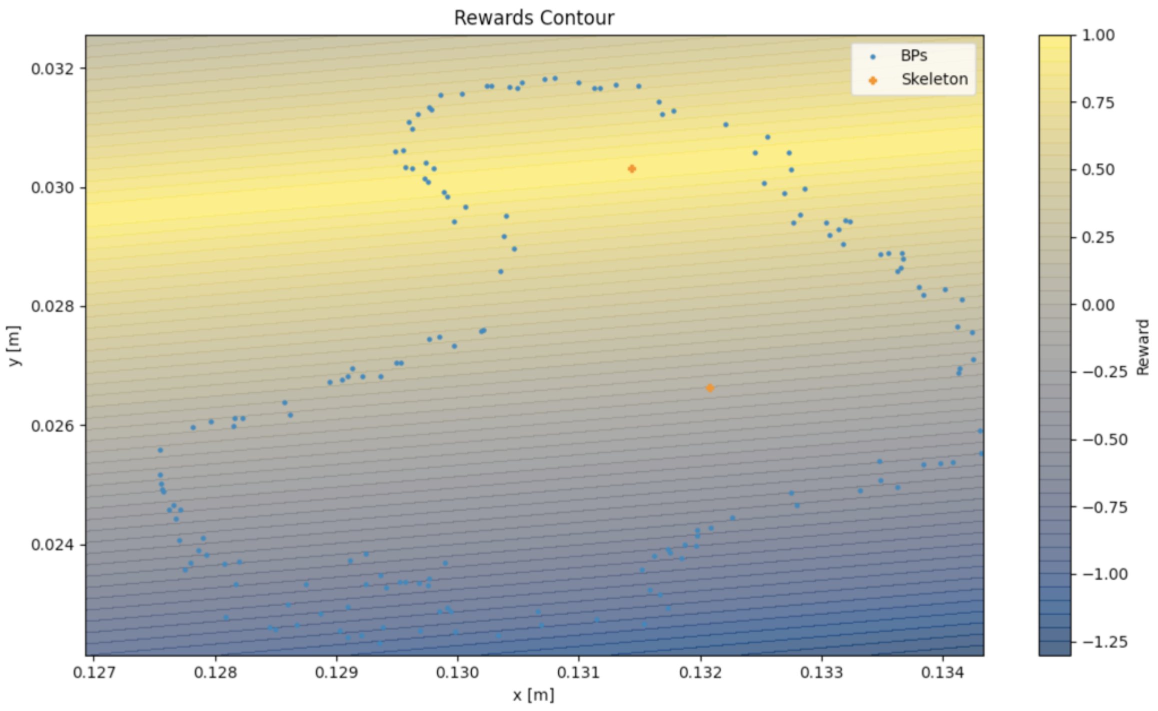

Figure 2.49: Contours of the Reward Function defined only by R_1

cisely as a penalization, to be activated when the agent is closer than d_{crash} to any cluster boundary particle. This penalization term is presented in Equation (2.7), where d is the distance to the closest cluster boundary particle and b is the maximum penalization which can be applied. As a result, one has to conceive the employed reward function as a weighted sum of $R_1(\vec{r}, \vec{t}, \vec{o})$ and $R_2(\vec{r}, \vec{t})$, where the weights have been previously tuned accordingly and where a penalization of $P(d)$ is applied when the agent is exceedingly close to any boundary particle.

$$P(d) = \frac{-b}{(2d_{crash})^2} \cdot d^2 + b \tag{2.7}$$

Bearing in mind the main objective of the implemented training procedure as well as the structure of the reward function involved within this training, the training procedure is as follows. Based on an initialized neural network for $Q(s, a)$ and a set of circuits described by pairs of skeleton points and their enclosing cluster boundaries, the agent's operation is simulated for each of these circuits. For each time step of each of these simulations, the agent is allowed to act based on the action which maximizes $Q(s, a)$ for its current state s. Based on its state s, this action a, and the resulting state s', the associated reward $R(s, a, s')$ is computed, and a loss term is computed as shown in Equation (2.8). In this expression, γ corresponds to the discount factor, determining the importance in $Q(s, a)$ of the possible rewards of the next step, and $\mathcal{A}$ is the set of possible actions. Then, with the objective of minimizing $L(s, a, s')$, the neural network defining $Q(s,a)$ is updated. This update is thus carried out for every time step of every simulation, until the resulting loss terms are sufficiently low to assume that training has finished.

$$L(s,a,s') = \Big(Q(s,a) - \big(R(s,a,s') + \gamma \max_{a' \in \mathcal{A}} Q(s',a')\big)\Big)^2 \tag{2.8}$$

What resulted from training this neural network based on a selection of circuits and on the implemented reward function was an agent capable of tracing a trajectory between skeleton points which properly avoided cluster boundaries, thus taking into account the cluster's topography when defining the connectivity between clusters. This avoidance of cluster boundaries is particularly visible in Figures 2.50 and 2.51, where a straight trajectory would not be capable of uniting both skeleton point.

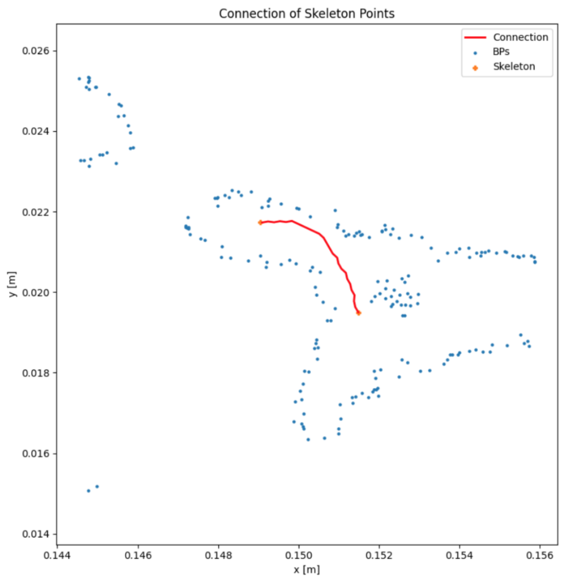

Figure 2.50: Trajectory between Skeleton Points as Traced by an Agent Trained via Deep Q-Learning - In blue: cluster boundary particles, in orange: skeleton points, in red: traced trajectory

Nevertheless, the main flaw of this implementation is that once the agent was trained with a given set of pairs of skeleton points, it was unable to trace a valid trajectory between skeleton points outside of this training set. This required that the application of this method to the current study would imply the training of one agent for every group of pairs of skeleton points within the dataset. Since the number of possible pairs of skeleton points in the domain of interest is significant and training of an agent was relatively costly, it was found that the improvement in the determination of skeleton point connectivities which this method allowed

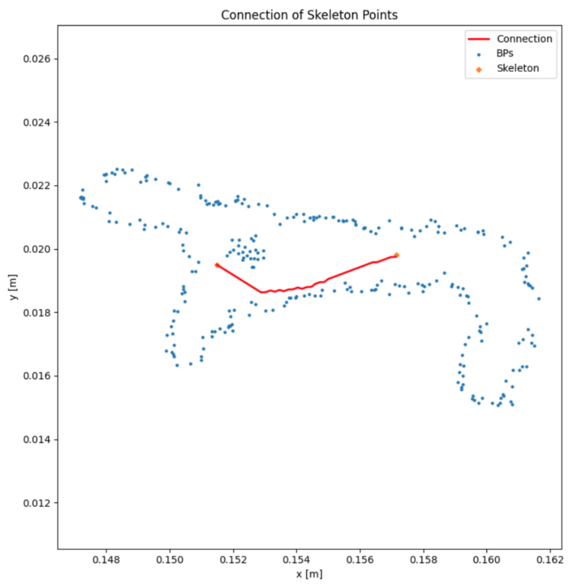

Figure 2.51: Trajectory between Skeleton Points as Traced by an Agent Trained via Deep Q-Learning - In blue: cluster boundary particles, in orange: skeleton points, in red: traced trajectory

for does not compensate its elevated complexity and computational cost.

2.6 Temporal Tracking of Cluster Topology

2.6.1 Introduction

Up to now, the current study has dealt with, parting from a dataset of particle positions sorted into a discrete number of levels of Z, classifying these particles into three-dimensional clusters whose topology is simplified in a boundary and a set of interior points. Nevertheless, it is also an objective of the current study to develop a method of temporal tracking of each cluster, such that by applying the previously described steps to two different datasets concerning the same spatial domain at different instants of time, one can connect the results of both clustering analyses in order to explore the evolution of each cluster's topology and volume over time.

One has to take into account however that this tracking is not as simple as following a set of clusters in time, assuming that the region of interest will always focus on the same set of clusters and that the number of clusters will always remain the same. On the one hand, if the spatial domain of analysis is fixed, it is reasonable to expect particle clusters to leave and enter the domain following the duct flow velocity, thus causing the appearance of previously unseen clusters as well as the loss of structures being tracked. On the other hand, one has to take into account that a cluster is merely defined as a set of particles in a region of relatively higher number density. Therefore, it is very important to bear in mind that new clusters may appear due to a sudden increase in the number density of a region within the domain of interest, or that existing clusters can cease to exist due to a dissociation of its member particles. In any case, it is still possible, albeit the possible birth and death of clusters, to pair clusters from one time instant with clusters of another, thus allowing for the desired tracking of particle clusters.

In the current study, a robust method of temporal tracking of cluster topologies has been developed, based on the pairing of clusters from different time instants. In the following, the implemented method is described, an adequate selection of its defining parameters is presented, and its associated results are discussed.

2.6.2 Description of Employed Method

As has been mentioned, the temporal tracking of particle clusters is at its core based on the pairing of clusters from adjacent time instants, in order to examine the temporal evolution of the cluster's characteristics. However, to pair particle clusters belonging to two adjacent time instants, it is convenient to employ a simplified version of these clusters, rather than the complete set of its member particles. Therefore, the current implementation makes use of the skeleton points expressing the cluster's topology to carry out this connection over time. More precisely, this method of temporal tracking is directly based on pairing skeleton points of the previous time instant with skeleton points of the next time instant. In order to do this, it is assumed that after the time step under consideration a cluster is not significantly deformed, and that the relative position of its associated skeleton points has not varied too much. As a result, the pairing of a skeleton point of the first time instant is simply reduced to finding the skeleton point of the second time instant which, appearing in a region in which a particle moving from the first skeleton point with the duct flow velocity would be expected to appear at the second time instant, is neighbor to the cluster boundary that is most similar to the cluster boundary neighboring the skeleton point of the first time instant.

When defining a region in which it is expected to find a skeleton point with which to connect the skeleton point under analysis, it is necessary to begin with the assumption of negligibly deformed clusters, in which the distribution of its internal skeleton points remains more or less unchanged. This region is constructed around the concept of a particle which, parting from the position of the skeleton point under analysis, moves in the direction of the duct flow with its associated bulk velocity. In fact, this region is defined employing a known radius around such expected position. Note that since the dataset utilized within this study focuses exclusively on the regions of the duct which are sufficiently separated from its walls, the assumption of a constant flow velocity along the domain under analysis is sensible (See [17] for more information on the axial velocity profile of the flow).

As a result, all skeleton points of the second time instant which are considered for the connection with the skeleton point under analysis are within a radius d_{focus} of $r_{expected}$, the latter being presented in Equation (2.9) taking into account the position of the skeleton point to be paired r_0, the time step Δt, and the bulk velocity v_{bulk}. This region of interest is further clarified in Figure 2.52. It is worth noting that in the current implementation, this region has been intentionally limited to the level of Z of the original skeleton point, such that it will only be paired to skeleton points of its same level of Z.

$$r_{expected} = r_0 + \Delta t \cdot v_{bulk} \tag{2.9}$$

Once a region of interest associated to a skeleton point of the first time instant has been defined, based on which one can define a group of candidate skeleton points of the second time instant, it is possible to rank these skeleton points based on the similitude of the topology of their neighboring cluster boundaries with that of the original skeleton point. In simple terms, one has to look at the boundary particles within a radius r_{sens} of each skeleton point, and determine, for a number *dirs* of directions parting from the skeleton point, at what distance will the closest cluster boundary be found. In order to do this, the neighborhood of each skeleton point is divided into a set of "slices", and for each of these, the distance to the closest boundary particle contained within such "slice" is extracted. Note that since only the neighboring boundary particles of the same level of Z as the skeleton point are analyzed, this neighborhood is a two-dimensional region of radius r_{sens}, as is visible in Figure 2.53.

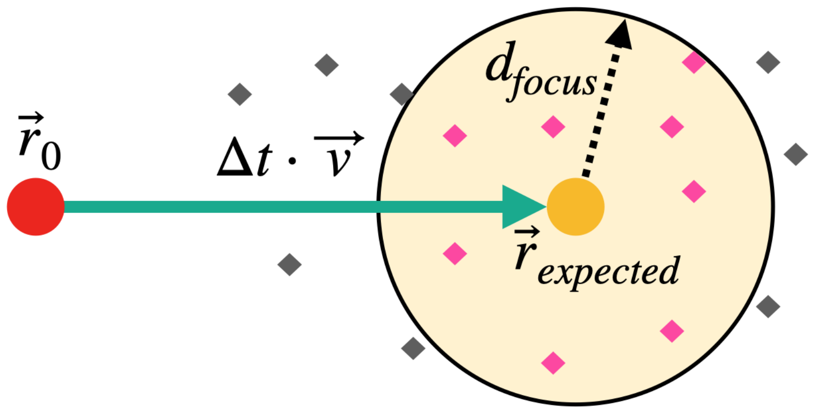

Figure 2.52: Region of Interest for the Pairing of a Skeleton Point - In red, position of the skeleton point under analysis. In yellow, expected position assuming uniform velocity in the direction of the duct flow. In pink, skeleton points within the region of interest, defined by a radius d_{focus} around $r_{expected}$. In gray, skeleton point outside of such region of interest.

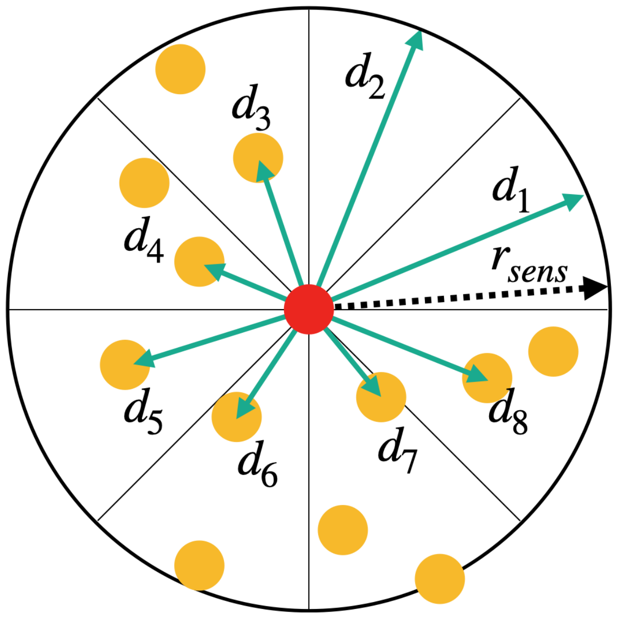

Figure 2.53: Distance Sensing for a Skeleton Point for $dirs = 8$ - In red, the skeleton point whose r_{sens}-neighboring boundary particles are examined. In yellow, the neighboring boundary particles. In blue, the distances measured to the closest boundary particles within each "slice".

Based on the $dirs$ distances measured by means of this distance sensing for each skeleton point, one can define a vector $\vec{d}$ of $dirs$ elements for each skeleton point defining its measured distances, such that its i-th component represents the distance measured in its i-th direction. If this vector is seen as descriptive of the topology of the cluster boundaries close to a skeleton point, in order to select an adequate skeleton point of the second time instant within the specified region of interest, it is only necessary to determine which candidate skeleton point is associated to a vector $\vec{d}$ of the least Euclidean distance with respect to the distance vector of the skeleton point of the first time instant.

In summary, to pair a skeleton point of the first time instant, one has first to extract all skeleton points of the second time instant within the aforementioned region of interest. Then, taking into account the vector of distances to neighboring boundary particles of the original skeleton point, $\vec{d_0}$, one has only to determine the extracted skeleton point whose distance vector is most similar to $\vec{d_0}$. This skeleton point will be the one with which the original skeleton point will be paired. It is nevertheless necessary to take into account that it may occur that no skeleton points may appear within the defined region of interest, such that the skeleton point of the first time instant will remain unpaired.

Based on this fundamental pairing of skeleton points of different time instants, one can carry out the pairing of whole clusters by connecting their member skeleton points. In the implemented method, the distances to neighboring cluster boundary particles are first measured for each skeleton point, for both time instants taken into account. Then, for each skeleton point of the first time instant, the skeleton points of the next time instant within its associated region of interest are extracted, and their distance vector $\vec{d}$ is compared, to select the skeleton point with which to pair the skeleton point under analysis. This is carried out for all skeleton points of the first time instant. Hence, one can pair particle clusters by analyzing, for each particle cluster of the first time frame, to which new cluster label are most of its skeleton points assigned.

What results is a method with which one can connect the results regarding a particular cluster at a specific instant of time with the same results associated to another cluster at a later time, and even determine what parts of an existing cluster have continued within the same cluster, broken away to create or join another cluster, or even have ceased to be part of a particle cluster. In order to take into account a larger number of time instants, one only has to apply the presented method for each pair of adjacent time frames.

2.6.3 Adequate Selection of Parameters

Based on the previously described method of temporal tracking of cluster topology, it is possible to discern four parameters affecting the operation of the algorithm. Firstly, in order to define the region of the domain containing candidate skeleton points with which to pair each skeleton point under analysis, a radius d_{focus} has to be specified. Secondly, the employed method of expressing the topology of cluster boundaries neighboring each skeleton point needs the definition of a number of directions $dirs$ to examine around each skeleton point, as well as a neighborhood radius r_{sens} enclosing the boundary particles under analysis. Lastly, it is also necessary to specify the velocity of the flow within the duct, since this parameter will directly affect the location of the region of interest for each skeleton point under analysis.

When specifying the radius determining the size of the region of interest associated to each skeleton point to be paired, one has to bear in mind to what extent the assumption of clusters moving in the direction of the duct flow without deforming is true. In simple terms, the size of this region determines the extent to which a cluster moving differently from the duct flow velocity will be captured by the implemented method. If this radius d_{focus} is greatly reduced, the region in which to look for candidate skeleton points of the second time frame is made smaller around $r_{expected}$. Therefore, this parameter will significantly affect the percentage of skeleton points for which no pair skeleton point in the next instant of time is found. Nevertheless, this distance could be increased in excess such that the skeleton point of most similar neighboring cluster boundary topology within the defined region is not what the human eye would associate with the skeleton point under analysis, thus resulting to an inaccurate pairing of skeleton points across time instants.

On the other hand, taking into account that the flow velocity within the channel is subjected to turbulent fluctuations with respect to the mean velocity, it is possible to define d_{focus} based on two sources of deviation from the expected position $r_{expected}$. On the one hand, d_{focus} should take into account the fluctuating particle velocities due to this turbulence, by including the distance which a particle moving at such fluctuating velocity would travel during the specified timestep. Based on Esmaily et al. (2020) [17], where the flow conditions are the same as in the simulation from which the employed dataset is obtained, it is possible to approximate this fluctuating particle velocity in all directions to 0.1 of the bulk speed v_{bulk}. On the other hand, the region of interest employed to pair each skeleton point should also take into account the fact that skeleton points do not exactly maintain their relative position within the same cluster, therefore requiring an additional margin of uncertainty added to the original region of interest defined only by the fluctuating particle velocity. Since it is nonetheless expected for the skeleton point to remain within the boundaries of the cluster itself, this second source of uncertainty can be expressed, for each skeleton point, as a function of the average of all distances to the closest boundary particles to such point in each direction. Therefore, every skeleton point will have a different radius defining such region of interest, which will be dependent on the average of all the distances measured by means of the previously described distance sensing scheme.

As a result of these two sources, $d_{focus,i}$ in the current implementation follows the decomposition presented in Equation (2.10), where $u_{p,rms}$ is the fluctuating particle velocity, v_{bulk} is the simulation bulk velocity, $\overline{\vec{d_i}}$ is the mean of all distances to the closest neighboring boundary particles in each direction, and k_{focus} is a constant defining the margin of uncertainty given to the region of interest due to the movement of skeleton points within the cluster.

$$d_{focus,i} = \frac{u_{p,rms}}{v_{bulk}} \cdot v_{bulk} \cdot \Delta t + k_{focus} \cdot \overline{\vec{d_i}} \tag{2.10}$$

What results from modifying this parameter is visible in Figure 2.54, where the cluster boundaries of a cluster at a single level of Z for two relatively close instants of time are presented along with their connected skeleton points. On the one hand, As k_{focus} decreases, it is possible to see that the pairing of skeleton points more closely follows the hypothesis of clusters moving with negligible deformation, such that the trajectory connecting paired skeleton points is more and more aligned with the direction of the flow velocity. In fact, if for the complete dataset, one measures the angle of each of these connecting trajectories with the duct axial direction, one can see that as k_{focus} decreases from 1.5 to 1.0 to 0.5, the average of this angle also decreases from 3.45 to 2.93 to 0.47 degrees, for two time instants separated 0.15 ms. On the other hand, for the same pair of time instants, as k_{focus} is decreased, the percentage of unpaired skeleton points increases, from 5.47% with $k_{focus} = 1.5$ to 39.53% with $k_{focus} = 0.5$. This is particularly notorious in Figure 2.54, where the number of unpaired skeleton points increases significantly as k_{focus} is reduced.

From a more conceptual point of view, an increase of k_{focus} beyond 1 implies an extension of the region of interest beyond the cluster boundaries. For instance, parting from an ideal spherical cluster whose boundary curve within a two-dimensional plane is a circumference, if this cluster were small enough to contain only one skeleton point at its center and move in the direction of the flow without deforming, it would be expected for the relative position of a new skeleton point within the same cluster in the next time step to at least remain within the cluster boundary curve. Therefore, even with a translation of the cluster taking into account $u_{p,rms}$ in any direction, the new skeleton point must nonetheless be contained within the cluster. Based on this reasoning and on the apt results obtained with this value of k_{focus}, this parameter is set to 1.

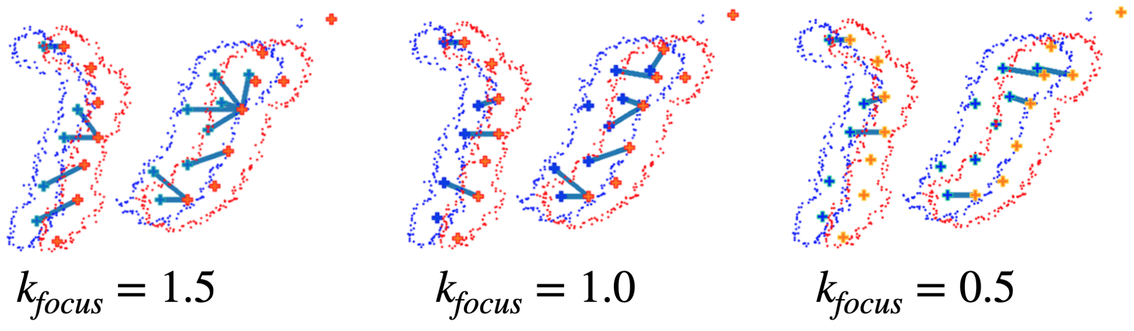

Figure 2.54: Evolution of Skeleton Point Pairing with k_{focus} - In blue points, old cluster boundary particles. In red points, new cluster boundary particles. In blue crosses, old skeleton points. In red crosses, new skeleton points. In blue lines, connections between paired skeleton points. For all three cases, $\Delta t = 0.15$ ms, $v = 7.7$ m/s, $k_{sens} = 3$, and $dirs = 10$. From left to right, k_{focus} decreases from 1.5 to 1.0 to 0.5.

Moreover, another distance parameter has to be specified for the implemented method of temporal tracking of particle clusters. As when one measures, for each skeleton point, the distance along every direction to neighboring cluster boundary particles, it is convenient to extract only the closest cluster boundaries in order to cheapen this procedure, it is necessary to impose a certain neighborhood radius r_{sens} defining the region from which to extract these relevant boundary particles. Furthermore, in order to ensure that the vector of distances regarding each skeleton point properly expresses the topology of the cluster boundaries around it, this radius must be aligned with a characteristic cluster size. For values of r_{sens} significantly smaller than this characteristic size, many of the boundary particles neighboring a skeleton point may not be enclosed within this neighborhood, such that this characteristic topology of a specific area of the cluster will not be captured. On the other hand, if r_{sens} is increased well beyond this characteristic size, since only the closest boundary particles in each direction are extracted, the algorithm will incur a greater computational cost than necessary by sampling an excessively large region of the domain. Thus, there will be a value for this radius at which a further increase does not affect the results significantly, but will only enlarge the associated computational cost.

In order to correlate r_{sens} with a characteristic size of the obtained clusters, this parameter was defined as proportional to the Kolmogorov length scale of the flow η. Then, if $r_{sens} = k_{sens} \cdot \eta$, after examining the effect of k_{sens} on the computational cost of the method as well as on the obtained results, $k_{sens} = 63$ was conceived as an apt configuration.

On the other hand, the choice of the number of directions $dirs$ about a skeleton point to take into account when carrying out the distance sensing of each skeleton point is related with the desired resolution when expressing the neighboring cluster topology in a vector of distances. The greater this number, the greater the amount of detail of the cluster boundary to be encapsulated in this vector. However, for excessively high numbers of divisions, it may occur that the random irregularities existing in a cluster boundary are also encapsulated in this vector of distances. In any case, for the current implementation, it was considered that $dirs = 10$ accurately expressed all cluster boundary shapes, without being excessively affected by noise existing in these descriptions of cluster boundaries by means of a set of points.

Lastly, the duct flow velocity defining the separation of the center of the region of interest associated to a skeleton point with respect to such skeleton point can be specified by several ways. If regions relatively close to the duct walls were to be analyzed, it would be necessary to model the existing average velocity profile, due to large variations of this convective velocity from the duct walls to the duct's center. However, since the implemented method deals with a spatial domain at which the average velocity profile does not vary significantly, and since the region of interest is already generated by means of a generous radius around $r_{expected}$, this duct flow velocity can be assumed constant and equal to the flow bulk velocity, which for the current dataset is 7.7 m/s.

2.6.4 Results

Once an appropriate set of parameters defining the method's performance have been selected, it follows to present its operation on adjacent instants of time of varying time step. When analyzing the results of this implementation on two datasets separated by a time interval of 0.15 ms, one obtains an apt tracking of cluster topology for a relatively straight-forward case. In Figures 2.55, 2.56, and 2.57, one can see how skeleton points from adjacent time instants are connected, for $Z = 0.0128$ m, $Z = 0.0136$ m, and $Z = 0.0144$ m, respectively. It is also possible to see the trajectories connecting paired skeleton points are as a whole fairly aligned with the direction of duct flow velocity, the average angle of such trajectories with the X axis being 2.92 degrees.

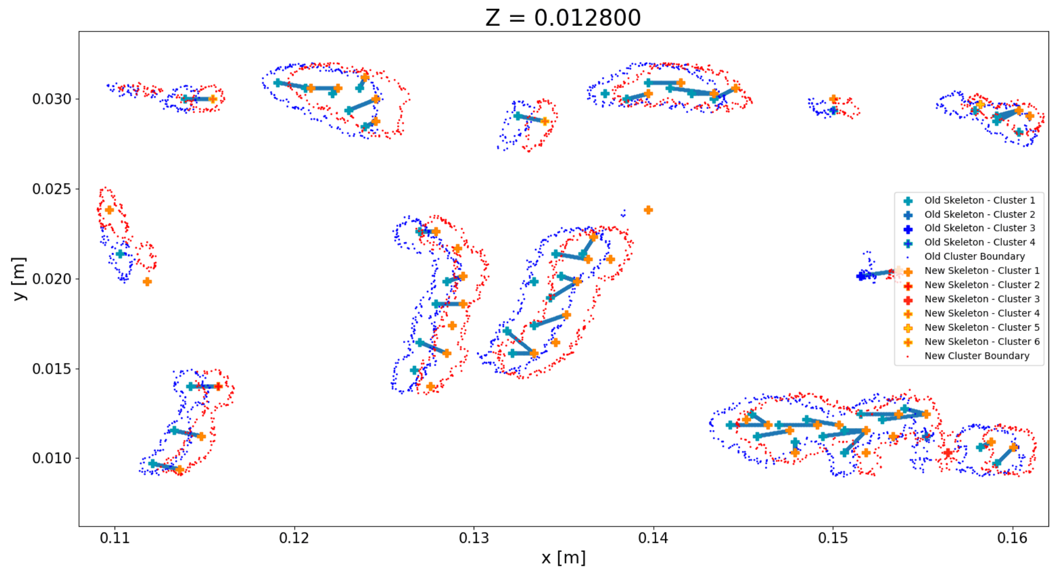

Figure 2.55: Temporal Tracking of Cluster Topology for $Z = 0.0128$ m - In blue points, old cluster boundary particles. In red points, new cluster boundary particles. In blue crosses, old skeleton points. In red crosses, new skeleton points. In blue lines, connections between paired skeleton points. $k_{focus} = 1$, $\Delta t = 0.15$ ms, $v = 7.7$ m/s, $k_{sens} = 3$, and $dirs = 10$

Moreover, one can also track, for each cluster label assigned to skeleton points in the first time instant, what amount of skeleton points is paired with skeleton points of each cluster label of the second time instant. An overview of this tracking is visible in Figure 2.58, where it is visible that the largest cluster of the first time instant is mostly conserved, a part of its skeleton point being however unpaired.

On the other hand, if the time step is raised to 0.75 ms, the complexity of the pairing

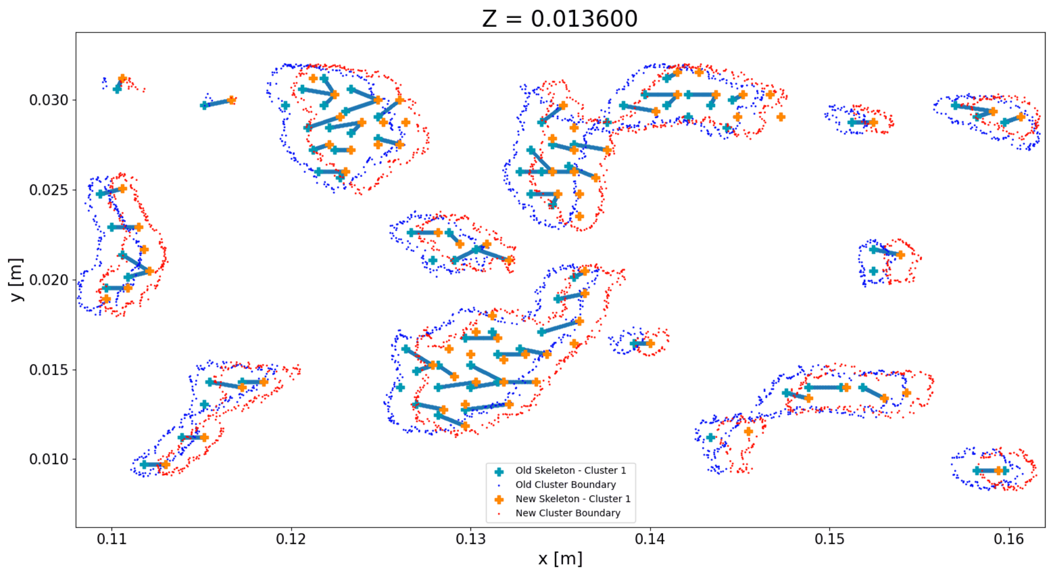

Figure 2.56: Temporal Tracking of Cluster Topology for $Z = 0.0136$ m - In blue points, old cluster boundary particles. In red points, new cluster boundary particles. In blue crosses, old skeleton points. In red crosses, new skeleton points. In blue lines, connections between paired skeleton points. $k_{focus} = 1$, $\Delta t = 0.15$ ms, $v = 7.7$ m/s, $k_{sens} = 3$, and $dirs = 10$

procedure is greatly increased. As is visible in Figures Figures 2.55, 2.56, and 2.57, presenting the connection of skeleton points of adjacent time instants for $Z = 0.0128$ m, $Z = 0.0136$ m, and $Z = 0.0144$ m, respectively, clusters of both instants of time are much more separated and deformed, such that the assumption of clusters moving without deforming is not as applicable. In any case, the implemented method of temporal tracking associated pairs of skeleton points in a reasonable way, such that the mean orientation of connecting trajectories between pairs of skeleton points is of 0.62 degrees with the direction of duct flow.

Moreover, it is possible to observe that many of the skeleton points of the first time instant which remain unpaired are close to the most downstream area of the domain, such that the region of interest in which to look for candidate skeleton points of the next time instant is outside of the domain of interest. Furthermore, it is also worth noting that some skeleton points of the first time instant belonging to very small clusters are often paired with skeleton points of doubtful similarity. This can be explained by the reduced notoriety of these smaller clusters, their particles possibly dissociating and destroying the particle structure before the next time instant.

For the same time step of 0.75 ms between two snapshots, one can visualize the pairing of skeleton points classified by their cluster labels in Figure 2.62. Curiously enough, when comparing these results with the analogously presented in Figure 2.58 for a fifth of the time step, one can see that the percentage of unpaired skeleton points from the first time instant

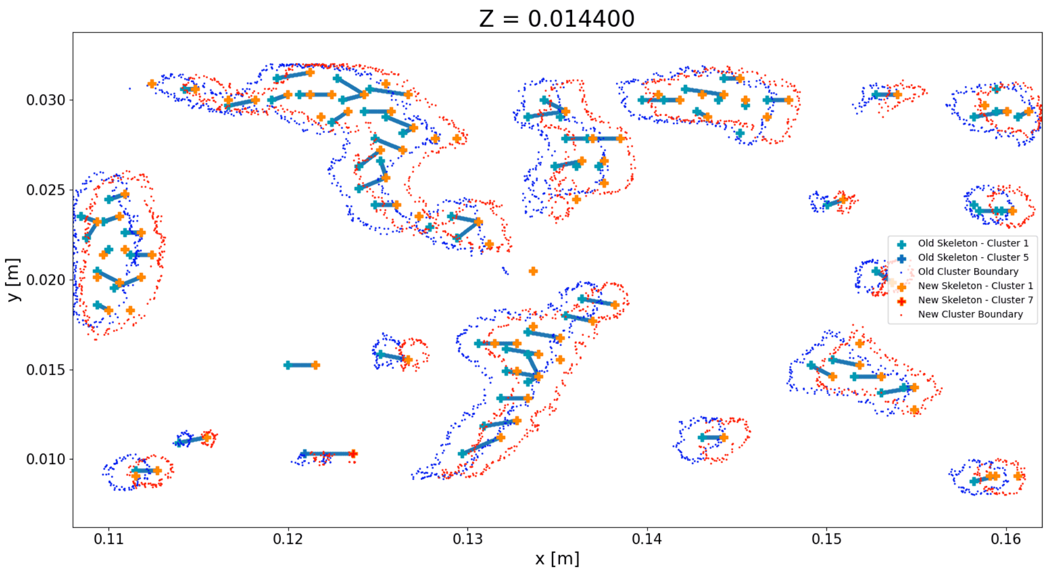

Figure 2.57: Temporal Tracking of Cluster Topology for $Z = 0.0144$ m - In blue points, old cluster boundary particles. In red points, new cluster boundary particles. In blue crosses, old skeleton points. In red crosses, new skeleton points. In blue lines, connections between paired skeleton points. $k_{focus} = 1$, $\Delta t = 0.15$ ms, $v = 7.7$ m/s, $k_{sens} = 3$, and $dirs = 10$

is greater once the time step is increased, rising from 12.56% for $\Delta t = 0.15$ ms to 25.30% for $\Delta t = 0.75$ ms. This is probably due to the fact that the hypothesis of particle structures moving exclusively in the direction of flow velocity without deformation is less applicable as the time step between both instants of time increases, such that the region of interest defined for each skeleton point of the first time instant is not as effective.

Furthermore, given an appropriate tracking of clusters for two adjacent time instants, it is also possible to extend this analysis to take into account several time frames. As has been mentioned, by connecting particle clusters, one can determine how the properties of a particular cluster evolve with time. In Figure 2.63, the tracked volume and number of member skeleton points of the largest cluster in the domain are presented, resulting, for a relatively small interval of time, a small variation in these parameters. Note that given that most cluster particles are associated to a single predominant cluster, the evolution of the largest cluster will generally provide a greater amount of insight into a cluster's evolution.

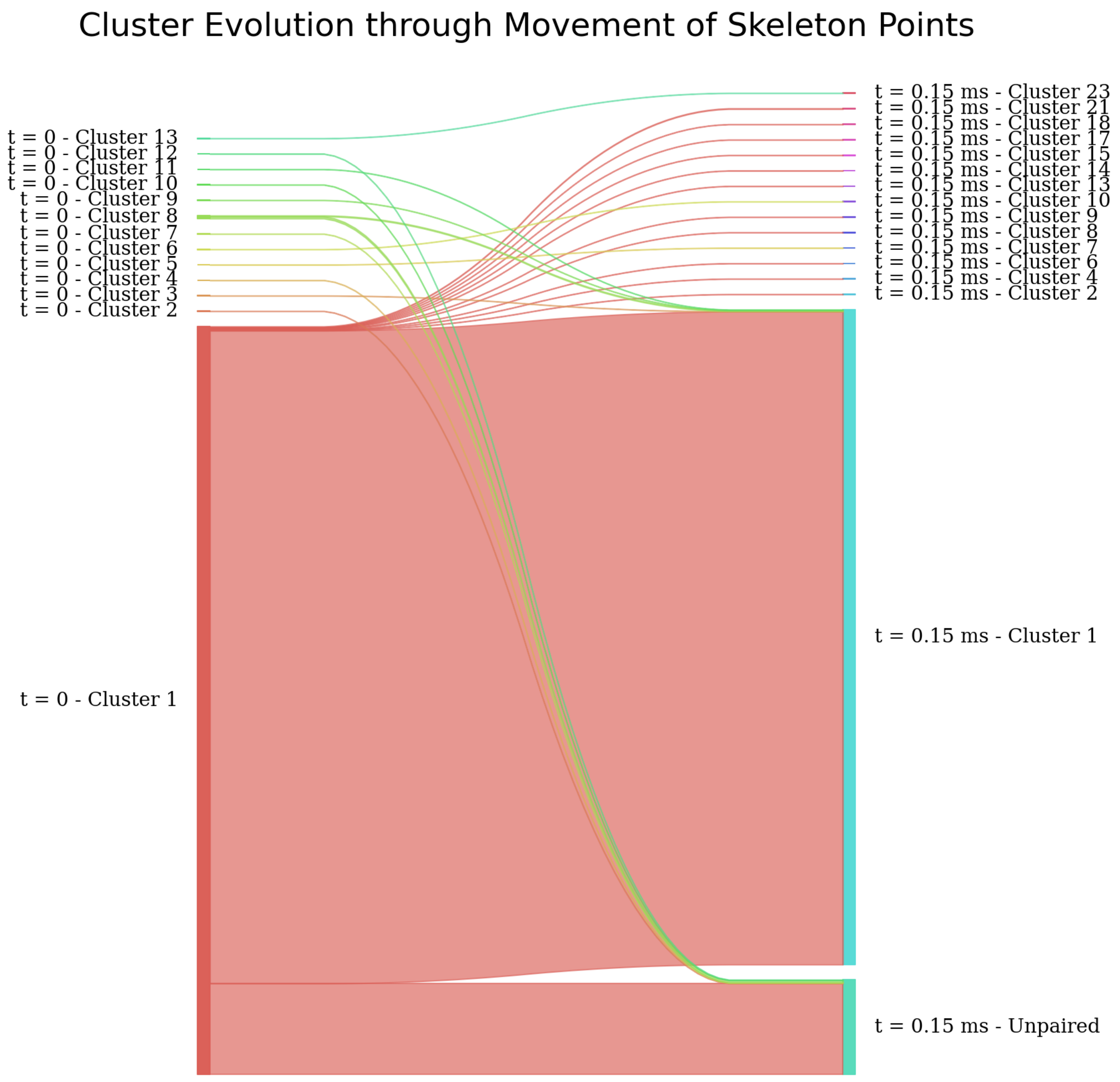

Figure 2.58: Pairing of Skeleton Points across Time Instants - $k_{focus} = 1$, $\Delta t = 0.15$ ms, $v = 7.7$ m/s, $k_{sens} = 3$, and $dirs = 10$

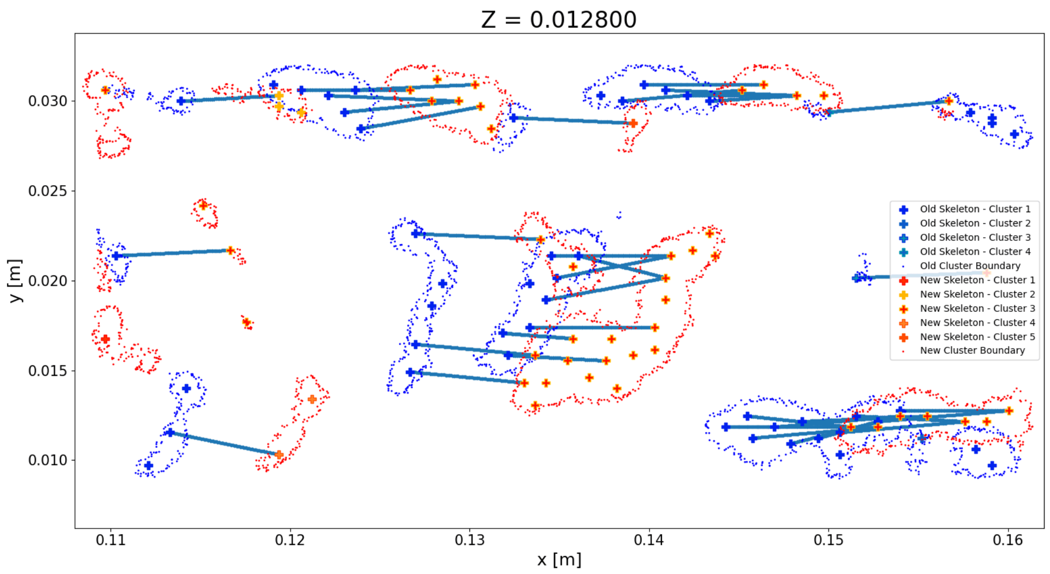

Figure 2.59: Temporal Tracking of Cluster Topology for $Z = 0.0128$ m - In blue points, old cluster boundary particles. In red points, new cluster boundary particles. In blue crosses, old skeleton points. In red crosses, new skeleton points. In blue lines, connections between paired skeleton points. $k_{focus} = 1$, $\Delta t = 0.75$ ms, $v = 7.7$ m/s, $k_{sens} = 3$, and $dirs = 10$

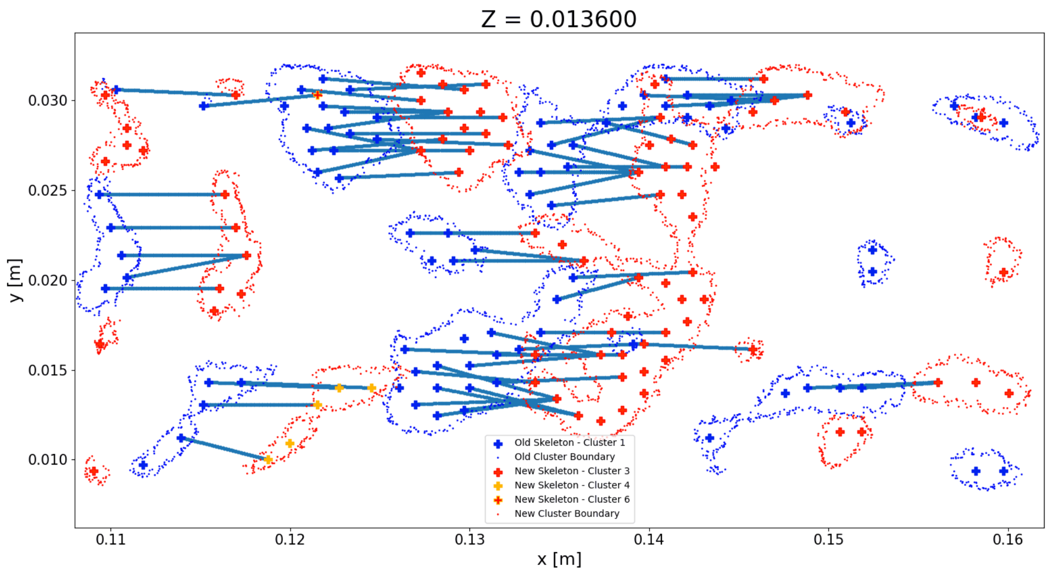

Figure 2.60: Temporal Tracking of Cluster Topology for $Z = 0.0136$ m - In blue points, old cluster boundary particles. In red points, new cluster boundary particles. In blue crosses, old skeleton points. In red crosses, new skeleton points. In blue lines, connections between paired skeleton points. $k_{focus} = 1$, $\Delta t = 0.75$ ms, $v = 7.7$ m/s, $k_{sens} = 3$, and $dirs = 10$

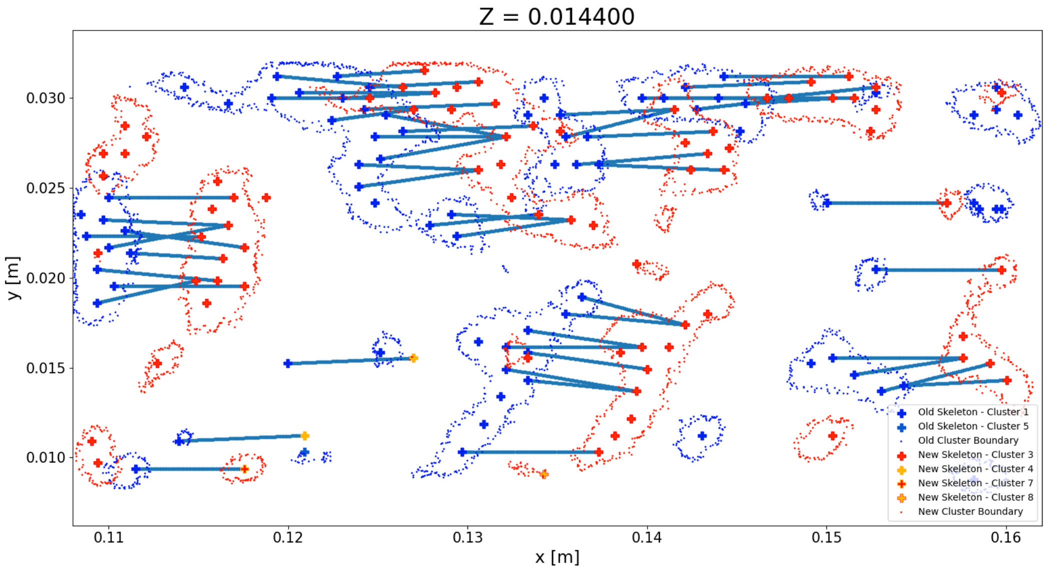

Figure 2.61: Temporal Tracking of Cluster Topology for $Z = 0.0144$ m - In blue points, old cluster boundary particles. In red points, new cluster boundary particles. In blue crosses, old skeleton points. In red crosses, new skeleton points. In blue lines, connections between paired skeleton points. $k_{focus} = 1$, $\Delta t = 0.75$ ms, $v = 7.7$ m/s, $k_{sens} = 3$, and $dirs = 10$

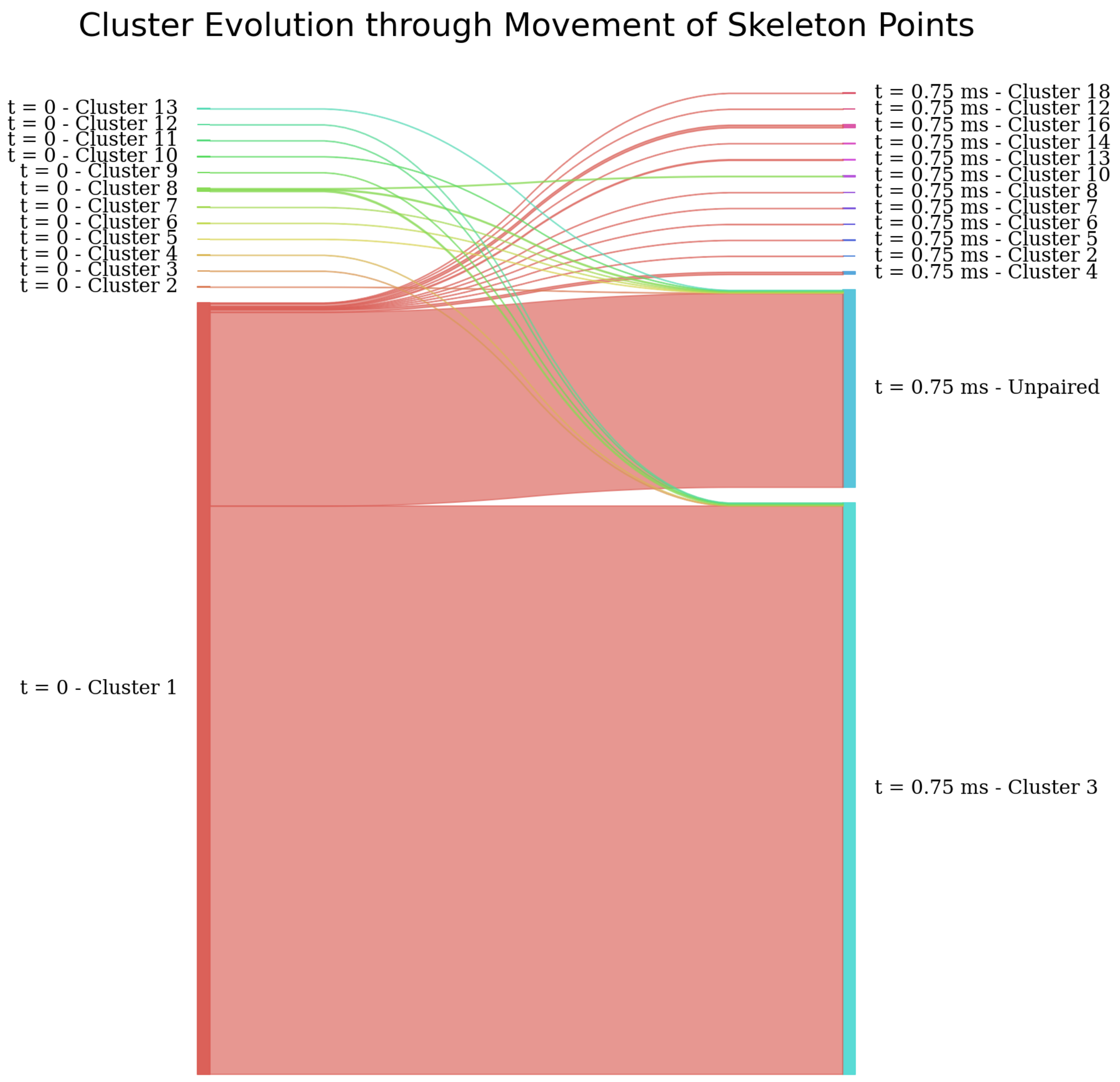

Figure 2.62: Pairing of Skeleton Points across Time Instants - $k_{focus} = 1$, $\Delta t = 0.75$ ms, $v = 7.7$ m/s, $k_{sens} = 3$, and $dirs = 10$

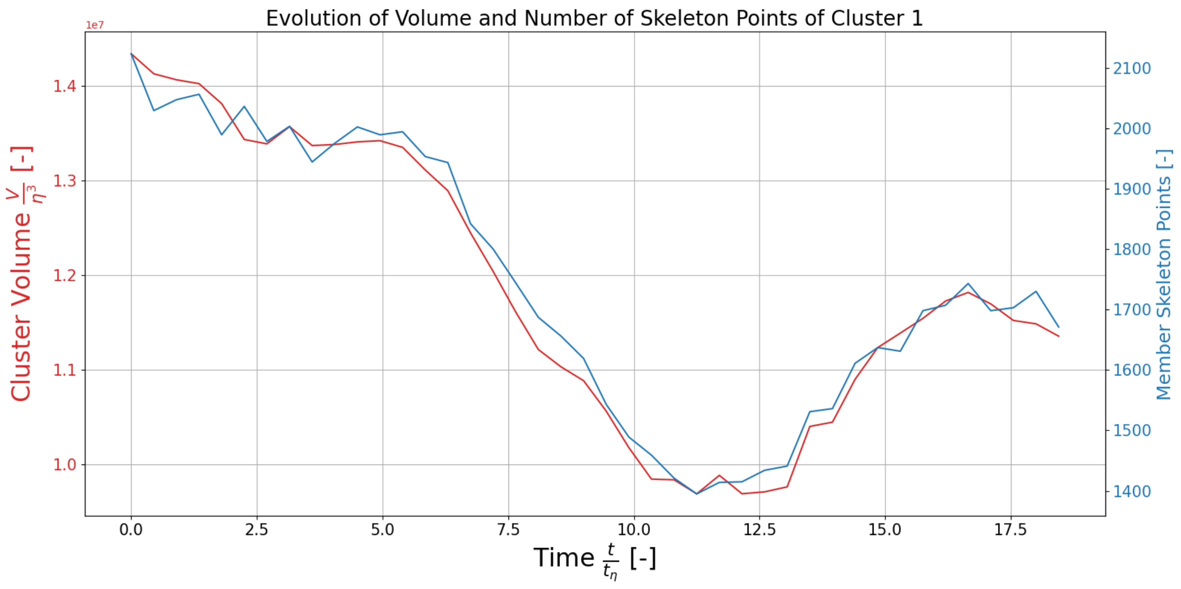

Figure 2.63: Evolution of Volume and Number of Skeleton Points of Cluster 1 - $k_{focus} = 0.5$, $\Delta t = 0.75$ ms, $v = 7.7$ m/s, $k_{sens} = 3$, and $dirs = 10$

Chapter 3

Evaluation based on Voronoi Tessellations Analysis

3.1 Application of Voronoi Tessellations to cluster Analysis

In order to evaluate the results of the presented methods of particle cluster analysis, it is necessary to employ an affordable method which is as unbiased as possible from the physical parameters of the problem. In other words, a method analyzing the preferential concentration of particles in a way which does not require the setting of an arbitrary length scale nor a density threshold.

For instance, a common method of cluster identification and characterization is box-counting. This method divides the domain under analysis into boxes of a specified size and proceeds to count the number of particles contained within each box. This method has been employed on the one hand to quantify the degree of preferential concentration in the particle-laden flow for a particular length scale, by comparing the probability density function regarding the number of particles per box in the preferentially concentrated case to the case in which particle positions are uniformly distributed, the latter case being associated to a Poisson probability density function [2]. On the other hand, box-counting can be employed to define a concentration field of the domain, based on which connected boxes containing a number of particles superior to a specified threshold describe particle clusters and connected empty boxes at some scale define particle voids [4]. Nevertheless, box-counting methods present a dependency on an arbitrary, *a priori* length scale defining the box size, which requires the tuning of an additional parameter to the peculiarities of the current problem [7].

On the other hand, identifying and characterizing particle clusters with the Voronoï analysis of the set of particle positions directly avoids the introduction of an arbitrary length scale, since the density threshold based on which a cluster label is assigned to a particle depends exclusively on the comparision with the equivalent case of uniformly distributed particles [7]. This Voronoï analysis determines, for each of the particle positions in the dataset, the region of the domain that is closer to such particle than to any other. In this way, one obtains a set of tessellations corresponding to each of these regions, the volume of each of them being inversely proportional to the local concentration of particles at the center of such volume [7].

Based on the Voronoï tesselation of the dataset, assigning a Voronoï volume to each of the particles in the domain, it follows to determine the probability density function describing how such volumes are distributed when normalized with the mean Voronoï volume in the problem. One can similarly perform the same Voronoï analysis on a dataset of the same size but populated with uniformly distributed particles, as per a random Poisson process, and obtain another probability density function representing the distribution of Voronoï volumes in this case [7]. By definition, preferential concentration of particles within the particle-laden flow will cause particles to cluster significantly more than in the randomly distributed case, such that very small and very large Voronoï volumes will be expected to occur more frequently than in the case following a random Poisson process. Therefore, the comparison of the probability density functions regarding the normalized Voronoï volumes of the preferentially concentrated case with that of the randomly distributed case will result in two intersection points, since for very high and very small volumes the preferentially concentrated case will present a higher probability, and for intermediate volumes the scenario with uniformly distributed particles will present a higher density function [7].

In this way, the identification and characterization of clusters following this Voronoï analysis is directly founded on the intersections between both of the aforementioned probability density functions. Particles whose associated normalized Voronoï volume is smaller than the volume corresponding to the first intersection are labeled as cluster particles, while as particles for which the resulting normalized Voronoï volume is larger than the second intersection between density functions are classified as void particles [7]. It is worth noting that the particles whose assigned normalized volume is between both intersections are classified as neither of the two. Moreover, the characterization of clusters is simply performed once connectivity between cluster tessellations is examined, in order to define a region of the domain as a single particle cluster based on the volumes of the adjacent particles labeled as clusters within that region.

Therefore, it is convenient to employ this method of analysis in order to properly evaluate the techniques developed within this study, since there is no requirement of setting an arbitrary length scale for the identification and characterization of cluster structures, and also since the implementation of such technique of analysis is relatively straight-forward taking into account that multiple libraries offer the possibility of performing the Voronoï tessellation of a set of points.

In order to adequately take into account the multiple levels of cluster analysis which the techniques developed within this study allow for in this evaluation, this chapter presents the evaluation of the clustering analysis of a single two-dimensional snapshot in Z as well as in the resulting clustering analysis of the whole three-dimensional domain. Moreover, this chapter also deals with the comparison of computational performances of both methods.

It is important to note that the current study carried out all Voronoï tessellations by making use of the existing Python library *SciPy*.

3.2 Evaluation of the Clustering Analysis of a 2D Snapshot

The evaluation of the developed method in its clustering analysis of an individual two-dimensional snapshot in Z is carried out by comparing, for each of the particles within the domain, whether the assigned label coincides which the label which a two-dimensional cluster identification based on Voronoï tessellations assigns. More precisely, once the previously exposed DBSCAN clustering analysis has classified each of the particles in the domain as belonging to a particle cluster or as belonging to a particle void, one can also carry out the Voronoï tesselation of the same two-dimensional domain containing projected particle positions. In this case, each Voronoï cell will have an associated area. Hence, by comparing the resulting normalized area probability density function with that of randomly distributed particles, one can determine if particles classified as cluster particles by the previous DBSCAN analysis have an associated Voronoï area which is smaller than the occurring threshold.

When comparing the probability density function of normalized Voronoï areas of the preferentially concentrated case with that resulting from randomly distributing particle positions as per a random Poisson process, it is convenient to take into account that there exists an analytical expression of such density function in the case of uniformly distributed particles in a two-dimensional domain [21]. This expression, presented in (3.1), simplifies the computation time of this two-dimensional analysis, since it does not require the simulation of multiple random configurations of particles. In this equation, $a = 1.0787$, $b = 3.0328$, $c = 3.3095$, $\Gamma(\cdot)$ corresponds to the gamma function, and $A/\langle A\rangle$ represents a normalized Voronoï area.

$$P(A/\langle A\rangle) = \frac{a \cdot b^{\frac{c}{a}}}{\Gamma(c/a)} \cdot (A/\langle A\rangle)^{c-1} e^{-b(A/\langle A\rangle)^{a}} \tag{3.1}$$

In this way, the determination of the intersection points between the probability density function of the preferentially concentrated set of particles and that of the random Poisson process is simplified, such that the Voronoï analysis of the original set of particles suffices. In 3.1, the resulting probability density function of the preferentially concentrated case is visualized, and its first intersection with the density function of the uniformly distributed set of particles is displayed. As a result, particles with an associated normalized Voronoï area smaller than the value of this intersection will be classified as cluster particles by this method of cluster identification based on Voronoï tesselations.

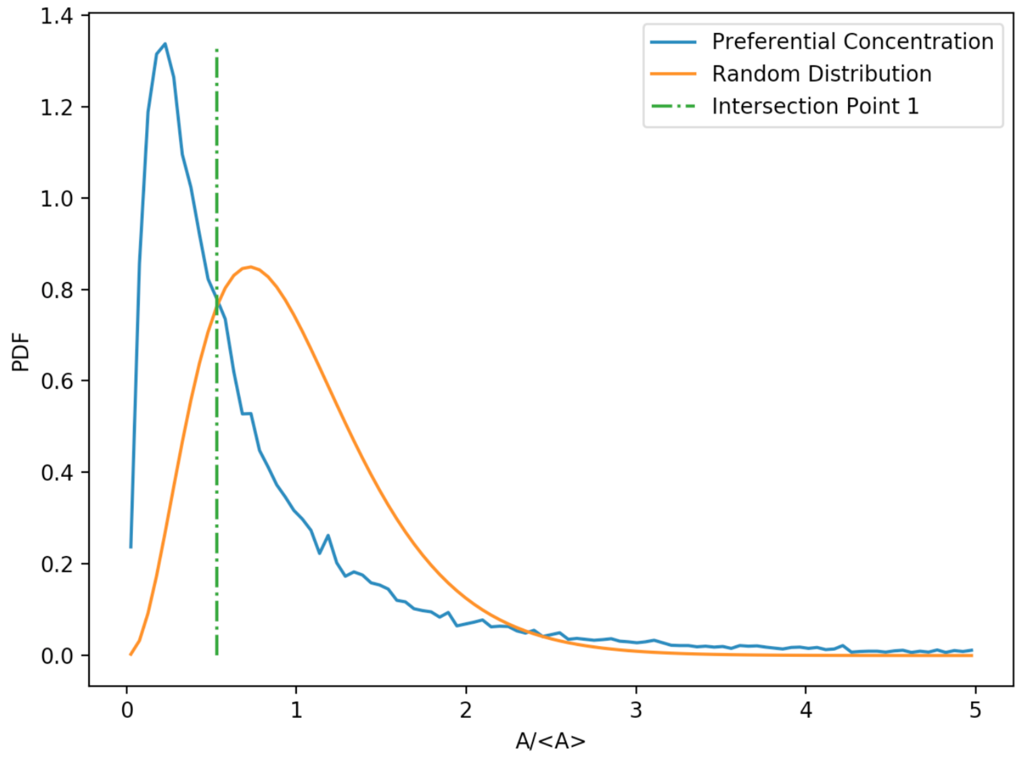

Figure 3.1: Intersection of Normalized Area Probability Density Functions - In blue, density function corresponding to the preferentially concentrated case. In orange, density function following a random Poisson process.

If this Voronoï analysis is applied to a single two-dimensional snapshot in Z of the dataset employed within this study, one obtains, for $Z = 0.02$ m, what is presented in Figure 3.2. Here, particles with an associated normalized Voronoï area smaller than that of the intersection displayed in Figure 3.2 are labeled as cluster particles, with a gray marker. All other particles are presented with a red marker. Given that the clustering analysis based on DBSCAN which this study implements exclusively differentiates between cluster and void particles, for the purpose of this evaluation it is assumed that all particles not classified as cluster particles by this Voronoï analysis are classified as void particles.

As can be expected, in Figure 3.2, particles in relatively denser regions of the domain are classified as belonging to a cluster. What is remarkable from this particular method of cluster analysis is that the setting of a threshold density for the definition of a cluster is carried out independently of any arbitrary length scale. Furthermore, it is necessary to note that particles

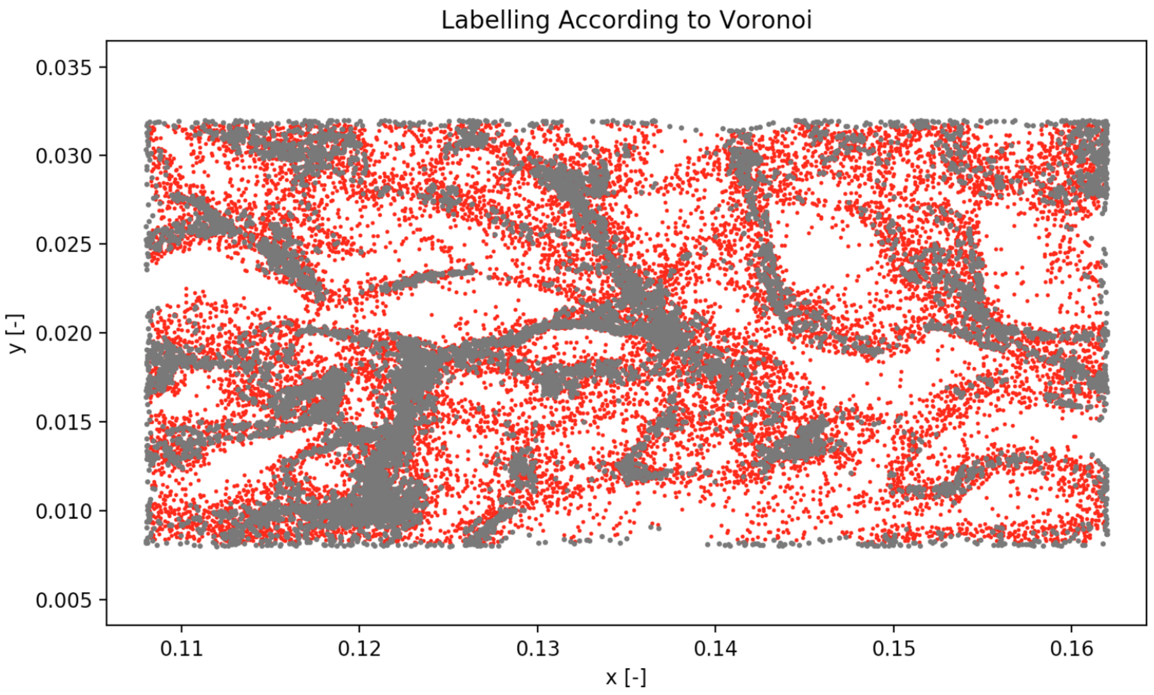

Figure 3.2: Identification of Cluster Particles Based on Voronoï Tesselation of a Single Two-dimensional Snapshot in $Z = 0.02$ m - In gray, particles labeled as cluster particles. In red, all other particles.

in the boundaries of the domain have been classified as cluster particles due to the fact that their volume, for the purposes of this calculation, has been set to null. Nevertheless, these particles are not considered for the later validation of the clustering techniques implemented within this study.

If one were to compare the results displayed in Figure 3.2 with the cluster labeling which the application of DBSCAN produces for the same snapshot in Z, each of the non-boundary particles in the domain would fall into one of the following four categories. In the first place, the particle can be classified as a cluster particle both by the application of DBSCAN of this study and by the Voronoï-based approach of this evaluation. Moreover, it could also be that both cluster identification techniques coincide in classifying the given particle as a void particle. Nevertheless, in the case that both techniques were to disagree in their labeling of the particle, it could occur that the Voronoï-based approach classifies the particle as a cluster particle while the DBSCAN-based approach of the current study labels the particle as a void particle, or that the opposite occurs. Based on these four groups of particles, it is possible to calculate the accuracy of the cluster identification method of the current study by computing the percentage of non-boundary particles in the domain for which both labeling techniques coincide.

If the method of clustering analysis developed within the context of this study is evaluated, for the same snapshot in Z as the one analyzed in Figure 3.2, with the same thickness of the projection sheet, one obtains an accuracy of 72.57%. The results of this evaluation are presented in Figure 3.3. Based on this image, it is possible to assert that the clusters obtained by means of the application of DBSCAN in this study results in clusters of a more significant thickness, within the context of a two-dimensional snapshot in Z. By looking closely at Figure 3.3, one can see that the main source of discrepancy between different clustering analysis techniques is the fact either the DBSCAN-based approach identifies thicker clusters than what the Voronoï-based technique results in, or that the latter identifies as clusters very thin structures that the former is incapable of detecting.

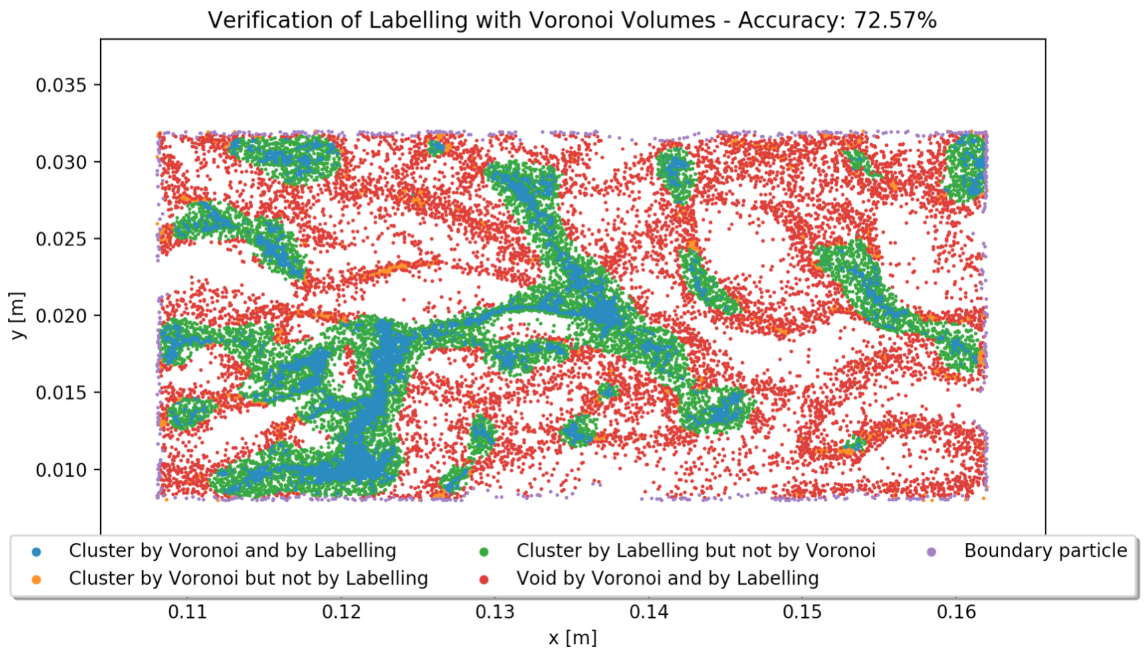

Figure 3.3: Evaluation of Cluster identification Based on Voronoï Tesselation of a Single Two-dimensional Snapshot in $Z = 0.02$ m and $t = 0.0004$ m - In blue, particles labeled as cluster particles by both methods. In red, particles labeled as void particles by both methods. In orange, particles labeled as cluster particles by the Voronoï-based approach but not by the DBSCAN-based approach. In green, particles labeled as cluster particles by the DBSCAN-based approach but not by the Voronoï-based approach. In purple, boundary particles.

If the thickness of the projection sheet for the snapshot in Z is to be halved to $t = 0.0002$ m, the same trend is seen to occur. As is natural, both of the parameters based on which the DBSCAN analysis of the two-dimensional domain is performed vary as the average number density of the plane is reduced from $23,472,994\ 1/m^2$ to $11,733,796\ 1/m^2$. More specifically, $MinPts^*$ is reduced to 130. The results, presented in Figure 3.4, again show that thin or small structures are not properly captured by the DBSCAN-based approach as clusters, and that the structures which the latter does classify as clusters are generally thicker. The resulting accuracy is similarly 70.83%.

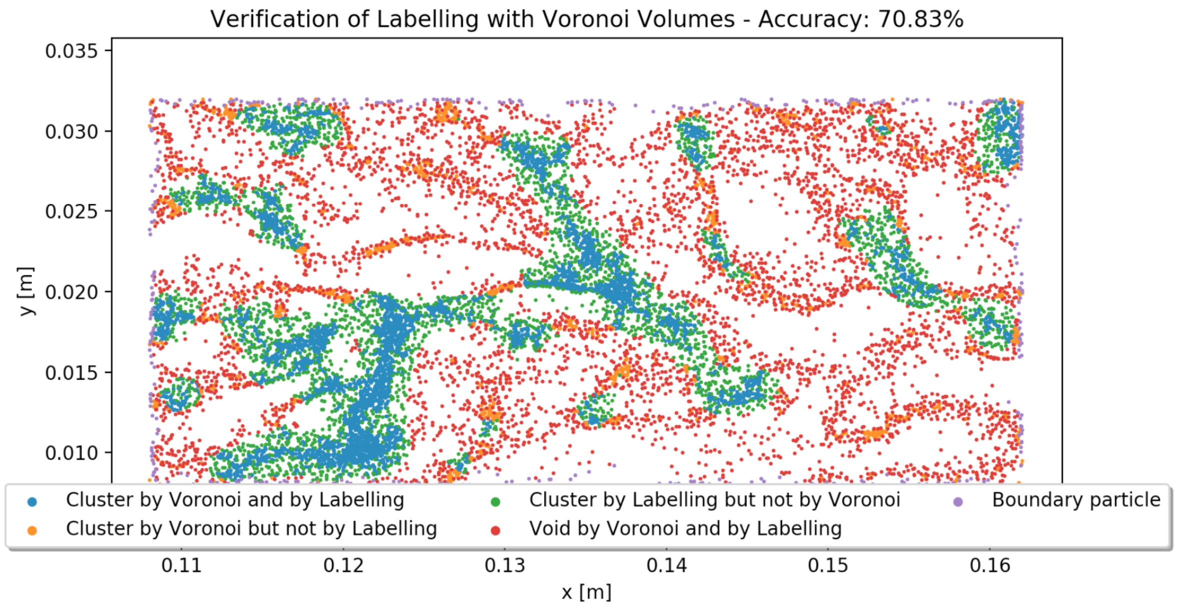

Figure 3.4: Evaluation of Cluster identification Based on Voronoï Tesselation of a Single Two-dimensional Snapshot in $Z = 0.02$ m and $t = 0.0002$ m - In blue, particles labeled as cluster particles by both methods. In red, particles labeled as void particles by both methods. In orange, particles labeled as cluster particles by the Voronoï-based approach but not by the DBSCAN-based approach. In green, particles labeled as cluster particles by the DBSCAN-based approach but not by the Voronoï-based approach. In purple, boundary particles.

Alternatively, if the thickness of the projection sheet is increased, such that 61322 particles are now included within the same two-dimensional domain and the value of $MinPts^*$ rises to 380, the accuracy of the DBSCAN-based approach does not vary significantly, it being 72.10%. Nevertheless, the number of small clusters which are not classified as such by the evaluated method decreases, as is visible from Figure 3.5.

What can be concluded from this evaluation is that the current application of DBSCAN coincides with an analogous application of Voronoï tessellations in order to determine particle clusters in a two-dimensional domain. However, it is visible that the requirement of $MinPts$ within a cluster limits the thickness of the clusters classified with the method developed in the current study. In any case, the capacity of the parameters configuring DBSCAN to adapt to the average number density of the domain is proven.

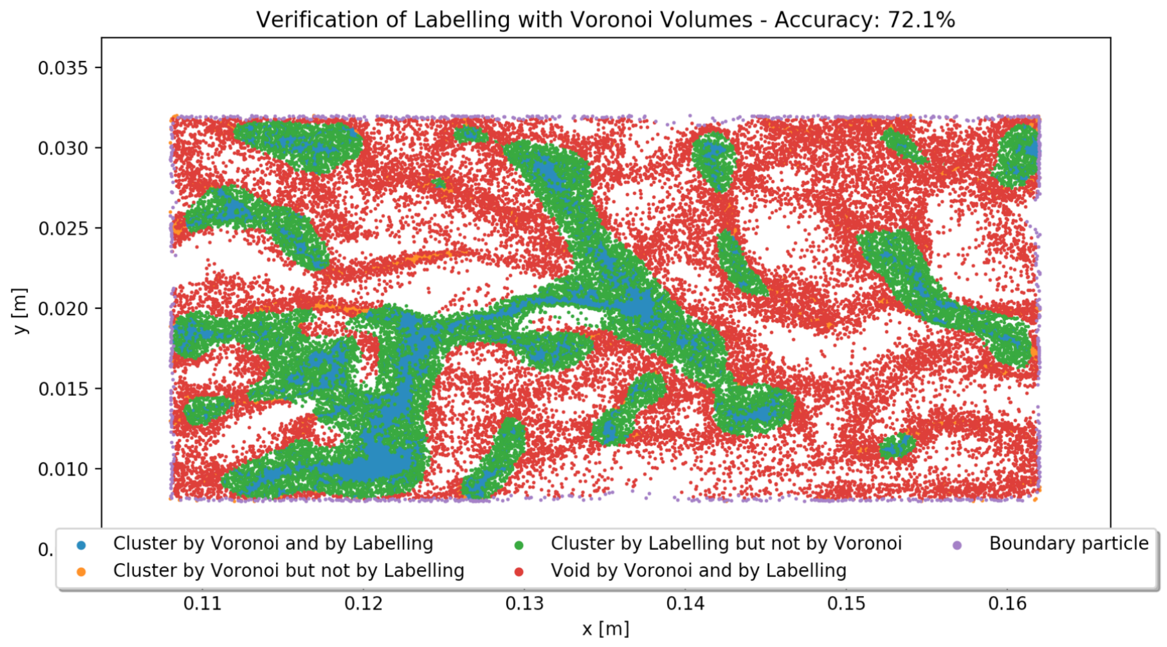

Figure 3.5: Evaluation of Cluster identification Based on Voronoï Tesselation of a Single Two-dimensional Snapshot in $Z = 0.02$ m and $\Delta Z = 0.0008$ m - In blue, particles labeled as cluster particles by both methods. In red, particles labeled as void particles by both methods. In orange, particles labeled as cluster particles by the Voronoï-based approach but not by the DBSCAN-based approach. In green, particles labeled as cluster particles by the DBSCAN-based approach but not by the Voronoï-based approach. In purple, boundary particles.

3.3 Evaluation of the Clustering Analysis of a 3D Database

In order to carry out an evaluation of how the implemented methods examine the particle positions in the dataset and classify such positions into three-dimensional regions of particle clusters or voids, it is necessary to once again validate the obtained results based on a method which is not associated to the choice of an arbitrary length scale, but rather solely requires a comparison with the case of uniformly distributed particles, as per a random Poisson process. Therefore, the evaluation of the clustering analysis of the complete three-dimensional dataset is also performed employing Voronoï tessellations, with a number of differences in its implementation with respect to the previous two-dimensional case.

On the one hand, this Voronoï analysis is not to be applied to the same dataset as is utilized within the context of this study, which projects particle positions into a finite number of planes of constant Z. On the contrary, this simplification of the three-dimensional domain into a set of two-dimensional domains is avoided, and thus the cell center of each Voronoï cell is to correspond to an actual unaltered particle position capable of appearing at any level of Z within the domain under study. This is necessary since the previously assigned three-dimensional cluster labels have to be verified, and thus cluster connectivities along different levels of Z have to occur spontaneously due to the nature of the dataset.

On the other hand, given the three-dimensional character of the dataset to which a Voronoï tessellation is to be applied, it follows that the property of each Voronoï cell to be examined in order to determine whether to classify it as a cluster or a void particle should cease to be its planar area, and should instead be its internal volume. More precisely, instead of comparing the probability density function of Voronoï cell areas in the preferentially concentrated case with that of the uniformly distributed set of particles, it follows instead to intersect the probability density functions of Voronoï cell volumes for both cases. Since the preferentially concentrated case is expected to present a higher frequency of both relatively small and relatively large Voronoï cells, both probability density functions will intersect at two volumes, the smaller of both representing the volume below which a Voronoï cell is classified as a cluster cell. Analogously to the validation of the clustering analysis of a single two-dimensional domain, it is convenient here to employ an analytical expression for the Voronoï cell volume probability density function, following a generalized gamma function fit, of the form expressed in Equation (3.2). This expression, in which according to Ferenc et al. (2007) [22], $a = 3.24174$, $b = 3.24269$, $c = 1.26861$, $\Gamma(\cdot)$ corresponds to the gamma function, and $V/\langle V\rangle$ represents the Voronoï cell volume normalized with respect to the mean Voronoï cell volume, when compared with the probability density function of the preferentially concentrated, unaltered set of three-dimensional particle positions, results in the intersections presented in Figure 3.6. It is worth noting that in order to efficiently calculate the volume associated to each Voronoï cell, its convex hull is generated, as is possible based on the existing Python library *SciPy*. Moreover, due to their spurious effect on the volume probability density function, Voronoï cells whose vertices appear outside of the domain of interest are discarded, these appearing close to the domain's boundaries.

$$P(V/\langle V\rangle) = \frac{c \cdot b^{\frac{a}{c}}}{\Gamma(a/c)} \cdot (V/\langle V\rangle)^{a-1} e^{-b(V/\langle V\rangle)^c} \tag{3.2}$$

Lastly, in this step of validation it is not sufficient to study whether each particle by itself is classifiable as belonging to a cluster or not, but rather one should also verify whether the three-dimensional particle structures which have been delimited within this study are sensible, in terms of their size and their shape. Therefore, it is also necessary to connect adjacent Voronoï cells whose volume is sufficiently low to be considered as part of a cluster, in order to define complex, three-dimensional regions of particle clusters, whose topology and volume can be compared with the particle structures which the three-dimensional extension of this study results in. Given that this procedure can prove to be computationally costly, and with the objective of carrying out a just evaluation of computational performance of the developed methods with their Voronoï-based alternative, it is necessary to examine the connectivity of adjacent cluster Voronoï cells in an efficient way. Instead of proceeding for each Voronoï vertex and taking into account the Voronoï cells adjacent to the vertex, the current implementation proceeds by analyzing each Voronoï ridge, and extracting the pair of Voronoï cell centers

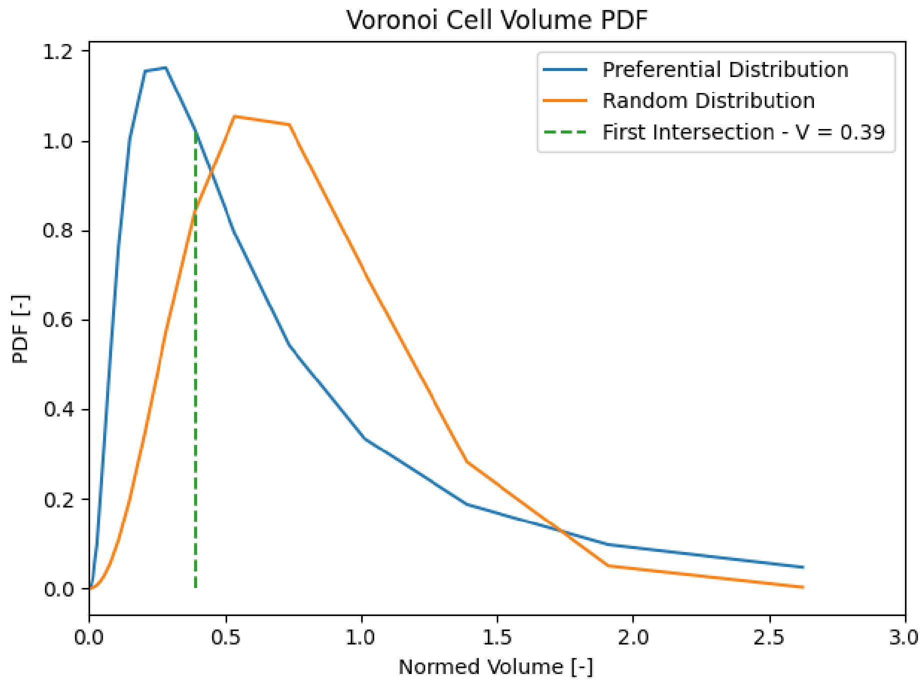

Figure 3.6: Approximated Intersection of Normalized Volume Probability Density Functions - In blue, density function corresponding to the preferentially concentrated case. In orange, density function following a random Poisson process.

between which it lies. This choice followed after perceiving that the latter method consumed significantly less time than the former, and thus was considered an apt alternative. As a result, adjacent Voronoï cell centers labeled as cluster particles are connected to form three-dimensional clusters with which the obtained results can be compared.

Based on these modifications in the application of Voronoï tessellations to the evaluation of the results obtained within this study, it is possible to examine to what extent the obtained three-dimensional particle structures are sensible in terms of their resulting topology and size.

3.3.1 Topological Coincidence of Clusters

When determining to what extent the classified three-dimensional clusters coincide in their shape with the analogous result obtained from the aforementioned application of Voronoï tessellations, it makes sense to take advantage of the condensation of the cluster's topology in a constellation of interior skeleton points. Given that an objective of the current study is to simplify the shape of a three-dimensional cluster into its boundary surface and a set of skeleton points, it is reasonable to expect that adequate quantitative and qualitative validations can be carried out parting from these interior points.

More precisely, given that these skeleton points represent positions interior to particle clusters, and that each of these points is assigned a cluster label based on three-dimensional connectivities, one can validate the resulting particle clusters by determining both to what extent skeleton points are included in regions which the presented Voronoï-based method classifies as pertaining to a particle cluster as well as whether skeleton points belonging to the same cluster label still share their cluster label according to this Voronoï validation. In simple terms, it is sought to examine whether the regions of the domain classified as occupied by particle clusters coincide according to both methods, and also to see if what one method labels as a single cluster is separated into several clusters by the other.

In order to do so, a simple pairing procedure has been followed. For each skeleton point, the closest Voronoï cell center is examined, and the Voronoï cluster label associated to such cell center is stored. Then, it follows to count, for each of the cluster labels assigned to skeleton points, what number of its assigned skeleton points is closest to a Voronoï cell center classified as a cluster cell, and what number of these points is related to each of the existing Voronoï cluster labels. The result is efficiently stored in an array with the same number of rows as labels assigned to skeleton points, and the same number of columns as existing Voronoï cluster labels. Then, the j-th element of the i-th row is to represent how many skeleton points with a cluster label i are closest to a Voronoï cell center of cluster label j.

After carrying out this counting procedure for the dataset of interest, what results is that out of the 1868 skeleton points taken into account, only 679 are closest to a Voronoï cell center classified as a cluster cell. In other words, 63.651% of the skeleton points in the dataset appear in regions which by this Voronoï-based alternative method do not belong to a particle cluster. It is nevertheless necessary to bear in mind that the skeleton points of the highest and lowest levels of Z in the domain are not contemplated in this value, since the Voronoï cells in the boundaries of the domain are not taken into account due to their spurious volume. If these skeleton points of extreme levels of Z were taken into account, the percentage of skeleton points appearing in void Voronoï cells would increase.

Moreover, if one dives deeper into the results of this validation, it is possible to discern an interesting distribution of the skeleton points associated to the largest cluster: out of the 1853 skeleton points included in this cluster, 1176 (63.465%) do not appear within any Voronoï cluster cell, and 586 (31.624%) occupy Voronoï cells associated with the largest cluster obtained with this alternative method, the remaining 4.911% of skeleton points with this label being scattered among other Voronoï clusters. On the other hand, the other 15 skeleton points of other cluster labels are not included within any Voronoï cluster cell.

Parting from this poor coincidence, one can visualize where do these skeleton points appear with respect to the Voronoï cluster cells with which they have been compared. In order to ease this visualization, it is necessary once again to simplify a three-dimensional domain into several two-dimensional domains. Thus, taking into account the finite number of Z levels at which skeleton points occur, the cell centers of Voronoï cells classified as cluster cells have been normally projected to the closest plane of constant Z, in order to examine to what extent do the cluster topologies defined by both methods differ.

The results of this visualization are presented in Figures 3.7, 3.8, and 3.9, where skeleton points are presented along projected positions of cluster cell centers, and these skeleton points are labeled according to the nature of their closest Voronoï cell centers. Moreover, the projected positions of Voronoï cell centers have been colored according to the different three-dimensional cluster labels which their connectivities allow for. From these results, it is possible to see that although most skeleton points are not closest to a Voronoï cluster cell, skeleton points within a single two-dimensional domain do portray a cluster topology which is similar to the one depicted by the cell centers.

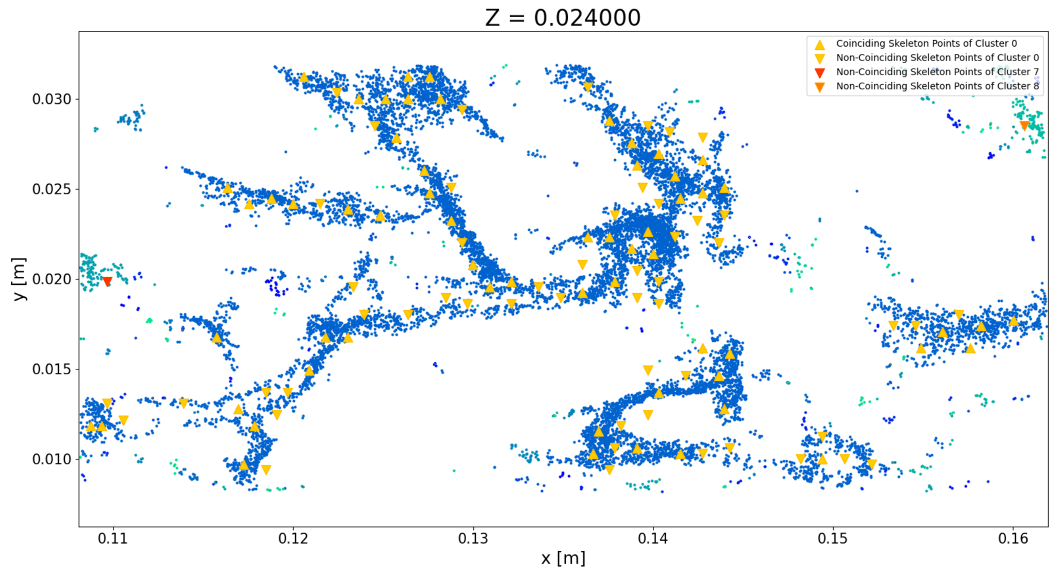

Figure 3.7: Topological Validation of Skeleton Points for $Z = 0.0240$ m - In green and blue points, Voronoï cluster cell centers, colored according to their three-dimensional cluster labels. In orange, red, and yellow triangles facing upwards, skeleton points whose closest Voronoï cell center is part of a particle cluster. In orange, red, and yellow triangles facing downwards, skeleton points whose closest Voronoï cell center is not part of a particle cluster.

Nevertheless, the fact that not all skeleton points exist in Voronoï cluster cells is explained by the way in which a cluster is defined in each method. While as when classifying a particle cluster made up of Voronoï cells smaller than a certain volume one can have any number of particles defining a cluster, the procedure implemented in this study in which DBSCAN is applied to a dataset of two-dimensional particle positions requires, by definition, a minimum number of particles to define a cluster. Although non-core data samples, or in this case particles

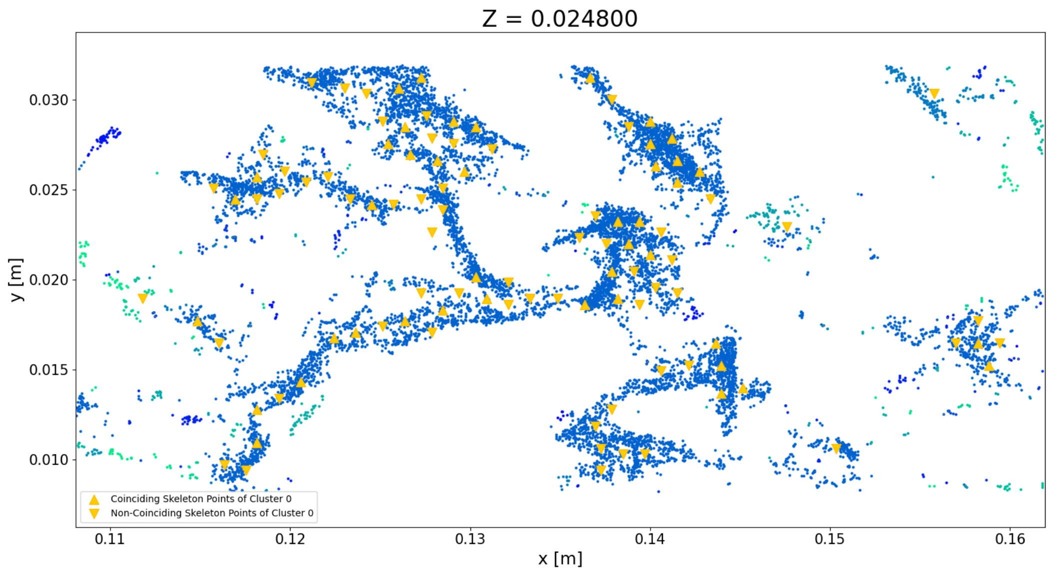

Figure 3.8: Topological Validation of Skeleton Points for $Z = 0.0248$ m - In green and blue points, Voronoï cluster cell centers, colored according to their three-dimensional cluster labels. In orange, red, and yellow triangles facing upwards, skeleton points whose closest Voronoï cell center is part of a particle cluster. In orange, red, and yellow triangles facing downwards, skeleton points whose closest Voronoï cell center is not part of a particle cluster.

without a sufficient number of neighboring particles, are purposely excluded from all clusters, the clusters which result from clustering with DBSCAN end up being much wider than the ones which Voronoï tessellations can result in. This results in skeleton points which appear outside of the concentrations of cell centers defining a particle cluster. In any case, it is possible to assert that the general cluster shape in every snapshot in Z is portrayed similarly by both methods, at most showing a significant disagreement for Voronoï-defined clusters made up of a very small number of cluster cells, which skeleton points are not capable of capturing.

On the other hand, due to this same requirement of a minimum number of points within a DBSCAN cluster, the alternative Voronoï-based method results in a significantly higher number of cluster labels. Many of these clusters appear within regions of the domain which the method developed in the current study associates to a single cluster. Nevertheless, both methods result in a single cluster label being applied to most of the particles in the dataset.

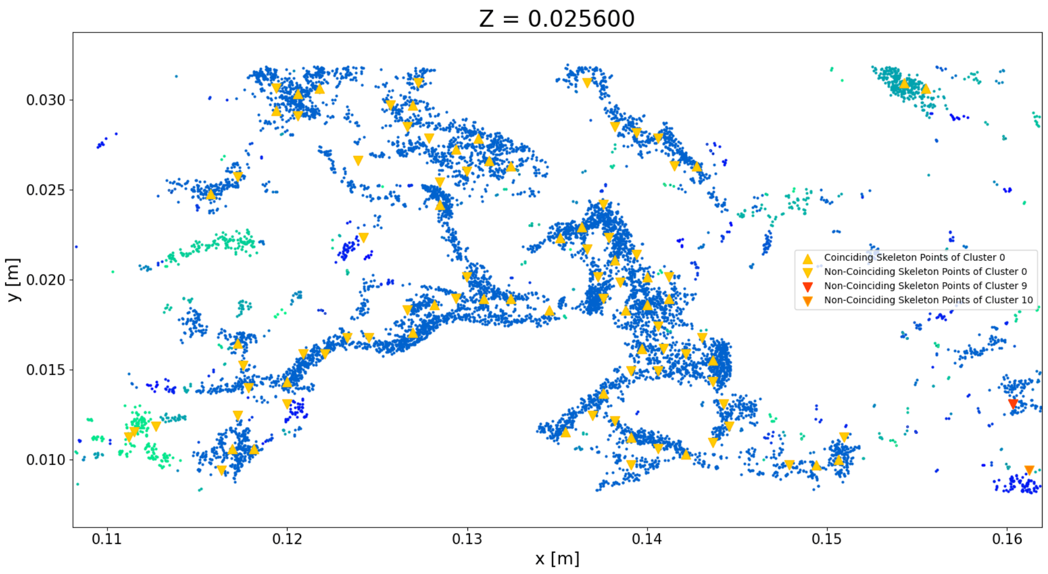

Figure 3.9: Topological Validation of Skeleton Points for $Z = 0.0256$ m - In green and blue points, Voronoï cluster cell centers, colored according to their three-dimensional cluster labels. In orange, red, and yellow triangles facing upwards, skeleton points whose closest Voronoï cell center is part of a particle cluster. In orange, red, and yellow triangles facing downwards, skeleton points whose closest Voronoï cell center is not part of a particle cluster.

3.3.2 Volumetric Coincidence of Clusters

It now follows to examine how the measures of the volumes of the resulting particle clusters vary from the DBSCAN-based method developed in this study to the Voronoï-based alternative with which its results are evaluated. On the one hand, this comparison allows one to observe what percentage of the domain is deemed to be occupied by particle clusters according to each method. On the other hand, with this evaluation it is possible to discern whether what one method classifies as a single cluster of relatively large volume is established to be multiple clusters of smaller volume by the other technique of analysis.

Before presenting the results of this comparison, it is necessary to describe how the volume of each cluster is estimated in the current implementation, since the inaccuracy of this method directly determines the certainty with which this evaluation can be carried out. In short, this estimation of a cluster's volume intends to work around the simplification of a three-dimensional domain into multiple two-dimensional domains by interpolating between the areas associated to the cluster in adjacent planes of constant Z. Since in order to obtain the constellation of points interior to a cluster expressing its topology, a uniform grid is generated within the cluster's area, where the number of cells containing particles of such cluster is known, it is possible to approximately discern the area associated to each DBSCAN cluster within a two-dimensional domain. Then, once these skeleton points are generated, connected, and labeled according to their three-dimensional connectivity, it is possible to state that what previously were independent two-dimensional DBSCAN clusters are now an approximation of the intersection areas of a three-dimensional cluster with planes of constant Z. Based on these intersection areas, one can estimate the cluster's volume by means of a lineal interpolation, knowing the separation between adjacent planes of constant Z. In Equation (3.3), one can see how this estimation of the volume $V(c)$ of a cluster c is carried out, where $A(c, z_i)$ represents the area which cluster c occupies in the two-dimensional domain of $Z = z_i$, there being N two-dimensional domains of constant Z.

$$V(c) = \sum_{i=2}^{N} \frac{A(c, z_i) + A(c, z_{i-1})}{2} \cdot (z_i - z_{i-1}) \tag{3.3}$$

As is expected, the error associated with this estimation of a cluster's volume is closely related with the separation between adjacent two-dimensional domains, since the closer these are, the less will the cluster's volume differ from a linear interpolation between adjacent areas. Moreover, in the case in which a cluster ceases to appear in one of the adjacent planes, the sum presented in Equation (3.3) is likewise carried out, only with a null area associated to this plane. The result of this estimation technique is the volume distribution presented in Figure 3.10. As is expected from what is obtained after determining the three-dimensional connectivities of the previously generated skeleton points, there is one cluster label which is applied to the great majority of cluster particles, whereas many other significantly smaller clusters also exist.

On the other hand, the estimation of the volume associated to each cluster resulting from a Voronoï-based clustering analysis is both simpler and more precise. Since the property determining whether a Voronoï cell corresponds to a cluster particle is its volume, this method of clustering classification innately requires a computation of the volume of each cell in the domain. This computation is carried out easily by determining the convex hull enveloping the Voronoï vertices defining the cell. Then, once adjacent cluster cells have been connected in order to define three-dimensional regions occupied by particle clusters, one can directly compute the volume associated to a cluster label by adding the volume of each of the Voronoï cells under such label. The error related to this measure of volume can be seen as caused by imprecisions in the calculation of the volume of a single cell, which assuming these Voronoï cells have a convex shape, can be considered negligible. In Figure 3.11, one can observe that there exists one particle cluster which connects a large number of cluster particles, whereas several other much smaller particle structures also appear.

When comparing both volume distributions exposed in Figures 3.10 and 3.11, one can extract a number of important conclusions. On the one hand, it stands out at a first glance that both distribution of volume along different particle clusters are very similar. Particularly, according to both methods of clustering analysis, there appears a cluster label associated to

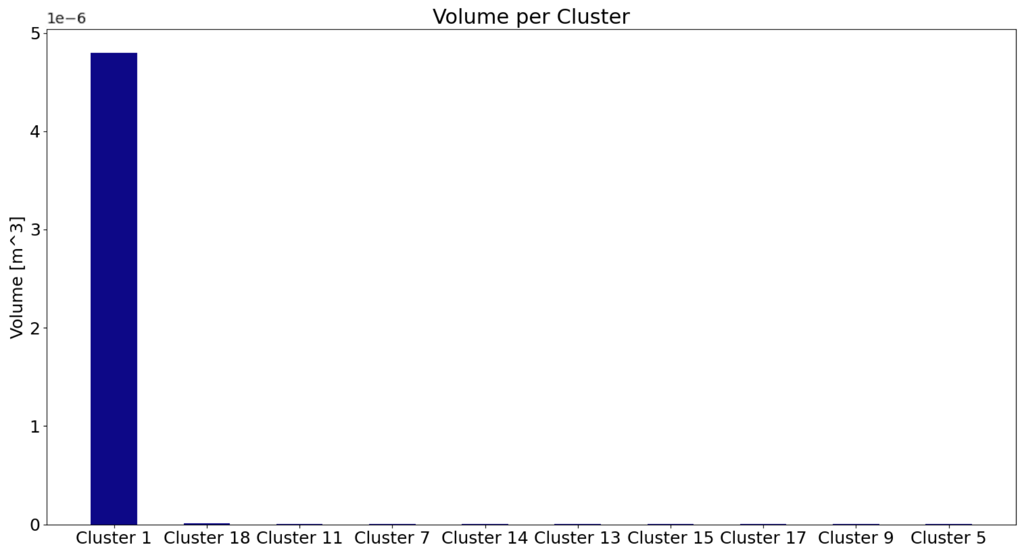

Figure 3.10: Estimated Volume of the 10 Largest Particle Clusters According to the Developed Methods of Clustering Analysis

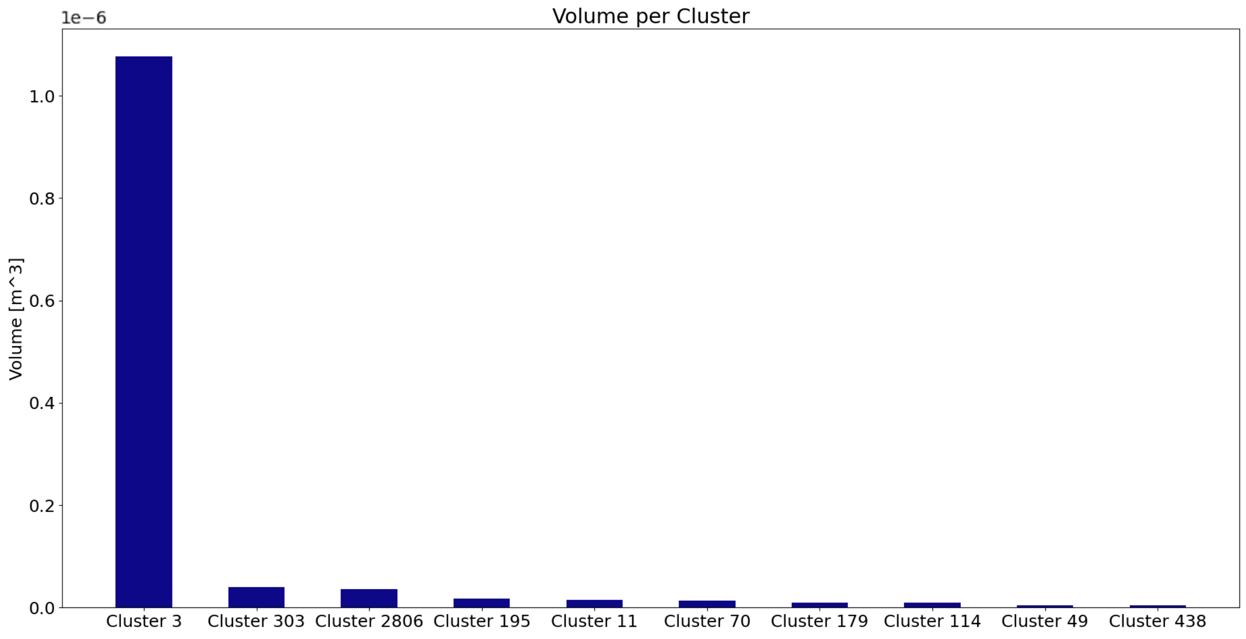

Figure 3.11: Estimated Volume of the 10 Largest Particle Clusters According to a Voronoï-based Method of Clustering Analysis

the great majority of cluster particles, such that the difference in volume between the largest cluster and the following clusters is significant. Nevertheless, purely from a qualitative point of view, the difference between the largest particle cluster and the remaining particle structures is more notorious according to the DBSCAN-based method of clustering analysis developed in the current study.

However, as soon as one delves into a quantitative examination of the results, it is possible to observe how the largest particle cluster following a Voronoï tessellation of unaltered particle positions is around four times smaller than its analogous cluster in the alternative method of cluster classification. This is most probably due to the fact that, as discussed in the previous validation, the clusters resulting from the DBSCAN alternative method are generally wider, since they have to include a minimum number of particles by definition. Thus, it is reasonable that the total volume of all cluster regions in the domain is larger according the current implementation than following a Voronoï-based alternative. However, it is also possible to argue that this difference in the maximum cluster volume is due to the fact that according to a cluster classification following Voronoï tessellations, not only do more clusters appear (for this particular dataset, 2816 different cluster labels are assigned), but also the difference between the largest and the second largest particle clusters is not as pronounced, thus resulting in a greater degree of distribution of volume among clusters. Lastly, a simpler reasoning of such divergence could be carried out by associating it to inaccuracies in the computation of cluster volumes according to Equation (3.3). In any case, it can be established that both methods coincide in defining a single cluster label to which the great majority of cluster particles is associated, accompanied by many other much smaller particle structures.

A further step in this validation can be carried out by comparing the probability density function of volume per cluster associated to the clustering technique developed within this study with that of different Voronoï-based alternatives in which the volume threshold defining which Voronoï cells are labeled as cluster cells is increased. In simple terms, volumes per cluster obtained with DBSCAN are contrasted with volumes per cluster based on Voronoï tessellations, where the maximum volume defining a cluster particle in these latter clustering techniques is modified. In particular, a base Voronoï analysis is carried out as well as two others in which the threshold volume is increased 25% and 50%.

At a first glance at Figure3.12, it can be seen that all probability density functions follow a similar trend. Nevertheless, it is worth noting that in order to obtain this result, all Voronoï clusters of volume lower than the minimum volume of all DBSCAN cluster have been excluded. Thus, the expected peak at much lower volumes for all Voronoï-based clustering alternatives does not appear and all functions converge. Furthermore, it can be seen that while the probability density functions of both alternatives do adopt a similar slope, the values of the Voronoï-based alternative are lower for higher cluster volumes. DBSCAN is once again not capable of defining particle clusters as small as those classified by means of Voronoï tessellations, due to the core requirement of a minimum number of particles defining the cluster. For high cluster volumes, it is not visible that as the threshold with which cluster particles are classified in all Voronoï-based alternatives is increased, the probability density functions do approximate that of the DBSCAN counterpart, since the probability density function at these volumes is noisy.

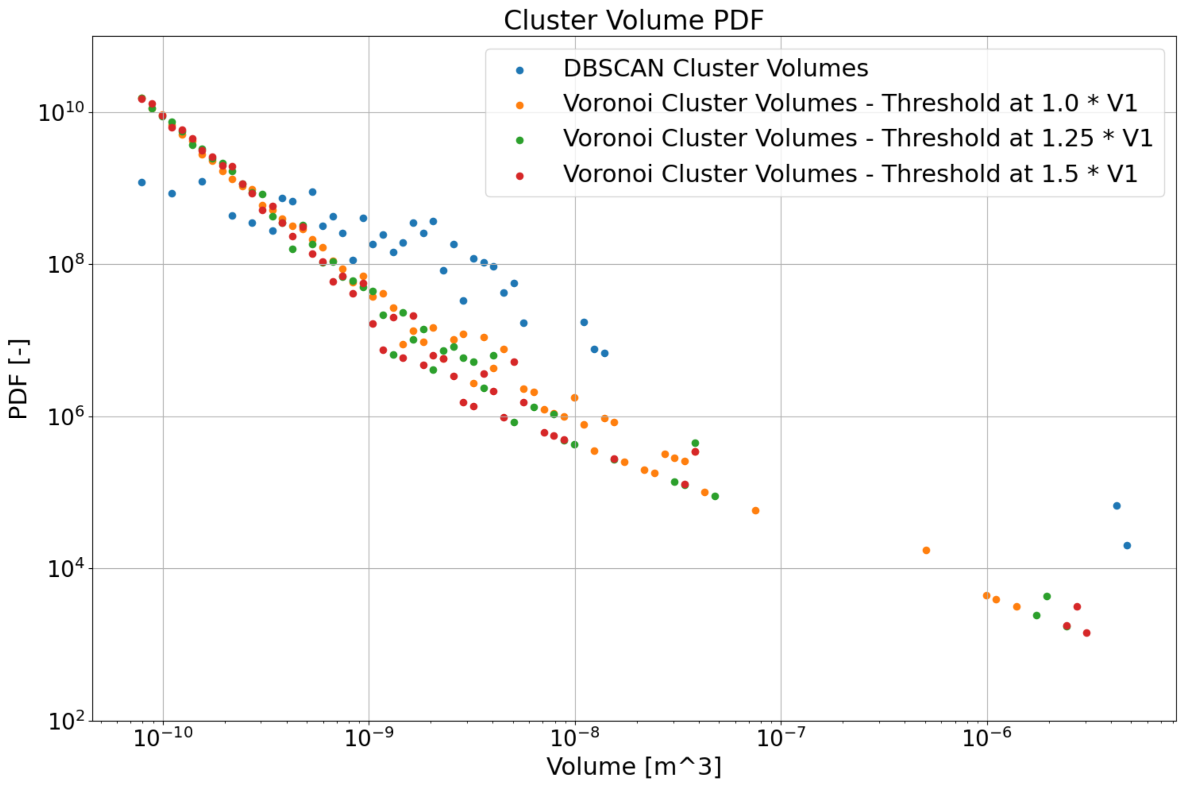

Figure 3.12: Probability Density Function of Volumes per Cluster - In blue, volumes per cluster of the DBSCAN-based clustering method. In orange, volumes per cluster of the Voronoï-based alternative. In green, volumes per cluster of the Voronoï-based alternative with a 25% shift of the volume threshold. In red, volumes per cluster of the Voronoï-based alternative with a 50% shift of the volume threshold.

3.4 Comparison of Computational Performance

When comparing the time required to carry out the clustering analysis of a dataset of particle positions in the DBSCAN-based method developed within this study with that of a Voronoï-based alternative method, it is necessary to take into account on the one hand that the starting conditions with which both methods are to operate must be the same, in order to ensure a just comparison. In this way, a dataset concerning the same spatial domain of the duct flow is employed as input for both methods, involving the same number of particles within the flow. Nevertheless, it is worth noting that the dataset based on which the DBSCAN-based method operates already has its particle positions projected into a discrete number of planes of constant Z, whereas the analogous Voronoï-based alternative works with the unprocessed set of particle positions, capable of appearing at any level of Z. Moreover, as should be clear after reading the previous sections in this chapter, employing Voronoï tessellations to perform a clustering analysis of this same dataset does not condense each cluster's topology into a boundary surface and a set of interior skeleton points, a step which is indeed carried out by the techniques developed in the current study. Therefore, although this comparison of execution times does ensure an equality in the complexity of the inputs to both methods, the DBSCAN-based alternative produces the same results as its Voronoï counterpart, and goes even further. It is worth noting that all of the presented execution times correspond to a 2015 MacBook Pro with a Dual-Core Intel Core i5 processor running at 2.7 GHz using 8 GB of RAM, running macOS Catalina.

To properly examine the time consumption of the method of clustering analysis based on DBSCAN developed in the current study, it is necessary to take into account that this method as a whole, for the analysis of a particular dataset at an individual instant of time, is divided into six substeps, each associated with a particular performance. In brief, these can be summarized as follows:

1. **clustering:** This step deals with the DBSCAN clustering analysis of each two-dimensional domain in the dataset, generating a set of unconnected, two-dimensional clusters based on the whole of projected particle positions in the dataset.
2. **boundary:** This step obtains a set of closed cluster boundary curves for each of the two-dimensional clusters defined in the previous step.
3. **discretize:** Here, each two-dimensional domain is discretized by means of a uniform grid, and the cells containing cluster particles are detected. This discretization is utilized to calculate cluster areas as well as in the next step.
4. **skeletonize:** Based on the previous discretization, a set of skeleton points interior to each two-dimensional cluster is defined.
5. **connectivities:** In this step, skeleton points from different two-dimensional domains are connected, in order to assign a three-dimensional cluster label to each of these points based on their connectivities.

6. **relabel:** Lastly, the cluster label of each skeleton point is assigned to its associated cluster particles, in order to classify each cluster particle according to its three-dimensional cluster membership. Also, three-dimensional cluster volumes are estimated.

If the execution of each of these steps on a single dataset is timed, one obtains the distribution of time consumption presented in 3.13. At a first glance, one can extract that the whole clustering analysis of the employed dataset requires close to 4300 seconds, and that the most expensive step within this procedure is that of determining two-dimensional cluster boundaries. Second to this subroutine is the three-dimensional connection of skeleton points, amounting however to much less required time.

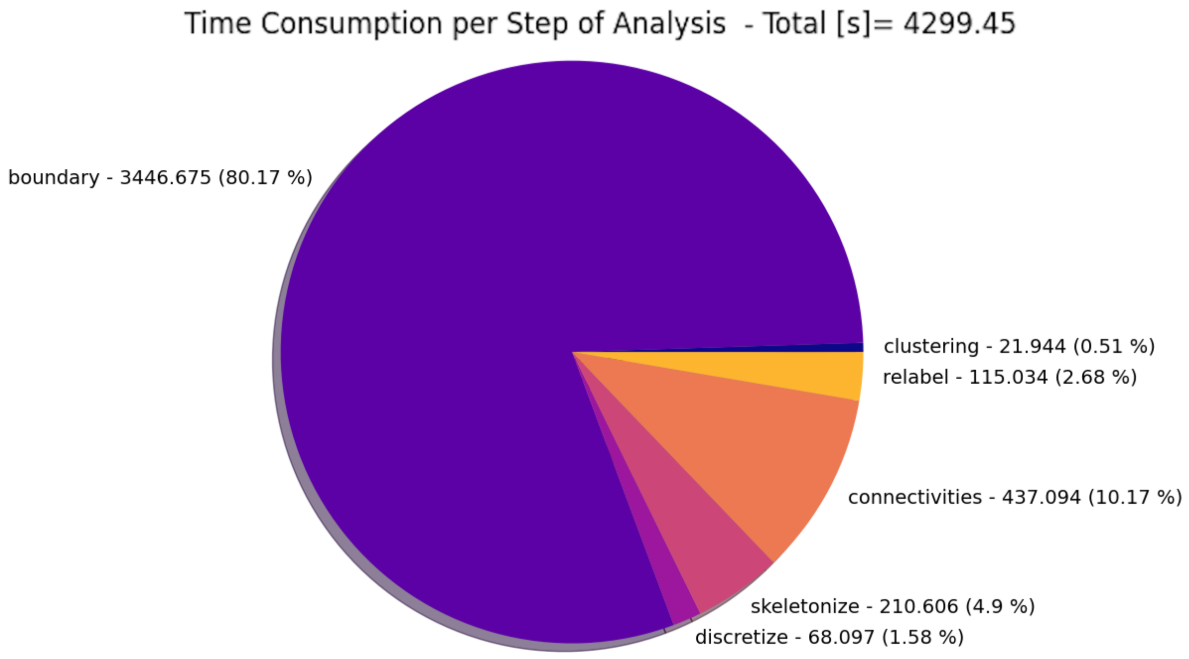

Figure 3.13: Time Consumption of the DBSCAN-based Clustering Method - All times are in seconds

However, in order to determine values of $MinPts$ and ϵ which are adequate for the average number density within a single two-dimensional domain, it is necessary to initially carry out an additional set of calculations, the results of which can be applied to any dataset of the same nature. Therefore, after obtaining an adapted value for these parameters, one can proceed to analyze the clustering structure of any dataset in which the average number density is the same, without having to recalculate them. For the current dataset, obtaining an unbiased value of $MinPts$ resulted in 1569.54 seconds of execution, while as obtaining an appropriate ϵ required 106.73 seconds.

Step	Execution Time [s]	Percentage [%]
Voronoï Tessellation and Classification	478	1.1
Three-Dimensional Cluster Particle Connection	42525	98.9
Total	43003	100

Table 3.1: Time Consumption of the Voronoï-based Clustering Alternative

On the other hand, the Voronoï-based alternative with which the method developed within this study is evaluated can be divided into two simple steps. The first of these steps deals with the Voronoï tessellation of the three-dimensional domain and the classification into cluster and void particles based on the Voronoï volume of each particle position in the dataset. The next step carries out the connection of all Voronoï cells whose center has been classified as a cluster particle, in order to assign three-dimensional cluster labels to each particle in the dataset. Based on what is presented in Table 3.1, one can observe that this alternative method requires about 43000 seconds for its complete execution, almost the totality of this time being occupied with connecting cluster particles in order to generate and apply a set of three-dimensional cluster labels.

When comparing both alternative methods, for a single execution on the same dataset, one can see that the total execution time of the Voronoï-based approach is around ten times larger than that of the method implemented in this study. Moreover, while as in the procedure employing DBSCAN the most expensive step deals with obtaining a set of cluster boundary points, when analyzing particle clusters with Voronoï tessellations, one incurs the greatest cost at the time of determining a valid set of three-dimensional cluster labels. Furthermore, it is necessary to insist on the fact that the DBSCAN-based alternative outputs more useful information regarding cluster topologies than its Voronoï counterpart, for a much lower computational cost. As a result, when analyzing the topology of particle clusters in turbulent flow, the method implemented in this study significantly outperforms an alternative procedure based on Voronoï tessellations, even providing more information regarding this analysis.

Chapter 4

Conclusions

In the current study, two powerful density-based clustering algorithms have been applied in order to identify particle clusters within a dataset of positions concerning particles diluted in turbulent flow. With regards to the employed dataset, special care has been taken in order to simplify a three-dimensional domain into a set of two-dimensional domains whose clustering structure is analyzed independently. Then, each two-dimensional domain has been treated separately by applying DBSCAN in the task of separating cluster particles from void particles, and even dividing cluster particles into different clusters based on their proximity within the same two-dimensional domain. As an objective of this study is the simplification of a three-dimensional particle cluster by means of its boundary particles and a set of interior constellation of points, the following steps of this study deal with the determination of cluster boundaries for each DBSCAN cluster in each two-dimensional domain, and based on these, the generation of a brute skeleton condensing the topology of its associated cluster. Furthermore, two-dimensional domains cease to be treated independently once connectivities between brute skeleton points at different values of Z are examined, thus creating three-dimensional particle cluster labels to be applied to these skeleton points, and later on to the cluster particles from which they have been derived.

Another objective of the current study has been to develop a method with which particle clusters can be tracked over time. Based on the aforementioned simplification of a three-dimensional particle cluster, from the results of this method of particle cluster identification for two adjacent time instants, the evolution of characteristics of these particle structures with time can be examined. This is done by pairing brute skeleton points from different time steps based on an expected deviation from pure translation with the duct flow and on an examination of similarities between neighboring cluster boundary particles.

Lastly, when evaluating the implemented method with an alternative technique of particle cluster identification based on Voronoï tessellations as is presented in Monchaux et al. (2010)

[7], a couple more conclusions can be extracted. On the one hand, when validating the DB-SCAN clustering analysis of a single two-dimensional domain, one can see that due to the nature of this clustering technique, the clusters which this method identifies are significantly thicker than those obtained by its Voronoï-based counterpart. On the other hand, the developed methods of identifying three-dimensional particle clusters are evaluated by comparing the topology of particle clusters resulting from both methods as well as by examining how the distribution of cluster volumes changes from one method to the other. When comparing the resulting cluster topologies, it is reasonable to conclude that while as most brute skeleton points do not appear in regions which the alternative method assigns to a particle cluster, the shapes that these points trace are qualitatively very similar to those which the method presented in Monchaux et al. (2010) [7] results in. Furthermore, the distribution of particle cluster volumes is very similar, although it is still noticeable that the the clusters defined by the methods of the current study are generally somewhat thicker, thus resulting in generally greater cluster volumes. Lastly, a comparison of execution times reveals that the Voronoï-based alternative is outperformed by the methods developed within this study, its execution on the same dataset requiring close to ten times as much time.

In conclusion, the innovative application of relatively new algorithms belonging to the growing area of unsupervised learning has resulted in a method of identifying and characterizing three-dimensional particle clusters which requires much less time than the existing alternative developed by Monchaux et al. (2010) [7]. Based on this improvement, it is worth examining whether other much more recent developments in the area of machine learning, such as for instance the exciting invention of Generative Adversarial Networks by [23], could be of any future use not only in the analysis of turbulent dispersed multiphase flow, but also in other quantitative analyses within the area of fluid mechanics.

Bibliography

[1] Jonathan D. Kulick John R. Fessler and John K. Eaton. “Preferential concentration of heavy particles in a turbulent channel flow”. In: *Physics of Fluids* 6 (1994).

[2] A. Aliseda et al. “Effect of preferential concentration on the settling velocity of heavy particles in homogeneous isotropic turbulence”. In: *Journal of Fluid Mechanics* 468 (2002), p. 77.

[3] A. Banko et al. “PREDICTION OF PREFERENTIAL CONCENTRATION STATISTICS FROM EULERIAN TWO-POINT CORRELATIONS”. In: *10th International Symposium on Turbulence and Shear Flow Phenomena (TSFP10)* (2017).

[4] R. Monchaux, M. Bourgoin, and A. Cartellier. “Analyzing preferential concentration and clustering of inertial particles in turbulence”. In: *International Journal of Multiphase Flow* 40 (2012), pp. 1–18.

[5] S. Balachandar and J. Eaton. “Turbulent Dispersed Multiphase Flow”. In: *Annu. Rev. Fluid Mech.* 42 (2010), pp. 111–133.

[6] L. Villafañe-Roca et al. “A robust method for quantification of preferential concentration from finite number of particles”. In: *Center for Turbulence Research* Annual Research Briefs 2016 (2016), pp. 123–135.

[7] R. Monchaux, M. Bourgoin, and A. Cartellier. “Preferential concentration of heavy particles: A Voronoï analysis”. In: *Phys. Fluids* 22 (2010).

[8] A. Jain and R. Dubes. *Algorithms for Clustering Data.* Upper Saddle River: Prentice-Hall, Inc., 1988.

[9] D. Xu and Y. Tian. “A Comprehensive Survey of Clustering Algorithms”. In: *Ann. Data. Sci. (2015)* 2 (2015), pp. 165–193.

[10] J. MacQueen. “Some Methods for Classification and Analysis of Multivariate Observations”. In: *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability* 1 (1967), pp. 281–297.

[11] L. Kaufman and P. Rousseeuw. *Finding Groups in Data: an Introduction to Cluster Analysis.* Hoboken: Wiley, 1990.

[12] T. Zhang, R. Ramakrishnan, and M. Livny. “BIRCH: An Efficient Data Clustering Method for Very Large Databases”. In: *1996 ACM SIGMOD International Conference on Management of Data* 4-6 June 1996 (1996), pp. 103–114.

[13] C. Rasmussen. “The Infinite Gaussian Mixture Model”. In: *Advances in Neural Information Processing Systems* 12 (1999), pp. 554–560.

[14] M. Ester et al. “A Density- Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise”. In: *Proc. 2nd Int. Conf. on Knowledge Discovery and Data Mining, Portland, OR, AAAI Press* (1996), pp. 226–231.

[15] M. Ankerst et al. “OPTICS: Ordering Points To Identify the Clustering Structure”. In: *Proc. ACM SIGMOD’99 Int. Conf. on Management of Data* (1999).

[16] E. Schubert et al. “DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN”. In: *ACM Trans. Database Syst. 42* 3 (2017).

[17] M. Esmaily et al. “A benchmark for particle-laden turbulent duct flow: a joint computational and experimental study”. In: *International Journal of Multiphase Flow* (2020).

[18] M. Sanchez. “FRACTAL DIMENSION FOR CLUSTERING AND UNSUPERVISED AND SUPERVISED FEATURE SELECTION”. In: *Manufacturing Engineering Centre - School of Engineering - Cardiff University* (2011).

[19] A. Sharf et al. “On-the-fly Curve-skeleton Computation for 3D Shapes”. In: *EUROGRAPHICS 2007* 26 (2007).

[20] D. Montana and L. Davis. “Training Feedforward Neural Networks Using Genetic Algorithms”. In: *BBN Systems and Technologies Corp.* (1989).

[21] A.L. Hinde and R.E. Miles. “Monte Carlo estimates of the distributions of the random polygons of the Voronoï tessellation with respect to a Poisson process”. In: *Journal of Statistical Computation and Simulation* 10 (1980), pp. 205–223.

[22] J.-S. Ferenc and Z. Néda. “On the size distribution of Poisson Voronoi cells”. In: *Physica A* 385 (2007), pp. 515–526.

[23] I. Goodfellow et al. *Generative Adversarial Nets.* Montréal, QC H3C 3J7: Departement d’informatique et de recherche opérationnelle, Université de Montréal, 2014.

Chapter 5

Appendices

A Organization of Presented Code

The following appendices contain the code implemented in the current study. The functions and connections of each script can be summarized as follows:

- **OpenSliceVTK.py:** This script simply loads a .VTK file containing three-dimensional particle positions, projects them into a specified number of planes of constant Z, and outputs the resulting projected positions into a .CSV file.
- **clusteringAnalysis.py:** For a single .CSV file containing projected particle positions obtained with **OpenSliceVTK.py**, this script invokes the necessary classes of other scripts in order to perform all of the steps involved in the identification of three-dimensional clusters parting from the simplified dataset. This script invokes **clustering3D.py**, **boundaryFinder.py** , **eulerianApproach.py** , **clusterConnect.py** , and **relabeller.py**.
- **clustering3D.py:** For a set of parallel two-dimensional domains containing projected particle positions, this script carries out the two-dimensional DBSCAN clustering analysis of each two-dimensional plane. In order to do so, it is also in charge of obtaining a set of parameters for DBSCAN which are adapted to the average number density of the domain. It can also command the visualization of the results. This script invokes **OPTICS.py**, **DBSCAN.py**, and **clusterPlot.py**.
- **OPTICS.py:** This script employs the clustering algorithm OPTICS in order to analyze the structure of a two-dimensional domain of projected positions, as well as to determine a set of parameters for DBSCAN which are adapted to the average number density of the domain. To visualize the results, this script invokes **clusterPlot.py**.

- **DBSCAN.py:** This script employs the clustering algorithm DBSCAN in order to analyze the structure of a two-dimensional domain of projected positions. To visualize the results, this script invokes **clusterPlot.py**.
- **clusterPlot.py:** Given a set of projected particle positions to which different cluster labels have been assigned, this script generates a .GIF file representing the clustering configuration of each two-dimensional domain.
- **boundaryFinder.py:** This script is in charge of determining closed curves defining the boundaries of two-dimensional clusters of projected particle positions, as the ones obtained with **clustering3D.py**. To visualize the results, this script invokes **skeletonPlot.py**.
- **skeletonPlot.py:** Given a set of closed curves defining two-dimensional cluster boundaries and skeleton points interior to each cluster, this script generates a .GIF file representing these elements within each two-dimensional domain.
- **eulerianApproach.py:** This script discretizes each two-dimensional domain by means of a regular grid, in order to calculate the area of each DBSCAN cluster generated by **clustering3D.py**, and also, for each two-dimensional cluster, determines a set of interior points defining its brute skeleton. To visualize the results, this script invokes **clusterPlot.py** and **skeletonPlot.py**.
- **clusterConnect.py:** Based on the cluster boundaries defined by **boundaryFinder.py** and the skeleton points generated in **eulerianApproach.py** for each two-dimensional cluster, this script examines the three-dimensional connectivity of skeleton points in order to come up with three-dimensional particle cluster labels. To visualize the results, this script invokes **skeletonPlot.py**.
- **relabeller.py:** This script is in charge of assigning the three-dimensional cluster labels obtained in **clusterConnect.py** to the corresponding projected particle positions, as well as of estimating the volume associated with each three-dimensional cluster. To visualize the results, this script invokes **clusterPlot.py**.
- **temporalTracking.py:** Here, for a pair of datasets corresponding to two adjacent instants of time, each of them analyzed via **clusteringAnalysis.py**, skeleton points are connected according to topological similarities, in order to connect three-dimensional clusters over time. Based on this, the temporal evolution of a cluster for several instants of time can be analyzed. To visualize the results, this script invokes **temporalPlot.py**.
- **temporalPlot.py:** Given a pair of datasets corresponding to two adjacent instants of time, each of them analyzed via **clusteringAnalysis.py**, this script generates a .GIF file which represents old and new cluster boundaries in the same two-dimensional domain, as well as the connections between skeleton points as per **temporalTracking.py**.
- **voronoiCluster.py:** This script is in charge of validating the results of **clusteringAnalysis.py** with an alternative clustering method based on the Voronoï tessellation of the same set of particle positions without having been projected into a number of planes of constant Z. To visualize the results, this script invokes **voronoiPlot.py**.

- **voronoiPlot.py:** Here, a .GIF file is generated that displays, for each two-dimensional domain in the dataset, a projection of Voronoï cluster cell centers as per **voronoiCluster.py** as well as skeleton points within the same plane of constant Z.

B OpenSliceVTK.py

```
import numpy as np
import vtk
from vtk.util import numpy_support as VN
import matplotlib.pyplot as plt


# Development of data analysis tools for the topological and temporal
↪  analysis of clusters of particles in turbulent flow
# Script Description: This script is designed to open a specified VTK file
↪  representing the positions of particles in a
# simulation of particle-laden flow and extract their position for a thin
↪  plane bisecting the simulation's domain. Class
# multipleZs is in charge of invoking class VTKopen, in order to define a
↪  series of computational frames out a dataset
# describing turbulent particle-laden flow within a square duct.
# Álvaro Tomás Gil - UIUC 2020

class VTKopen:
    """Class in charge of reading the actual .vtk file, cropping and
     ↪  projecting its positions based on the specified ranges,
    and saving the results to a .csv file.
    For the inputs:
    filename: String with file name
    xrange & yrange: Lists of two elements defining ranges in X and Y,
↪   defining the span of each computational plane,
    normalized w.r.t the duct dimensions in each direction
    zs: Thickness of the sheet associated to every computational frame,
↪   defining the amount of particles to project onto
    the plane. This thickness is normalized w.r.t. the duct dimension in Z
    dZ: Spacing between adjacent computational planes, normalized w.r.t. the
↪   duct dimension in Z
    normalize: Boolean. If True, normalized coordinates will appear"""
    def __init__(self, filename, xrange, yrange, zs, dZ, normalize=False):
        reader = vtk.vtkPolyDataReader()
        reader.SetFileName(filename + '.vtk')
        reader.Update()
        self.polydata = reader.GetOutput()

        if np.isscalar(zs):
            zs = np.array([zs])

        self.r = np.array([0, 0, 0])
        self.fullr = [np.array([0, 0, 0])] * len(zs)
        self.x = xrange
        self.y = yrange
        self.zs = zs
        self.dZ = dZ
        self.n = self.polydata.GetNumberOfPoints()
        self.norm = normalize

        #Initial sweep of the data file, to determine dimensions in each
        ↪  direction for plotting and normalization.
        for i in range(0, self.n, 1000):
            self.r = np.vstack((self.r, list(self.polydata.GetPoint(i))))
        self.r = self.r[1:]

        self.dims = np.zeros(3)
        for i in range(3):
            self.dims[i] = round(max(self.r[:, i]), 2)
            self.r[:, i] = self.r[:, i] / max(self.r[:, i])

    def read(self, filename='test'):
        """This method is in charge of reading every line of the .vtk file,
        ↪  extracting positions within the specified
        range, normalizing if necessary, and saving the extracted data as a
↪   .csv file"""

        total = 0
        for i in range(0, self.n, 1):
            vraw = list(self.polydata.GetPoint(i))
            v = [vraw[j] / self.dims[j] for j in range(3)]

            if v[0] > self.x[0] and v[0] < self.x[1] and v[1] > self.y[0] and
            ↪  v[1] < self.y[1]:
                which = np.vstack((v[2] < self.zs + self.dZ, v[2] > self.zs -
                ↪  self.dZ))
                which = np.ndarray.flatten(np.argwhere(np.all(which,
                ↪  axis=0)))
                # "which" describes within which sheet (or Z level) such a
                ↪  point should be classified
                if len(which) > 0:
                    total += 1
                    if self.norm:
                        v[2] = self.zs[which[0]]
                        temp = np.vstack((self.fullr[which[0]], v))
                    else:
                        vraw[2] = self.zs[which[0]] * self.dims[2]
                        temp = np.vstack((self.fullr[which[0]], vraw))

                    self.fullr[which[0]] = temp
            print(float(100 * i / self.n), '% percent read')

        for i in range(len(self.zs)):
            self.fullr[i] = self.fullr[i][1:, :]

        print(str(total) + ' particles in data set')

        self.data = np.array([0, 0, 0])
        for i in self.fullr:
            self.data = np.vstack((self.data, i))
        self.data = self.data[1:, :]

        np.savetxt(filename + '.csv', self.data, delimiter=",")

    def plots(self, onlyHist=True):
        """This method is in charge of developing a histogram of particle
        ↪  concentration along Y, to study the influence of the
        duct walls, and of plotting the particle positions for a single Z
↪   value"""
        plt.figure()
        plt.hist(self.r[:, 1]*self.dims[1], bins=100)
        plt.xlabel('y [m]')
        plt.ylabel('Number of Particles [-]')
        plt.title('Concentration of Particles along Y')

        if not onlyHist:
            plt.figure()
            plt.scatter(self.fullr[:, 0], self.fullr[:, 1], s=0.5)
            plt.title(str(len(self.fullr)) + ' particles in data set')
            plt.xlabel('x [m]')
            plt.ylabel('y [m]')
            plt.axis('equal')
        plt.show()


class multipleZs:
    """Class in charge of invoking VTKopen, by introducing the desired values
    ↪  of Z for each computational plane"""
    def __init__(self, filename='prt_TG_ductVe8_780000', xrange=[0.4, 0.6],
    ↪  yrange=[0.2, 0.8], thick=0.01, spacing=0.02, wallMargin=0.2,
    ↪  normalize=False):
        self.dZ = thick
        self.sp = spacing
        self.margin = wallMargin
        self.x = xrange
        self.y = yrange
        self.filename = filename
        self.norm = normalize

        self.zs = np.unique(
            np.concatenate((np.arange(0.5, 1 - self.margin, self.sp),
             ↪  np.arange(self.margin, 0.5, self.sp))))
        self.vtk = VTKopen(filename, xrange, yrange, self.zs, self.dZ,
         ↪  normalize)

    def writeFile(self):
        thick = str(round(self.dZ, 3)).replace('.', ',')
        spacing = str(round(self.sp, 3)).replace('.', ',')
        filename = self.filename + '_Large_dZ_' + thick + '_Spacing_' +
         ↪  spacing

        self.vtk.read(filename)

root = ['./Data/t_','/prt_TG_ductVe8_77']
root = ['./Data/v78/t_','/prt_TG_ductVe8_78']
times = [11, 12, 13, 14, 15]
for t in times:
    t0 = str(t * 100)
    if len(t0) < 4:
        ts = '0' * (4 - len(t0)) + t0
    else:
        ts = t0
    filename0 = root[0] + t0 + root[1] + ts
    vt = multipleZs(filename0)
    vt.writeFile()
```

C clusteringAnalysis.py

```
import numpy as np
import matplotlib.pyplot as plt
```

```
from clustering3D import clustering3D
from clusterConnect import skeletonConnect
from relabeller import relabeller
from eulerianApproach import eulerianAnalysis, eulerianSkeleton,
↪  optimumCellSize
from boundaryFinder import boundaryFinder
import time

# Development of data analysis tools for the topological and temporal
↪  analysis of clusters of particles in turbulent flow
# Script Description: This script carries out all the steps of analysis for a
↪  single data frame of the simulation of
# particle-laden turbulent flow. Parting from a .csv datafile of particle
↪  positions sorted into a set of dicrete planes
# of constant Z, this script applies DBSCAN clustering to each of these
↪  snapshots in Z to separate cluster from void
# particles, obtains a set of closed curves defining each clusters'
↪  boundaries, generates a constellation of interior
# points defining each cluster's brute skeleton, connects these interior
↪  points along different levels of Z, and based
# on these connections assigns a set of cluster labels which associate each
↪  cluster particle to a three-dimensional cluster.
# Álvaro Tomás Gil - UIUC 2020


class clusteringAnalysis:
    """This class carries out all the steps of analysis for a single data
     ↪  frame of the simulation of
    particle-laden turbulent flow. Parting from a .csv datafile of particle
↪   positions sorted into a set of dicrete planes
    of constant Z, this script applies DBSCAN clustering to each of these
↪   snapshots in Z to separate cluster from void
    particles, obtains a set of closed curves defining each clusters'
↪   boundaries, generates a constellation of interior
    points defining each cluster's brute skeleton, connects these interior
↪   points along different levels of Z, and based
    on these connections assigns a set of cluster labels which associate each
↪   cluster particle to a three-dimensional cluster.
    For the inputs:
    filename: This is the name of the .csv file from which to begin this
↪   analysis. Note that with different time frames,
    projection sheet thicknesses, or spacings between these sheets, parts of
↪   this filename will vary
    t: This number references the current time frame to analyze. Based on
↪   this time frame, a different folder will be treated"""
```

```
    def __init__(self,
     ↪  filename='prt_TG_ductVe8_770000_Large_dZ_0,01_Spacing_0,02', t=0,
     ↪  root='./Data/t_'):
        self.t = t
        self.folder = root + str(t) + '/'
        self.filename = self.folder + filename + '.csv'
        self.times = [0, 0, 0, 0, 0, 0]

    def run(self, skip=0, time=False):
        """This is the main method of the class.
        skip: Int which sets at which point of the analysis to stop loading
↪   previous data and start generating new data.
        time: Boolean setting whether to plot the time consumption of
↪   different steps"""
        self.clustering(loadNskip=skip > 0)
        self.boundaries(loadNskip=skip > 1)
        self.eulerize(loadNskip=skip > 2)
        self.skeletonize(loadNskip=skip > 3)
        self.connect(loadNskip=skip > 4)
        self.relabel()

        if time:
            self.timePiePlot()

    def clustering(self, loadNskip=False, take=20, knownMinPts=212,
     ↪  knownEps=0.0015):
        """This method is in charge of applying DBSCAN clustering analysis to
         ↪  separate the particles contained in every
        separate plane of constant Z into cluster and void particles. Special
↪   care goes into the definition of the
        parameters with which to carry out DBSCAN, both requiring different
↪   methods of class clustering3D to be invoked.
        For the inputs:
        loadNskip: Boolean which commands the method to skip execution and
↪   directly return previously saved results
        take: Number of planes of constant Z to analyze from the original
↪   .csv dataset
        knownMinPts: Since the determination of an un-biased value of this
↪   parameter is costly, one can directly input
        an adequate value instead.
        knownEps: Since the determination of an un-biased value of this
↪   parameter is costly, one can directly input
        an adequate value instead.
        For the outputs:
        MinPts: MinPts parameter with which DBSCAN is carried out
        eps: epsilon parameter with which DBSCAN is carried out
        data: 2D array of 3D particle positions, sorted into parallel planes
↪   of constant Z
        DBSCANlabels: list of equal lenght as data, assigning a cluster label
↪   applicable within each 2D domain. A label
        of -1 corresponds to a void particle."""
        start = time.time()
        if loadNskip:
            self.MinPts = np.load(self.folder + 'MinPts.npy')
            self.eps = np.load(self.folder + 'eps.npy')
            self.data = np.load(self.folder + 'allPoints.npy')
            self.DBSCANlabels = np.load(self.folder + 'DBSCANlabels.npy')
        else:
            # 3D DBSCAN clustering of Particle Coordinates
            self.c3d = clustering3D(self.filename, take=take,
             ↪  folder=self.folder)
            # Extraction of Unbiased Hyperparameters
            self.c3d.MinPts(known=knownMinPts)
            self.c3d.meanReach(known=knownEps)
            self.c3d.sweep()

            self.MinPts = self.c3d.MinPts
            self.eps = self.c3d.eps
            self.data = self.c3d.data
            self.DBSCANlabels = self.c3d.labels

            np.save(self.folder + 'MinPts.npy', self.c3d.MinPts)
            np.save(self.folder + 'eps.npy', self.c3d.eps)
            np.save(self.folder + 'allPoints.npy', self.c3d.data)
            np.save(self.folder + 'DBSCANlabels.npy', self.c3d.labels)
        self.times[0] = time.time() - start

    def boundaries(self, loadNskip=False):
        """This second method has the task of extracting the cluster
         ↪  particles in each cluster which serve as boundaries
        with respect to the rest of the domain.
        For the input:
        loadNskip: Boolean which commands the method to skip execution and
↪   directly return previously saved results
        For the outputs:
        bound3d: 2D array of 3D positions corresponding to the curves
↪   defining each cluster's boundaries for each of the
        parallel planes of constant Z in the domain
        """
        start = time.time()
        if loadNskip:
            self.bound3d = np.load(self.folder + 'bound3d.npy')
        else:
            self.bF = boundaryFinder(self.data, self.DBSCANlabels, self.eps,
             ↪  folder=self.folder)
            self.bF.sweep()
            self.bound3d = self.bF.bound3d

            np.save(self.folder + 'bound3d.npy', self.bF.bound3d)
        self.times[1] = time.time() - start

    def eulerize(self, loadNskip=False):
        """This method discretizes each plane of constant Z by means of a
         ↪  uniform grid. This discretization is used to
        compute the area associated to each two-dimensional DBSCAN cluster as
↪   well as to generate the skeleton inner to
        the cluster.
        For the input:
        loadNskip: Boolean which commands the method to skip execution and
↪   directly return previously saved results
        For the outputs:
        DBSCANareas: list of dicts, describing the area associated to each
↪   DBSCAN cluster label in a single Z level. This
        list has as many elements as different parallel planes of constant Z
↪   exists, and each of these elements is a dict
        whose keys are the DBSCAN labels of the Z level and whose values are
↪   the areas of those clusters.
        cellCenters: 2D array of 3D positions corresponding to the cell
↪   centers of all discretized planes of constant Z
        cellLabels: 1D array of equal length as cellCenters, where each
↪   element corresponds to the DBSCAN label assigned to
        each grid cell, based on the DBSCAN label of the particles it
↪   contains."""
        start = time.time()
        if loadNskip:
            self.DBSCANareas = np.load(self.folder + 'DBSCANareas.npy',
             ↪  allow_pickle=True)
            self.cellCenters = np.load(self.folder + 'cellCenters.npy')
            self.cellLabels = np.load(self.folder + 'cellLabels.npy')
        else:
            self.eu = eulerianAnalysis(self.data, self.DBSCANlabels,
             ↪  self.eps, boundMethod=None, folder=self.folder)
            self.eu.run()

            self.DBSCANareas = self.eu.areas
            self.cellCenters = self.eu.rc
            self.cellLabels = self.eu.lc
            np.save(self.folder + 'DBSCANareas.npy', self.eu.areas)
            np.save(self.folder + 'cellCenters.npy', self.eu.rc)
            np.save(self.folder + 'cellLabels.npy', self.eu.lc)
        self.times[2] = time.time() - start

    def skeletonize(self, loadNskip=False):
        """Taking into account the previous discretization of each
        ↪  two-dimensional domain, this method determines a set
        of points interior to the two-dimensional domain which condense its
↪   topology.
        For the input:
        loadNskip: Boolean which commands the method to skip execution and
↪   directly return previously saved results
        For the outputs:
        skel: 2D array of 3D positions of these skeleton points
        """
        start = time.time()
        if loadNskip:
            self.skel = np.load(self.folder + 'skeletonize.npy')
        else:
            self.sk = eulerianSkeleton(self.cellCenters, self.cellLabels,
            ↪  self.bound3d, self.eps, folder=self.folder)
            self.sk.run()

            self.skel = self.sk.skel

            np.save(self.folder + 'skeletonize.npy', self.sk.skel)
        self.times[3] = time.time() - start

    def connect(self, loadNskip=False):
        """Once these interior points have been generated, this method
        ↪  examines the connectivity between them, both for
        the same level of Z as well as along different levels. Based on these
↪   connectivities, the method groups the
        skeleton points into different cluster labels, which take into
↪   account the 3D connectivities within different
        clusters at different planes of constant Z.
        For the input:
        loadNskip: Boolean which commands the method to skip execution and
↪   directly return previously saved results
        For the outputs:
        SKlabels: List of equal length as skel, assigning a cluster label to
↪   each skeleton point.
        """
        start = time.time()
        if loadNskip:
            self.SKlabels = np.load(self.folder + 'SKlabels.npy')
        else:
            self.skC = skeletonConnect(self.skel, self.bound3d, self.eps,
             ↪  folder=self.folder)
            self.skC.run()

            self.SKlabels = self.skC.labels

            np.save(self.folder + 'SKlabels.npy', self.skC.labels)
        self.times[4] = time.time() - start

    def relabel(self):
        """Lastly, this method applies the cluster labels obtained by
         ↪  examining connectivities of different skeleton points
        to the cluster particles grouped by means of DBSCAN. This method
↪   results in a new list of labels to apply to each
        cluster particles, taking into account cluster connections in 3D."""
        start = time.time()
        self.reL = relabeller(self.data, self.DBSCANlabels, self.skel,
         ↪  self.SKlabels, self.DBSCANareas, self.folder)
        self.reL.run()
        self.times[5] = time.time() - start
        np.save(self.folder + 'DBSCANVolumes.npy', self.reL.volumes)

    def timePiePlot(self, pctM=0.00):
        """This method simply generates a plot of the time consumption
         ↪  associated to each step of the analysis."""
        names = ['clustering', 'boundary', 'discretize', 'skeletonize',
         ↪  'connectivities', 'relabel']
        dict = {}
        for i,j in zip(names, self.times):
            dict[i] = j

        total = sum(dict.values())
        title = 'Time Consumption per Step of Analysis  - Total [s]= ' +
         ↪  str(round(total, 3))
        labels = []
        values = []
        for v in dict.keys():
            if dict[v] / total > pctM:
                labels.append(v + ' - ' + str(round(dict[v], 3)) + ' (' +
                 ↪  str(round(dict[v] * 100 / total, 2)) + ' %)')
            else:
                labels.append(v)
            values.append(dict[v] / total)
        labdis = 1.07
        cmap = plt.get_cmap("plasma")
        c = np.arange(len(dict.keys())) / len(dict.keys())
        colors = cmap(c)

        fig = plt.figure()
        fig.set_size_inches(12, 7)
        plt.title(title, fontsize=22)
        plt.pie(dict.values(), labels=labels, shadow=True, startangle=0,
        ↪   labeldistance=labdis, colors=colors, textprops={'fontsize': 14})
        plt.axis('equal')  # Equal aspect ratio ensures that pie is drawn as
        ↪   a circle.


root = ['prt_TG_ductVe8_78', '00_Large_dZ_0,01_Spacing_0,02']
# root = ['prt_TG_ductVe8_77', '00_Large_dZ_0,01_Spacing_0,02']
folder = './Data/v78/t_'
times = [8]
for t in times:
    t0 = str(t)
    if len(t0) < 2:
        ts = '0' * (2 - len(t0)) + t0
    else:
        ts = t0
    filename = root[0] + ts + root[1]
    c = clusteringAnalysis(filename=filename, t=t * 100, root=folder)
    c.run()
```

D clustering3D.py

```
import numpy as np
import matplotlib.pyplot as plt
from OPTICS import exploreOPTICS, reachabilityConvergence, compareWithRandom
from DBSCAN import exploreDBSCAN
from clusterPlot import clusterPlot
import time
import csv

# Development of data analysis tools for the topological and temporal
↪   analysis of clusters of particles in turbulent flow
# Script Description: This script is to determine the adequate pair or
↪   parameters epsilon and MinPts based on which a DBSCAN
# clustering analysis of several two-dimensional snapshots in Z is carried
↪   out. This script is also capable of evaluating
```

```
# the time consumed during its execution.
# Álvaro Tomás Gil - UIUC 2020


class clustering3D:
    """This is the main class of the script, which loads the dataset from its
    ↪  corresponding file, extracts the specified
    number of snapshots in Z to analyze, determines the optimum pair of
↪   MinPts, epsilon values based on a single snapshot,
    and sweeps throughout all of these planes in Z to carry out a DBSCAN
↪   clustering analysis. It also has the capability
    of plotting the results and evaluating its time consumption. For the
↪   inputs:
    filename: String which specifies file from which to specify the number of
↪   snapshots in Z to analyze
    take: Integer which specifies the number of snapshots in Z to analyze,
↪   starting from the midplane of the square duct.
    folder: folder in which to save resulting plots"""
    def __init__(self, filename, take=1, folder=''):
        self.folder = folder
        with open(filename, 'r') as f:
            reader = csv.reader(f, delimiter=',')
            self.uncut = np.array(list(reader)).astype(float)

        self.allZ = np.unique(self.uncut[:, 2])
        self.middle = int(np.floor(len(self.allZ) / 2))

        if take & 0x1 == 1:
            add = [a for a in range(int((take - 1) / 2) + 1)]
            subs = [-a for a in range(int((take - 1) / 2) + 1)]
        else:
            add = [a for a in range(int(take / 2) + 1)]
            subs = [-a for a in range(int(take / 2))]

        ind = np.unique(add + subs) + self.middle
        self.zs = np.ndarray.flatten(np.array([self.allZ[x] for x in ind]))
        self.data = np.array([0, 0, 0])

        for z in self.zs:
            self.data = np.vstack((self.data, self.uncut[self.uncut[:, 2] ==
            ↪  z, :]))
        self.data = self.data[1:, :]

        #Trial is the snapshot in Z which is chosen to obtain the optimum
        ↪  pair of MinPts and eps for the DBSCAN analysis
        self.trial = self.uncut[self.uncut[:, 2] == self.allZ[self.middle],
        ↪   :2]

    def MinPts(self, start=10, end=500, itera=30, xi0=0.05, known=0):
        """This method is in charge of determining the optimum value of
        ↪   MinPts to later be employed in the DBSCAN analysis
         of the dataset. In order to do so, it iterates throughout several
↪   values of MinPts. For the input:
         start: Integer representing the first value of MinPts to examine
         end: Integer representing the last value of MinPts to examine
         itera: Integer representing the number of values of MinPts to
↪   examine
         xi0: Float describing the value of xi to be employed in the OPTICS
↪   analysis of every examination
         known: Integer describing the known optimum MinPts in order to
↪   override the execution of this step."""
        startT = time.time()
        print('Determination of MinPts*')
        if known > 0:
            self.MinPts = known
            self.MinPtsT = 1569.539
        else:
            MinPts0 = np.linspace(start, end, itera)
            rC = reachabilityConvergence(filename=None, MinPts=MinPts0,
            ↪   xi=xi0, data=self.trial)
            rC.convergeConcaves(plot=False)
            self.MinPts = rC.convMinPts
            self.MinPtsT = time.time() - startT
        print('Execution Time: ' + str(round(self.MinPtsT, 3)))

    def meanReach(self, known=0):
        """This method is in charge of determining the optimum value of
        ↪   epsilon to later be employed in the DBSCAN analysis
        of the dataset. In order to do so, it calculates the mean
↪   reachability distance of a set of randomly distributed
        particles. For the input:
        known: Float describing the known optimum epsilon in order to
↪   override the execution of this step."""
        if known > 0:
            self.eps = known
            self.epsT = 106.725
        else:
            start = time.time()
            print('Determination of Mean Reachability of Random
            ↪   Distribution')
            cR = compareWithRandom(filename=None, MinPts=self.MinPts,
             ↪  xi=0.05, data=self.trial)
            cR.run()
            self.eps = cR.meanReach * 0.95
            self.epsT = time.time() - start
        print('Execution Time: ' + str(round(self.epsT, 3)))

    def sweep(self):
        """Once both optimum parameters are known, this method performs the
         ↪  DBSCAN analysis of each of the snapshots in
        Z in question."""
        start = time.time()
        print('Sweep Along Z of Clustering Routine')

        self.labels = np.array([0])
        for z in self.zs:
            print('Z = ' + str(round(z, 3)))
            take = self.data[self.data[:, 2] == z, :]
            db = exploreDBSCAN(self.eps, self.MinPts, data=take)
            db.sweep()
            db.onlyCoreElements()
            self.labels = np.hstack((self.labels, db.labelsN[-1]))

        self.labels = self.labels[1:]
        self.sweepT = time.time() - start
        print('Execution Time: ' + str(round(self.sweepT, 3)))

    def visualize(self):
        """This method plots the clustering analysis' results."""
        cP = clusterPlot(self.data, self.labels, self.folder)
        cP.plotAll('3D DBSCAN Analysis - Z in ' + str(self.zs))

    def timeEvaluation(self):
        """This method analyzes the execution time associated to each of the
         ↪  steps in the analysis."""
        self.totalT = self.MinPtsT + self.epsT + self.sweepT

        labels = 'MinPts', 'meanReach', 'Sweep along Z'
        sizes = [self.MinPtsT, self.epsT, self.sweepT]
        explode = (0, 0, 0.1)

        fig1, ax1 = plt.subplots()
        wedges, texts, autotexts = ax1.pie(sizes, explode=explode,
         ↪  labels=labels, autopct='%1.1f%%', shadow=True,
                                           startangle=180, rotatelabels=True)
        ax1.axis('equal')
        ax1.legend(wedges, labels, loc="best")
        plt.title('Execution Time Evaluation - ' + str(len(self.zs)) + '
         ↪  Slices Taken - Total Time [m] = ' + str(
            round(self.totalT / 60, 3)))
```

E OPTICS.py

```
import numpy as np
from sklearn.cluster import OPTICS
import matplotlib.pyplot as plt
from clusterPlot import clusterPlot
import csv

# Development of data analysis tools for the topological and temporal
 ↪  analysis of clusters of particles in turbulent flow
# Script Description: This script is designed to analyze a given dataset of
 ↪  particles with OPTICS in order to obtain the set
# of parameters eps and MinPts to be employed in a clustering scheme with
 ↪  DBSCAN. The script includes a simple class exploreOptics
# which performs the OPTICS analysis of the dataset for several values of
 ↪  MinPts and xi/eps, another class reachabilityConvergence
# which examines the evolution of the data's reachability plot with MinPts
 ↪  and obtains a value for MinPts at which the shape of the
# plot stabilizes, and another class compareWithRandom, which compares the
 ↪  reachability plot corresponding to such optimum MinPts
# with the mean reachability value resulting from applying an OPTICS analysis
 ↪  to a randomly distributed set of particles of the same
# length as the original dataset.
# Álvaro Tomás Gil - UIUC 2020

class exploreOPTICS:
    """This class is in charge of performing a preliminary OPTICS analysis of
     ↪  a 2D computational domain displaying particle positions. It
    has the capability of applying the clustering algorithm to the database
 ↪  for multiple different values of xi and MinPts, in order to
    analyze the effect of these parameters on the reachability plot of the
 ↪  data. For the input:
    xi: Float or list of floats, defining the xi or eps values based on which
 ↪  to perform xi or eps-clustering with OPTICS
    MinPts: Integer or list of integers, defining the list of MinPts based on
 ↪  which to perform xi or eps-clustering with OPTICS
    plots: Number of plots defining, for different values of xi or eps, the
 ↪  clustering results
```

```
    filename: In case data is None, file name from which to load data
    data: Data array on which to perform OPTICS analysis
    method: String defining which method to apply for OPTICS clustering
    distance: String defining which method to apply to compute distances
↪   within the algorithm"""
    def __init__(self, xi, MinPts, plots, filename=None, data=None,
    ↪   method='xi', distance='euclidean'):

        self.clust = []
        self.reach = []
        self.space = []
        self.dR = []
        self.normedR = []
        self.labels = []

        if np.isscalar(xi):
            self.xi = np.array([xi])
        else:
            self.xi = xi

        if np.isscalar(MinPts):
            self.MinPts = np.array([MinPts])
        else:
            self.MinPts = MinPts

        if data is not None:
            self.data = data
        else:
            with open(filename, 'r') as f:
                reader = csv.reader(f, delimiter=',')
                self.data = np.array(list(reader)).astype(float)

        self.N = len(self.data)
        self.method = method
        self.distance = distance
        print('OPTICS: ' + str(self.N) + ' particles in data set.')
        self.n = np.array([0, 0, 0, 0])
        self.pl = [int(v) for v in np.linspace(0, len(self.xi) - 1, plots)]

        plt.figure()
        plt.scatter(self.data[:, 0], self.data[:, 1], s=0.8)
        plt.xlabel('x [m]')
        plt.ylabel('y [m]')
        plt.axis('equal')
        if filename == None:
            plt.title('Particle Distribution' + ' - ' + str(self.N) + '
             ↪  particles')
        else:
            plt.title(filename + ' - ' + str(self.N) + ' particles')

    def sweep(self):
        """For every combination of the specified values of xi/eps and
         ↪  MinPts, this method carries out the clustering analysis of the
        dataset based on those parameters. The results of such clustering are
↪   stored in clust, and the reachability distances of each
        clustering analysis are also calculated via reachability."""
        for j, u in enumerate(self.MinPts):
            for i, v in enumerate(self.xi):
                print('        ' + str(i + 1) + 'th xi: ' + str(round(v, 4))
                 ↪  + '; ' + str(j + 1) + 'th MinPts: ' + str(
                    np.floor(u)))
                if self.method == 'xi':
                    c = OPTICS(min_samples=int(u), xi=v, algorithm='auto',
                     ↪  metric=self.distance)
                else:
                    c = OPTICS(min_samples=int(u), eps=v, algorithm='auto',
                     ↪  metric=self.distance)
                c.fit(self.data)

                # While as clust is a list of OPTICS objects, labels contains
                 ↪  the labels of such objects
                self.clust.append(c)
                self.labels.append(c.labels_)

                # Every row of n contains: [xi, MinPts, Number of Clusters,
                 ↪  Percentage of Noise]
                nC = [v, u, len(np.unique(np.array(c.labels_))) - 1,
                 ↪  len(np.argwhere(c.labels_ == -1)) * 100 / self.N]
                self.n = np.vstack((self.n, nC))

                if i in self.pl:
                    # Plot the clustering disposition of the radii selected
                     ↪  in array pl
                    title = 'OPTICS with MinPts = ' + str(np.floor(u)) + '
                     ↪  and xi = ' + str(round(v, 4)) + '. ' + str(
                        nC[-2]) + ' clusters detected.'
                    cP = clusterPlot(self.data, c.labels_)
                    cP.plotAll(title)
            # cP.sizeHistogram(title)

            self.reachability(u)
        self.n = self.n[1:, :]

    def reachability(self, pts):
        """This function computes and stores the reachability distances for
        ↪  clustering analyses associated with the
        MinPts specified via pts. Note that the reachability plot will not
↪   depend on xi nor eps."""
        ofPts = np.ndarray.flatten(np.argwhere(self.n[1:, 1] == pts))
        c = self.clust[ofPts[-1]]
        reachability = c.reachability_[c.ordering_]
        space = np.arange(len(self.data))
        minR = np.nanmin(reachability[np.isfinite(reachability)])

        self.reach.append(reachability)
        self.space.append(space)
        self.dR.append(np.diff(reachability) / np.diff(space))
        self.normedR.append(reachability / minR)

    def plot(self, forXi=True, forMinPts=True):
        """This method is in charge of plotting the evolution of the number
        ↪  of detected clusters and the percentage of
        particles labelled as noise with a varying xi/eps and/or a varying
↪   MinPts"""
        if forXi:
            for i, v in enumerate(self.MinPts):
                fig, ax1 = plt.subplots()
                color = 'tab:red'
                ax1.set_xlabel('xi [-]')
                ax1.set_ylabel('Clusters Detected [-]', color=color)
                ax1.plot(self.xi, self.n[self.n[:, 1] == v, 2], color=color)
                ax1.tick_params(axis='y', labelcolor=color)
                ax1.set_title('OPTICS with MinPts = ' + str(v) + ' and
                ↪  Euclidean Distance')

                ax2 = ax1.twinx()
                color = 'tab:blue'
                ax2.set_ylabel('Percentage of Noise [%]', color=color)
                ax2.plot(self.xi, self.n[self.n[:, 1] == v, 3], color=color)
                ax2.tick_params(axis='y', labelcolor=color)
                fig.tight_layout()

        if forMinPts:
            for i, v in enumerate(self.xi):
                fig, ax1 = plt.subplots()
                color = 'tab:red'
                ax1.set_xlabel('MinPts [-]')
                ax1.set_ylabel('Clusters Detected [-]', color=color)
                ax1.plot(self.MinPts, self.n[self.n[:, 0] == v, 2],
                 ↪  color=color)
                ax1.tick_params(axis='y', labelcolor=color)
                ax1.set_title('OPTICS with xi = ' + str(v) + ' and Euclidean
                 ↪  Distance')

                ax2 = ax1.twinx()
                color = 'tab:blue'
                ax2.set_ylabel('Percentage of Noise [%]', color=color)
                ax2.plot(self.MinPts, self.n[self.n[:, 0] == v, 3],
                 ↪  color=color)
                ax2.tick_params(axis='y', labelcolor=color)
                fig.tight_layout()

        # ax = fig.add_subplot(111, projection='3d')
        # ax.scatter(self.n[:,0], self.n[:,1], self.n[:,2], marker = 'o', c =
        ↪  self.n[:,2], cmap = 'inferno')
        # ax.set_title('Clusters Detected - OPTICS with Euclidean Distance')
        # ax.set_xlabel('xi [-]')
        # ax.set_ylabel('MinPts [-]')

    def PlotReachabilityWithLabels(self):
        """This method plots the reachability distances associated to all
        ↪  possible values of MinPts, also including
         within the plot how the cluster label assignment varies with
↪   xi/eps."""
        for pts in self.MinPts:

            plt.figure(figsize=(10, 7))
            plt.ylabel('Reachability')
            plt.title('Reachability Plot - MinPts: ' + str(pts))

            ofPts = np.ndarray.flatten(np.argwhere(self.n[:, 1] == pts))
            space = self.space[ofPts[0]]
            reachability = self.reach[ofPts[0]]
            minR = np.nanmin(reachability[np.isfinite(reachability)])
            maxR = np.nanmax(reachability[np.isfinite(reachability)])
            lines = np.linspace(minR + 0.1 * (maxR - minR), maxR - 0.1 *
             ↪  (maxR - minR), len(ofPts))
            plt.plot(space, reachability)

            for i in range(len(lines)):

                if len(lines) == 1:
                    dLine = 0.05 * (maxR - minR)
                else:
                    dLine = 0.1 * (lines[1] - lines[0])
                c = self.clust[ofPts[i]]
                y = lines[i] * np.ones((len(space), 1))
                labels = c.labels_[c.ordering_]
                unique_labels = np.unique(np.array(labels))
                colors = [plt.cm.viridis(each) for each in np.linspace(0, 1,
                ↪   len(unique_labels))]

                xi = self.n[ofPts[i], 0]
                font = {'family': 'serif', 'color': 'black', 'weight':
                ↪   'normal', 'size': 12}
                plt.text(space[0], lines[i] + dLine, 'xi = ' + str(round(xi,
                ↪   5)), fontdict=font)

                for k, col in zip(unique_labels, colors):
                    size = 1.6
                    if k == -1:
                        # Black used for noise.
                        col = [128 / 256, 128 / 256, 128 / 256]
                        size = 1

                    class_member_mask = (labels == k)
                    plt.plot(space[class_member_mask], y[class_member_mask],
                    ↪   'o', markerfacecolor=tuple(col),
                             markersize=size, markeredgecolor=tuple(col))


class reachabilityConvergence:
    """This class is employed to examine possible ways of convergence of the
    ↪   reachability plot resulting from several OPTICS
    based on different values of MinPts. All the calculations are performed
↪   within class exploreOPTICS, this class
    is only in charge of processing and displaying the results. For the
↪   input:
    filename: In case data is None, file name from which to load data
    data: Data array on which to perform OPTICS analysis
    xi: Float or list of floats, defining the xi or eps values based on which
↪   to perform xi or eps-clustering with OPTICS
    MinPts: Integer or list of integers, defining the list of MinPts based on
↪   which to perform xi or eps-clustering with OPTICS"""
    def __init__(self, filename=None, MinPts=100, xi=0.05, data=None):
        self.MinPts = MinPts
        self.Optics = exploreOPTICS(xi, self.MinPts, 0, filename, data)
        self.errR = []
        self.Optics.sweep()
    def converge(self):
        """This class is in charge of plotting the reachability plots for
        ↪  different values of MinPts in order to examine
        their evolution in shape. Note however that the reachability
↪   distances plotted for each value of MinPts correspond
        to the reachability distances normed by the minimum reachability
↪   distance for that value of MinPts."""
        plt.figure(figsize=(10, 7))
        plt.ylabel('Reachability')
        plt.title('Normed Reachability Plot')
        colors = [plt.cm.inferno(each) for each in np.linspace(0, 1,
        ↪  len(self.MinPts))]
        for i, u in enumerate(self.MinPts):
            legend = 'MinPts = ' + str(np.floor(u))
            space = self.Optics.space[i]
            plt.plot(space, self.Optics.normedR[i], label=legend,
            ↪  color=tuple(colors[i]))

            if i > 0:
                v1 = np.nan_to_num(self.Optics.normedR[i])
                v0 = np.nan_to_num(self.Optics.normedR[i - 1])
                self.errR.append(np.linalg.norm(v1 - v0))
        plt.legend(loc='upper left')

        #This secondary plot displays how the normed reachability varies from
        ↪  MinPts to MinPts, taking into account the
        #dataset as a whole.
        plt.figure()
        plt.title('Variation in Normed Reachability w.r.t. Previous MinPts')
        plt.ylabel('Variation in Normed Reachability')
        plt.xlabel('MinPts')
        plt.plot(self.MinPts[1:], self.errR)

    def convergeDerivative(self):
        """This method is in charge of examining how the derivative of the
        ↪  reachability varies with MinPts."""
        plt.figure(figsize=(10, 7))
        plt.title('Derivative of Reachability')
        for i, u in enumerate(self.Optics.dR):
            legend = 'MinPts = ' + str(np.floor(self.Optics.MinPts[i]))
            space = self.Optics.space[0]

            plt.plot(space[1:], u, label=legend)
        plt.legend(loc='upper left')
    def convergeConcaves(self, plot=True):
        """This methods analyzes how the number of dents within the
         ↪  reachability plot varies with increasing value of
        MinPts. In order to determine what consitutes a dent, the function
↪   employs the gradient of the reachability.
        Moreover, this method determines at which MinPts the number of dents
↪   within the reachability plot more or less
        stabilizes."""
        self.concaves = []

        for i, v in enumerate(self.Optics.reach):
            diffR = np.gradient(v)
            conc = 0
            hasDown = False

            for j, u in enumerate(diffR):
                if u < 0 and not hasDown:
                    hasDown = True

                if u > 0 and hasDown:
                    conc += 1
                    hasDown = False

            self.concaves.append(conc)

        diffConv = np.gradient(self.concaves)
        diffMinPts = np.gradient(self.MinPts)
        converged = np.ndarray.flatten(np.argwhere(np.abs(diffConv /
         ↪  diffMinPts) < 0.1))
        convMinPts = [self.MinPts[converged[0]], self.concaves[converged[0]]]
        self.convMinPts = int(convMinPts[0])

        if plot:
            plt.figure(figsize=(10, 7))
            plt.title('Number of Concave-Up Occurences in Reachability Plot -
             ↪  N = ' + str(self.Optics.N))
            plt.plot(self.MinPts, self.concaves)
            plt.xlabel('MinPts [-]')
            plt.ylabel('Concave-Up Occurences [-]')
            plt.plot(convMinPts[0], convMinPts[1], 'o', color='red')
            font = {'family': 'serif', 'color': 'black', 'weight': 'normal',
             ↪  'size': 12}
            plt.text(convMinPts[0], convMinPts[1] + 5, 'MinPts* = ' +
             ↪  str(int(convMinPts[0])), fontdict=font)
```

```
class compareWithRandom:
    """This class is in charge of, given a single MinPts value, compare the
    ↪  reachability plot resulting from employing
    it in an OPTICS analysis with the reachability plot that results from
↪   applying OPTICS to a randomly distributed set
    of particles of the same lenght as the original dataset. For the input:
    filename: In case data is None, file name from which to load data
    data: Data array on which to perform OPTICS analysis
    xi: Float or list of floats, defining the xi or eps values based on which
↪   to perform xi or eps-clustering with OPTICS
    MinPts: Integer or list of integers, defining the list of MinPts based on
↪   which to perform xi or eps-clustering with OPTICS
    plots: Boolean determing whether to plot the reachability plots of the
↪   dataset and of the randomly distributed set of
    particles."""
    def __init__(self, filename=None, MinPts=100, xi=0.05, data=None,
    ↪  plots=False):
        # At this point it is assumed that MinPts is a single value
        self.MinPts = MinPts
        if np.isscalar(xi):
            self.xi = np.array([xi])
        else:
            self.xi = xi
        self.Optics = exploreOPTICS(xi, MinPts, 0, filename, data)
        self.Optics.sweep()
        self.space = self.Optics.space[0]
        self.reach = self.Optics.reach[0]
        self.N = self.Optics.N
        self.plots = plots

    def randomlyDistributed(self):
        """This method generates a random distribution of particles within
        ↪  the same domain as the original dataset and
        executes an OPTICS analysis on it."""
        maxs = np.array([np.nanmax(self.Optics.data[:, 0]),
        ↪  np.nanmax(self.Optics.data[:, 1])])
        mins = np.array([np.nanmin(self.Optics.data[:, 0]),
        ↪  np.nanmin(self.Optics.data[:, 1])])
        self.points = np.multiply(maxs - mins, np.random.rand(self.N, 2)) +
        ↪  mins
        self.OpticsR = exploreOPTICS(self.xi[0], self.MinPts, 0,
        ↪  filename='Random Distribution', data=self.points)
        self.OpticsR.sweep()
        self.meanReach =
        ↪  np.mean(self.OpticsR.reach[0][np.isfinite(self.OpticsR.reach[0])])
    def compareReachabilities(self):
        """This method plots and compares the reachability distances of the
        ↪  original dataset and those of the randomly
        distributed set of particles."""
        plt.figure(figsize=(10, 7))
        plt.ylabel('Reachability')
        plt.title('Comparison between Reachability Plots - N = ' +
        ↪  str(self.N))
        plt.plot(self.Optics.space[0], self.Optics.reach[0],
        ↪  label='Preferential Distribution')
        plt.plot(self.OpticsR.space[0], self.OpticsR.reach[0], label='Random
        ↪  Distribution')
        plt.legend(loc='upper left')

    def determineXifromRandom(self):
        """Based on the mean reachability obtained from randomly distributing
        ↪  the same number of particles in the same
        domain area, this method determines which value of xi mostly
↪   classifies particles with reachability higher than
        such mean value as noise"""

        self.xiScore = np.zeros(len(self.xi))
        for i, u in enumerate(self.xi):
            c = self.Optics.clust[i]
            labels = c.labels_[c.ordering_]
            for j, v in enumerate(labels):
                if v == -1 and self.isNoise[j] == -1:
                    self.xiScore[i] += 1

                if v != -1 and self.isNoise[j] == 1:
                    self.xiScore[i] += 1

        self.xiScore = self.xiScore / (self.N * 0.01)
        self.optXi = self.xi[np.argmax(self.xiScore)]

        self.Optics.PlotReachabilityWithLabels()
        plt.plot(self.space, self.meanReach * np.ones(len(self.space)))

        plt.figure()
        plt.title('Affinity Score for each Xi')
        plt.ylabel('Score [%]')
        plt.xlabel('xi [-]')
        plt.plot(self.xi, self.xiScore)

    def plotOptimumXi(self):
        """This method plots the reachability plot of the employed dataset
        ↪  and also displays how the optimum xi value
        obtained in the previous method groups the dataset into different
↪   clusters."""
        plt.figure(figsize=(10, 7))
        plt.ylabel('Reachability')
        plt.title('Reachability Plot - MinPts =  ' + str(int(self.MinPts)) +
        ↪  ' & xi* = ' + str(self.optXi))

        plt.plot(self.space, self.reach)

        ofXi = np.ndarray.flatten(np.argwhere(self.xi == self.optXi))[0]
        c = self.Optics.clust[ofXi]
        labels = c.labels_[c.ordering_]
        unique_labels = np.unique(np.array(labels))
        colors = [plt.cm.Dark2(each) for each in np.linspace(0, 1,
        ↪  len(unique_labels))]
        font = {'family': 'serif', 'color': 'black', 'weight': 'normal',
        ↪  'size': 12}
        plt.text(self.space[5], self.meanReach * 1.02, 'Mean Reachability of
        ↪  Random Distribution', fontdict=font)

        for k, col in zip(unique_labels, colors):
            size = 1.6
            if k == -1:
                # Black used for noise.
                col = [128 / 256, 128 / 256, 128 / 256]
                size = 1

            class_member_mask = (labels == k)
            plt.plot(self.space[class_member_mask], self.meanReach *
            ↪  np.ones(len(np.argwhere(class_member_mask))), 'o',
                     markerfacecolor=tuple(col), markersize=size,
                     ↪  markeredgecolor=tuple(col))

    def easyLabel(self):
        """This method simply applies a label value of -1 (Noise) to
        ↪  particles with reachability greater than the obtained
        mean reachability and a label value of 1 (Cluster) to the rest."""
        self.isNoise = np.ones(len(self.space))
        for i, u in enumerate(self.space):
            if self.reach[i] > self.meanReach:
                self.isNoise[i] = -1
        # ORDERING OF LABELS IS NOT THE SAME AS THE ONE IN DATA!!!
        c = self.Optics.clust[0]
        self.easyData = self.Optics.data[c.ordering_, :]
    def run(self):
        """Execution of the previous methods"""
        self.randomlyDistributed()
        if self.plots:
            self.compareReachabilities()
        self.easyLabel()
        if len(self.xi) > 1:
            self.determineXifromRandom()
            self.plotOptimumXi()


# files = [0.005, 0.008, 0.013, 0.017, 0.021, 0.024, 0.028, 0.032, 0.036,
↪  0.041]
#
# filename = './Varying Sampling/' + 'prt_TG_ductVe8_770000' + '_dZ_' +
↪  str(files[-6]).replace('.', ',') + '.csv'
# filename = 'prt_TG_ductVe8_770000_Large_dZ_0,02.csv'

# xi0 = 0.005
# MinPts0 = np.linspace(10,500,30)

# # Exploration of Convergence of Reachability Plots, to obtain MinPts*
# rC = reachabilityConvergence(filename, MinPts0, xi0)
# rC.convergeConcaves()

# MinPts = rC.convMinPts
# xi = np.linspace(0.005, 0.01, 15)

# Comparison with a Random Distribution of Particles, to obtain xi*
# cR = compareWithRandom(filename, MinPts, xi)
# cR.run()
# xi = cR.optXi

# # Traditional OPTICS Explotation
# op = exploreOPTICS(xi, MinPts, 1, filename)
# op.sweep()
# op.PlotReachabilityWithLabels()

# from matplotlib.backends.backend_pdf import PdfPages

# def multipage(filename, figs=None, dpi=200):
#         pp = PdfPages(filename)
#         if figs is None:
#                 figs = [plt.figure(n) for n in plt.get_fignums()]
#         for fig in figs:
```

```
#                 fig.savefig(pp, format='pdf')
#         pp.close()
```

F DBSCAN.py

```
import numpy as np
from sklearn.cluster import DBSCAN
import matplotlib.pyplot as plt
import csv
from clusterPlot import clusterPlot


# Development of data analysis tools for the topological and temporal
↪  analysis of clusters of particles in turbulent flow
# Script Description: This script is designed to execute a DBSCAN clustering
↪  analysis of a 2D snapshot in Z displaying
# particle positions, taking into account multiple combinations of eps and
↪  MinPts. This script is also capable of
# plotting the results of such clustering routine and excluding non-core
↪  elements from the cluster label.
# Álvaro Tomás Gil - UIUC 2020

class exploreDBSCAN:
    """This class is in charge of performing a  DBSCAN analysis of a 2D
    ↪  computational domain displaying particle positions. It
    has the capability of applying the clustering algorithm to the database
↪   for multiple different values of eps and MinPts,
    in order to analyze the effect of these parameters on the clustering of
↪   the data. For the input:
    eps: Float or list of floats, defining the eps values based on which to
↪   perform clustering with DBSCAN
    MinPts: Integer or list of integers, defining the list of MinPts based on
↪   which to perform clustering with DBSCAN
    plots: Number of plots defining, for different values of xi or eps, the
↪   clustering results
    filename: In case data is None, file name from which to load data
    data: Data array on which to perform OPTICS analysis"""
    def __init__(self, eps, MinPts, plots=0, filename=None, data=None):

        if np.isscalar(eps):
            self.eps = np.array([eps])
        else:
            self.eps = eps

        if np.isscalar(MinPts):
            self.MinPts = np.array([MinPts])
        else:
            self.MinPts = MinPts

        if data is not None:
            self.data = data
        else:
            with open(filename, 'r') as f:
                reader = csv.reader(f, delimiter=',')
                self.data = np.array(list(reader)).astype(float)

        self.clust = []
        self.labels = []
        self.n = np.array([0, 0, 0])
        self.N = len(self.data)
        self.pl = [int(v) for v in np.linspace(0, len(self.eps) - 1, plots)]
        print('DBSCAN: ' + str(self.N) + ' particles in data set.')

    def sweep(self):
        """For every combination of the specified values of eps and MinPts,
        ↪  this method carries out the clustering analysis of the
        dataset based on those parameters. The results of such clustering are
↪   stored in clust, and labels."""
        for i, v in enumerate(self.eps):
            for j, u in enumerate(self.MinPts):
                print('        ' + str(i + 1) + 'th epsilon: ' + str(round(v,
                ↪  4)) + '; ' + str(j + 1) + 'th MinPts: ' + str(
                    np.floor(u)))
                c = DBSCAN(eps=v, min_samples=np.floor(u),
                ↪  metric='minkowski', p=2)
                c.fit(self.data)
                self.clust.append(c)
                self.labels.append(c.labels_)

                #Every row of n includes [eps, MinPts, Number of Clusters]
                nC = [v, u, len(np.unique(np.array(c.labels_)))]
                self.n = np.vstack((self.n, nC))

                if i in self.pl:
                    # Plot the clustering disposition of the radii selected
                    ↪  in array pl
                    title = 'DBSCAN with MinPts = ' + str(np.floor(u)) + '
                    ↪  and $\epsilon$ = ' + str(
                        round(v, 4)) + '. ' + str(nC[-1]) + ' clusters
                         ↪  detected.'
                    cP = clusterPlot(self.data, c.labels_)
                    cP.plotAll(title)
        # cP.sizeHistogram(title)

    def onlyCoreElements(self):
        """This method defines a new list of labels in which non-core
        ↪  elements previously included within clusters are
        now excluded from any cluster and labelled as noise."""
        self.labelsN = self.labels
        for i, c in enumerate(self.clust):
            cores = c.core_sample_indices_
            take = np.setdiff1d(np.arange(0, self.N), cores)
            self.labelsN[i][take] = -1

    def plot(self):
        """This function plots the evolution of the detected number of
        ↪  particles with the neighborhood radius eps"""
        self.n = self.n[1:][:]
        fig = plt.figure()
        if len(self.MinPts) == 1:
            plt.plot(self.eps, self.n[:, 2])
            plt.title('DBSCAN with MinPts = ' + str(self.MinPts[0]) + ' and
             ↪  Euclidean Distance')
            plt.xlabel('$\epsilon$ [-]')
            plt.ylabel('Clusters Detected [-]')
        else:
            ax = fig.add_subplot(111, projection='3d')
            ax.scatter(self.n[:, 0], self.n[:, 1], self.n[:, 2], marker='o',
             ↪  c=self.n[:, 2], cmap='inferno')
            ax.set_title('Clusters Detected - DBSCAN with Euclidean
             ↪  Distance')
            ax.set_xlabel('$\epsilon$ [-]')
            ax.set_ylabel('MinPts [-]')
        plt.show()

    # def isolateCluster(self, label):
    #     isolated = np.array([0, 0])
    #     for c in self.clust:
    #         take = self.data[c.labels_ == label]
    #         isolated = np.vstack((isolated, take))
    #
    #     isolated = isolated[1:]
    #     return isolated
```

G clusterPlot.py

```
import numpy as np
import matplotlib.pyplot as plt
import imageio

# Development of data analysis tools for the topological and temporal
↪  analysis of clusters of particles in turbulent flow
# Script Description: This script is in charge of presenting the position and
↪  cluster label of a set of particles whose
# position has been discretized into a set of planes of constant Z. This is
↪  done with a GIF animation, plotting the resulting
# cluster assignments of each two-dimensional domain separatedly.
# Álvaro Tomás Gil - UIUC 2020


class clusterPlot:
    """This script is in charge of presenting the position and cluster label
    ↪  of a set of particles whose
    position has been discretized into a set of planes of constant Z. This is
↪   done with a GIF animation, plotting the resulting
    cluster assignments of each two-dimensional domain separatedly.
    For the input:
    data: 2D array of 3D particle positions, sorted into parallel planes of
↪   constant Z
    labels: list of equal length as data, assigning a cluster label
↪   applicable within each 2D domain. A label
    of -1 corresponds to a void particle.
    folder: folder in which to save resulting plots
    minClusters: minimum number of cluster labels with which to separate
↪   exclusively into void and cluster particles, and
    not within different assigned clusters"""
    def __init__(self, data, labels, folder='', minClusters=3000):
        self.labels = labels
        self.data = data
        self.folder = folder
        if len(np.unique(np.array(labels))) < minClusters:
            self.polyChrome = True
        else:
            self.polyChrome = False

        self.threedee = False
        if np.size(self.data, 1) == 3:
            self.zs = np.unique(self.data[:, 2])
            self.threedee = True
            if len(self.zs) == 1:
                self.data = self.data[:, :2]
                self.threedee = False

        self.unique_labels = np.unique(np.array(labels))
        strength = np.linspace(0, 0.8, len(self.unique_labels))
        np.random.shuffle(strength)
        self.colors = [plt.cm.nipy_spectral(each) for each in strength]
        np.random.shuffle(strength)
        self.colorsB = [plt.cm.nipy_spectral(each) for each in strength]

    def plotAll(self, title):
        if self.threedee:
            dz = abs(self.zs[1] - self.zs[0]) / 4

            def update(choose):
                fig, ax = plt.subplots()
                fig.set_size_inches(18.5, 9.5)

                relevant = ((self.data[:, 2] <= choose + dz) & (self.data[:,
                ↪  2] >= choose - dz))
                self.plotInstance(self.data[relevant, :2],
                                  [self.labels[i] for i in
                                  ↪  range(len(self.labels)) if
                                  ↪  relevant[i]], ax)

                ax.set_title('Z = ' + '{0:03f}'.format(choose), fontsize=24)
                ax.set_xlabel('x [m]', fontsize=18)
                ax.set_ylabel('y [m]', fontsize=18)
                ax.tick_params(axis='both', which='major', labelsize=15)
                ax.set_xlim(np.min(self.data[:, 0]), np.max(self.data[:, 0]))
                ax.set_ylim(np.min(self.data[:, 1]), np.max(self.data[:, 1]))
                plt.axis('equal')

                fig.canvas.draw()
                image = np.frombuffer(fig.canvas.tostring_rgb(),
                ↪  dtype='uint8')
                image = image.reshape(fig.canvas.get_width_height()[::-1] +
                ↪  (3,))
                plt.close()
                return image

            kwargs_write = {'fps': 1.0, 'quantizer': 'nq'}
            imageio.mimsave(self.folder + title + '.gif', [update(i) for i in
            ↪  self.zs], fps=2)
        else:
            fig, ax = plt.subplots()
            fig.set_size_inches(18.5, 9.5)
            ax.set_title(title)
            ax.set_xlabel('x [m]')
            ax.set_ylabel('y [m]')
            plt.axis('equal')
            self.plotInstance(self.data, self.labels, ax)

    def plotInstance(self, data, labels, ax):
        if self.polyChrome:

            for k, col, colB in zip(self.unique_labels, self.colors,
            ↪  self.colorsB):
                size = 3
                if k == -1:
                    # Black used for noise.
                    col = [1, 0, 0]
                    size = 1

                class_member_mask = (labels == k)
                xy = data[class_member_mask]
                if len(xy) > 0:
                    ax.scatter(xy[:, 0], xy[:, 1],
                    ↪  c=np.reshape(np.array(col), (1, -1)),
                               edgecolors=np.reshape(np.array(colB), (1,
                               ↪  -1)), s=30, label='Cluster ' + str(k))
            ax.legend()
        else:
            s = 0.8 + (labels != -1) * 0.8
            col = {1: 'blue', 0: 'grey'}
            c = [col[d] for d in (labels != -1)]
            ax.scatter(data[:, 0], data[:, 1], s=s, c=c)

    def sizeHistogram(self, title):
        """This method plots a histogram of the number of particles contained
        ↪  within each DBSCAN cluster"""
        unique, counts = np.unique(np.array(self.labels), return_counts=True)
        plt.figure()
        plt.hist(counts, bins=100, density=True)
        plt.title(title)
        plt.xlabel('Particles per Cluster [-]')
        plt.ylabel('Fraction of Clusters [-]')
```

H boundaryFinder.py

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import NearestNeighbors
import time
from skeletonPlot import skeletonPlot

# Development of data analysis tools for the topological and temporal
↪  analysis of clusters of particles in turbulent flow
# Script Description: This script is in charge of determining, for a set of
↪  isolated particle clusters within different
# snapshots in Z, the cluster particles which form the boundary of the
↪  cluster. In order to do so, the main class
# "boundaryFinder" invokes, for every snapshot in Z, the class
↪  "boundaryTraveller", the latter obtaining a closed particle
# boundary for each of the clusters in the dataset.
# Álvaro Tomás Gil - UIUC 2020

class boundaryFinder:
    """This is the main class of the script, in charge of calling the
    ↪  secondary class for every Z value in the dataset,
    performing an additional polishing of the obtained cluster boundaries,
↪   and plotting the resulting boundaries. For the
    input:
    data: 2D Array containing the 3D positions of all particles, arranged
↪   into a discrete set of Z levels, to which the
    particle positions have been normally projected
    labels: List which labels each of the elements in data based on their
↪   DBSCAN cluster label. In case this label is -1,
    the referenced element corresponds to a void particle
    eps: Float defining the eps value based on which clustering analysis with
↪   DBSCAN has been performed. This is used as a
    reference length in this method.
    folder: folder in which to save resulting plots"""
    def __init__(self, data, labels, eps, folder=''):
        self.data = data
        self.labels = labels
        self.eps = eps
        self.folder = folder
        self.zs = np.unique(self.data[:, 2])
        self.fakeSkel = np.array([0, 0, 0])


    def bound2dUnsupervised(self, z):
        """This method is in charge of invoking the secondary class in order
        ↪  to perform the boundary analysis of the clusters
        for which Z is the same as the specified value."""
        self.bT = boundaryTraveller(z, self.data, self.labels, self.eps)
        boundary, plane, cluster = self.bT.run()
        return boundary, plane, cluster

    def sweep(self, runs=1, plot3d=True, plot=[]):
        """This method is central to the current class, as it sweeps for the
        ↪  different Z values existing in the dataset,
        and calls the cluster boundary determination. Moreover, this method
↪   calls the method in charge of polishing
        the resulting cluster boundaries and plots the final results."""
        start = time.time()
        print('Sweep along Z of Boundary Finder')
        self.boundaries = []
        self.bound3d = np.array([0, 0, 0])
        self.clusters = np.array([0, 0, 0])

        for i, z in enumerate(self.zs):
            print('Z = ' + str(round(z, 3)))
            complete = []
            for j in range(runs):
                boundary, plane, cluster = self.bound2dUnsupervised(z)
                # Fake skel is an array whose exclusive function is for
                ↪  plotting the resulting cluster boundaries
                self.fakeSkel = np.vstack((self.fakeSkel, cluster[0, :]))
                complete = complete + boundary
            boundary = list(set(complete))
            self.boundaries.append(boundary)

            #bound3d is the 2d array of all cluster boundary particle
            ↪  positions
            self.bound3d = np.vstack((self.bound3d, plane[boundary, :]))
            self.clusters = np.vstack((self.clusters, cluster))


        print('Execution Time: ' + str(round(time.time() - start, 3)))

        self.bound3d = self.bound3d[1:, :]
        self.clusters = self.clusters[1:, :]

        self.polishBoundaries()

        if plot3d:
            self.plotAll()
        for i in plot:
            z = self.zs[i]
            self.boundaryPlot(z)

    def polishBoundaries(self, p=0.2, kb=0.3):
        """This method polishes the obtained cluster boundaries by deleting
        ↪  boundary particles with less than a fraction
        p of the average number of neighboring boundary particles within a
↪   neighborhood of kb*epsilon"""
        self.nbrs = NearestNeighbors(radius=kb * self.eps, algorithm='auto')
        self.nbrs.fit(self.bound3d)
        dis, ind = self.nbrs.radius_neighbors()
        neighs = []
        for i, v in enumerate(ind):
            neighs.append(len(v))

        ave = np.mean(neighs)
        keep = [i for i in range(len(neighs)) if neighs[i] > p * ave]
        print('Ave Neighs: ' + str(ave) + '. Min Neighs: ' +
        ↪  str(np.min(neighs)) + '. Killed: ' + str(
            len(self.bound3d) - len(keep)))
        self.bound3d = self.bound3d[keep, :]

    def boundaryPlot(self, z):
        """Given an array of boundary particle positions 'boundary', this
        ↪  method plots these results. """
        take = self.bound3d[:, 2] == z
        boundary = self.bound3d[take, :]
        take = self.clusters[:, 2] == z
        cluster = self.clusters[take, :]
        fig, ax = plt.subplots()
        fig.set_size_inches(18.5, 9.5)
        ax.set_title('Boundary Particles of Cluster for Z = ' +
        ↪  '{0:03f}'.format(boundary[0, 2]), fontsize=23)
        ax.set_xlabel('x [m]', fontsize=18)
        ax.set_ylabel('y [m]', fontsize=18)
        plt.axis('equal')
        ax.scatter(boundary[:, 0], boundary[:, 1], s=0.8, c='red')
        ax.scatter(cluster[:, 0], cluster[:, 1], s=0.8, c='blue')

    def plotAll(self):
        """This method invokes the plotting of cluster boundaries for all of
        ↪  the considered Z levels"""
        self.fakeSkel = self.fakeSkel[1:]
        sP = skeletonPlot(self.bound3d, self.fakeSkel, folder=self.folder)
        sP.snapPlot(title='Lagrangian Boundary')


class boundaryTraveller:
    """For a particular Z level, this class is in charge of determining the
    ↪  cluster particles which represent cluster
    boundaries. For the cluster particles contained in less dense regions,
↪   the algorithm proceeds by measuring the angles
    between successive particle-neighboring particle vectors. If any of these
↪   angles for a particular cluster particle
    is larger than a specified threshold value, the cluster particle is
↪   considered to be a boundary, and the neighboring
    particles adjacent to such gap are stored for later analysis. For the
↪   inputs:
    z: Float describing the z value of data to analyze
    data: 2D Array containing the 3D positions of all particles, arranged
↪   into a discrete set of Z levels, to which the
    particle positions have been normally projected
    labels: List which labels each of the elements in data based on their
↪   DBSCAN cluster label. In case this label is -1,
    the referenced element corresponds to a void particle
    radius: Float defining the reference length to employ for the
↪   neighborhood radius of each particle under analysis
    dirs: Dividing 2*Pi by this float, one obtains the minimum separation
↪   angle between adjacent neighboring particles
    for the particle under analysis to be considered as a boundary
    k: Multiplied by radius, this float defines the neighborhood radius based
↪   on which the neighbors of each particle are
    analyzed
    dense: Proportion of the maximum number density for a particle to be
↪   considered by this algorithm
    """
    def __init__(self, z, data, labels, radius=0.001, dirs=6, k=0.5,
    ↪  dense=0.75):
        self.z = z
        self.dir = dirs
        self.plane = data[data[:, 2] == z, :2]
        self.labels2d = labels[data[:, 2] == z]
        self.cluster = self.plane[self.labels2d != -1, :] #only cluster
        ↪  particles
        self.clustInd = np.ndarray.flatten(np.argwhere(self.labels2d != -1))
        ↪  # references plane

        self.nbrs = NearestNeighbors(radius=radius * k,
        ↪  algorithm='auto').fit(self.cluster)
        self.distances, self.indices =
        ↪   self.nbrs.radius_neighbors(self.cluster)

        self.N = []
        for i in self.indices:
            self.N.append(len(i))

        self.notdense = [i for i in range(len(self.N)) if self.N[i] <
        ↪   dense*np.max(self.N)] #extracts less dense particles

        self.taken = np.array(
            [])  # Taken and Queue both index Cluster and ClustInd, whileas
            ↪   ClustInd references self.plane
        self.queue = np.array([])
        self.boundary = []  # References self.plane

        self.theta = [(i - 1) * 2 * np.pi / self.dir for i in range(1,
        ↪   self.dir + 1)]
        self.ri = np.array([[np.cos(th), np.sin(th)] for th in self.theta])

        self.test = np.random.randint(0, len(self.cluster))

        print('        boundaryTraveller: dirs = ' + str(dirs) + '; k = ' +
        ↪   str(k) + ';')

    def nextInLine(self):
        """This method is in charge of determine, at each iteration of the
        ↪   algorithm, which particle to analyze next. If
        the queue of particles is empty, a particle from the less denser
↪   group of particles is randomly sampled."""
        if len(self.queue) > 0:
            take = self.queue[0]
            self.queue = np.delete(self.queue, 0)
        else:
            poss = np.setdiff1d(self.notdense, self.taken)
            take = poss[np.random.randint(0, len(poss))]

        self.taken = np.append(self.taken, take)
        return int(take)

    def substractAngles(self, this, dumping=False):
        """Given a particle indexed by 'this', this method examines all of
        ↪   its neighboring particles and measures the
        angles between adjacent particle-neighboring particle vectors. All
↪   neighboring particles for which the angle
```

```
        of separation is greater than the specified threshold are stored in
↪    'adj' to be added to the queue. If dumping
        is allowed, all the other neighboring particles are excluded from
↪    further processing by the algorithm via 'dump'."""
        pos = self.cluster[this, :]
        neighs = self.indices[this]
        neighs = neighs[neighs != this]
        vecs = np.array([0, 0])
        angs = np.array([])

        for i in neighs:
            that = self.cluster[i, :]
            mag = np.linalg.norm(that - pos)
            if mag > 0:
                vecs = np.vstack((vecs, (that - pos) / mag))
                theta = np.arctan2(vecs[-1, 1], vecs[-1, 0])
                if theta < 0:
                    theta += 2 * np.pi
                angs = np.append(angs, theta)

        if len(angs) > 0:
            vecs = vecs[1:, :]
            sortI = np.argsort(angs)
            angs = np.sort(angs)

            sortI = np.append(sortI, sortI[0])
            angs = np.append(angs, angs[0] + 2 * np.pi)

            delta = np.diff(angs)
            gaps = np.flatnonzero(delta >= 2 * np.pi / self.dir)
            adjacent = np.concatenate((gaps, gaps + 1))
            adj = np.unique(neighs[sortI[adjacent]])

            if dumping:
                dump = [i for i in neighs if i not in adj]
            else:
                dump = []

        # print('Newline:                          ', angs*180/np.pi,
        ↪   delta*180/np.pi, adjacent, adj)

        else:
            adj = []
            dump = []

        return adj, dump
```

```
    def processPoint(self, this):
        """For a cluster particle indexed via 'this', this method extracts
        ↪  whether this particle can be considered as a
        cluster boundary particle. It also stores relevant neighboring
↪   particles of this particle in the particle
        queue for further analysis, and all dumped neighboring particles in
↪   the list of taken particles to exclude them
        from further analysis."""
        take, discard = self.substractAngles(this)

        if len(take) > 0:
            self.boundary.append(int(self.clustInd[this]))
            add2queue = [v for v in take if v not in self.taken]
            self.queue = np.append(self.queue, add2queue)

        if len(discard) > 0:
            add2taken = [v for v in discard if v not in self.taken]
            self.taken = np.append(self.taken, add2taken)

    def run(self):
        """This method executes the boundary determining algorithm, by
        ↪  sampling a cluster particle as per
        nextInLine and processing such particle with processPoint. What
↪   results is boundary, a list of indexes of
        boundary particles referencing plane, plane, an array of particle
↪   positions for the current Z level, and cluster,
        an array of cluster particle positions for the current Z level."""
        while len(self.taken) < len(self.notdense):
            this = self.nextInLine()
            # print('Next Point in Line: ', this)
            self.processPoint(this)
        # print(self.boundary.shape, self.plane.shape, self.cluster.shape)
        self.plane = np.hstack((self.plane, self.z *
        ↪  np.ones((len(self.plane), 1))))
        self.cluster = self.plane[np.setdiff1d(self.clustInd, self.boundary),
        ↪  :]
        return self.boundary, self.plane, self.cluster
```

I skeletonPlot.py

```
import numpy as np
from matplotlib import cm
import matplotlib.pyplot as plt
import imageio


# Development of data analysis tools for the topological and temporal
↪  analysis of clusters of particles in turbulent flow
# Script Description: This script is in charge of plotting, either by means
↪  of a frozen 3D figure or by means of a GIF file,
# the constellation of skeleton points interior to each two-dimensional
↪  cluster along with the boundaries of each cluster.
# Álvaro Tomás Gil - UIUC 2020

class skeletonPlot:
    """This class is in charge of plotting, either by means of a frozen 3D
    ↪  figure or by means of a GIF file,
    the constellation of skeleton points interior to each two-dimensional
↪   cluster along with the boundaries of each cluster.
    For the inputs:
    bound3d: 2D array of 3D positions corresponding to the curves defining
↪   each cluster's boundaries for each of the
    parallel planes of constant Z in the domain
    skel: 2D array of 3D positions of skeleton points. In case skel2 is not
↪   None, this could also represent the positions
    of the points defining the trajectory between skeleton points with which
↪   the connectivity between them is examined.
    skel2: 2D array of 3D positions of skeleton points, in the case in which
↪   the connectivity between them is displayed
    labels: list of the same length as skel2, assigning a cluster label to
↪   each skeleton point
    folder: folder in which to save resulting plots"""
    def __init__(self, bound3d, skel, skel2=None, labels=None, folder=''):
        self.bound3d = bound3d
        self.skel = skel[np.argsort(skel[:, 2]), :][::-1]
        self.zs = np.unique(bound3d[:, 2])
        self.skel2 = skel2
        self.labels = labels
        self.folder = folder

        # Combinations of marker and marker boundary colors are made in order
        ↪  to increase the possibilities of marker types
        self.unique_labels = np.unique(self.labels)
        strength = np.linspace(0, 0.8, len(self.unique_labels))
        np.random.shuffle(strength)
        self.colors = [plt.cm.nipy_spectral(each) for each in strength]
        np.random.shuffle(strength)
        self.colorsB = [plt.cm.nipy_spectral(each) for each in strength]

        # Trajectory and boundary points are colored in terms of the Z value
        ↪   they appear in
        self.cm = cm.get_cmap('winter')
        normalized = (self.skel[:, 2] - np.min(self.zs)) / (np.ptp(self.zs))
        self.skelC = self.cm(normalized)
        normalized = (self.bound3d[:, 2] - np.min(self.zs)) /
        ↪   (np.ptp(self.zs))
        self.bounC = self.cm(normalized)

    def messyPlot(self, title='Topological Skelletonization of Clusters',
    ↪   labelB='BPs', labelS='Skeleton'):
        """This method defines a single 3D figure in which both cluster
        ↪   boundaries and skeleton points are displayed"""
        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3d')
        fig.set_size_inches(18.5, 9.5)
        ax.set_title(title)
        ax.set_xlabel('x [m]')
        ax.set_ylabel('y [m]')
        ax.set_zlabel('z [m]')
        ax.scatter(self.bound3d[:, 0], self.bound3d[:, 1], self.bound3d[:,
        ↪   2], alpha=0.5, s=0.5, label=labelB)
        ax.scatter(self.skel[:, 0], self.skel[:, 1], self.skel[:, 2],
        ↪   c='red', s=0.5, alpha=1, label=labelS)
        ax.legend()

        X = self.bound3d[:, 0]
        Y = self.bound3d[:, 1]
        Z = self.bound3d[:, 2]
        # From
        ↪   https://stackoverflow.com/questions/13685386/matplotlib-equal-unit-length-with-e
        # Create cubic bounding box to simulate equal aspect ratio
        max_range = np.array([X.max() - X.min(), Y.max() - Y.min(), Z.max() -
        ↪   Z.min()]).max()
        Xb = 0.5 * max_range * np.mgrid[-1:2:2, -1:2:2, -1:2:2][0].flatten()
        ↪   + 0.5 * (X.max() + X.min())
        Yb = 0.5 * max_range * np.mgrid[-1:2:2, -1:2:2, -1:2:2][1].flatten()
        ↪   + 0.5 * (Y.max() + Y.min())
        Zb = 0.5 * max_range * np.mgrid[-1:2:2, -1:2:2, -1:2:2][2].flatten()
        ↪   + 0.5 * (Z.max() + Z.min())
        # Comment or uncomment following both lines to test the fake bounding
        ↪   box:
        for xb, yb, zb in zip(Xb, Yb, Zb):
            ax.plot([xb], [yb], [zb], 'w')

    def snapPlot(self, title, this=True, labelB='BPs', labelS='Skeleton'):
        """This method plots each two-dimensional domain separately, but
        ↪   joins all of them in a GIF animation which allows
        a 3D evolution of them to be visualized."""
        self.this = this
        self.labelB = labelB
        self.labelS = labelS

        def update(choose):
            fig, ax = plt.subplots()
            fig.set_size_inches(18.5, 9.5)

            self.plotInstance(choose, ax)
            ax.set_title('Z = ' + '{0:03f}'.format(choose), fontsize=24)
            ax.set_xlabel('x [m]', fontsize=18)
            ax.set_ylabel('y [m]', fontsize=18)
            ax.tick_params(axis='both', which='major', labelsize=15)
            plt.axis('equal')
            ax.set_xlim(np.min(self.bound3d[:, 0]), np.max(self.bound3d[:,
            ↪   0]))
            ax.set_ylim(np.min(self.bound3d[:, 1]), np.max(self.bound3d[:,
            ↪   1]))

            fig.canvas.draw()
            image = np.frombuffer(fig.canvas.tostring_rgb(), dtype='uint8')
            image = image.reshape(fig.canvas.get_width_height()[::-1] + (3,))
            plt.close()
            return image

        kwargs_write = {'fps': 1.0, 'quantizer': 'nq'}
        imageio.mimsave(self.folder + title + '.gif', [update(i) for i in
        ↪   self.zs], fps=2)

    def plotInstance(self, choose, ax):
        if len(self.zs) >= 2:
            dz = (self.zs[1] - self.zs[0]) / 4
        else:
            dz = 0.001 * self.zs[0]

        take = self.bound3d[:, 2] == choose
        if self.this:
            takeS = ((self.skel[:, 2] >= choose - dz) & (self.skel[:, 2] <=
            ↪  choose + dz))
        else:
            takeS = self.skel[:, 2] >= choose
        ax.scatter(self.bound3d[take, 0], self.bound3d[take, 1],
        ↪  c=self.bounC[take], s=1, label=self.labelB)
        ax.scatter(self.skel[takeS, 0], self.skel[takeS, 1],
        ↪  c=self.skelC[takeS], s=5, label=self.labelS, marker='D')

        if self.labels is not None:
            takeS2 = ((self.skel2[:, 2] >= choose - dz) & (self.skel2[:, 2]
            ↪  <= choose + dz))
            data = self.skel2[takeS2, :]
            labels = [self.labels[i] for i in range(len(self.labels)) if
            ↪  takeS2[i]]
            self.plotLabels(ax, data, labels)

    def plotLabels(self, ax, data, labels):

        for k, col, colB in zip(self.unique_labels, self.colors,
        ↪  self.colorsB):
            size = 15
            if k == 0:
                # Black used for noise.
                col = [1, 0, 0]
                size = 1

            class_member_mask = (labels == k)
            xy = data[class_member_mask]
            if len(xy) > 0:
                ax.scatter(xy[:, 0], xy[:, 1], c=np.reshape(np.array(col),
                ↪  (1, -1)),
                           edgecolors=np.reshape(np.array(colB), (1, -1)),
                           ↪  s=50, marker='P',
                           label='SP of Cluster ' + str(k))
        ax.legend()
```

J eulerianApproach.py

```
import numpy as np
import matplotlib.pyplot as plt
from clusterPlot import clusterPlot
from skeletonPlot import skeletonPlot
from sklearn.neighbors import NearestNeighbors
import time

# Development of data analysis tools for the topological and temporal
↪  analysis of clusters of particles in turbulent flow
# Script Description: This script is designed to analyze a DBSCAN-labelled
↪  array of particle positions in order to discretize its 2D
# domain in an Eulerian way, determine cluster boundaries based on which of
↪  the Eulerian cells are populated by particles labelled
# by DBSCAN as cluster particles, compute the area associated to each DBSCAN
↪  label, and determine an Eulerian skeleton with Eulerian
# cell centers in areas labelled as cluster areas by DBSCAN.
# Álvaro Tomás Gil - UIUC 2020


class eulerianAnalysis:
    """This class discretizes the 2D domain and determines which Eulerian
    ↪  cells are populated by cluster particles. It also determines
    cluster boundaries based on populated cells which are neighbors to
↪   non-populated cells.
    cluster: 3-column array of cluster particles, sorted by 2D snapshots in Z
    labels: Array of labels, labeling each element in cluster according to a
↪   previous DBSCAN analysis.
    eps: Neighborhood radius of the previous DBSCAN analysis, used as a
↪   reference length
    kx, ky: Constants determining Eulerian cell sizes in X and in Y when
↪   multiplied by eps
    boundMethod: Specifies the method to be used to determine cluster
↪   boundaries
    lagBounds: Array of cluster boundaries obtained by "boundaryTraveller",
↪   to use as a reference for Eulerian cell sizes
    if kx and ky are None
    folder: folder in which to save resulting plots"""
    def __init__(self, cluster, labels, eps, kx=0.2, ky=0.2,
    ↪  boundMethod='basic', lagBounds=None, folder=''):
        print('Eulerian Analysis of Boundaries and Areas')
        cluster = cluster[labels != -1, :]
        self.clusters = cluster
        labels = labels[labels != -1]
        self.folder = folder
        self.cluster = cluster
        self.labels = labels
        self.eps = eps
        self.method = boundMethod
        self.reduce = cluster
        self.redL = labels

        if kx is None or ky is None:
            kx = self.determineDims(lagBounds=lagBounds)
            ky = kx

        self.dx = kx * eps
        self.dy = ky * eps
        self.zs = np.unique(cluster[:, 2])

        self.xrange = [np.min(cluster[:, 0]), np.max(cluster[:, 0])]
        self.yrange = [np.min(cluster[:, 1]), np.max(cluster[:, 1])]
        self.xn = int(np.floor((self.xrange[1] - self.xrange[0]) / self.dx))
        self.yn = int(np.floor((self.yrange[1] - self.yrange[0]) / self.dy))

        self.x = np.linspace(self.xrange[0], self.xrange[1], self.xn)
        self.y = np.linspace(self.yrange[0], self.yrange[1], self.yn)
        self.dx = self.x[1] - self.x[0]
        self.dy = self.y[1] - self.y[0]
        self.a = self.dx*self.dy

        self.xc = np.linspace(self.dx/2 + self.x[0], self.x[-1] - self.dx/2,
        ↪   self.xn - 1)
        self.yc = np.linspace(self.dy / 2 + self.y[0], self.y[-1] - self.dy /
        ↪   2, self.yn - 1)
        self.isfull = -1*np.ones((self.xn - 1, self.yn - 1, len(self.zs)))

        #rc: 2D array containing the Eulerian cell centers for all snapshots
        ↪   in Z
        #lc: Label array defining whether a cluster populates an Eulerian
        ↪   cell, and in such case, of which DBSCAN label
        self.rc = np.zeros(((self.xn - 1) * (self.yn - 1) * len(self.zs), 3))
        self.lc = -1*np.ones(((self.xn - 1) * (self.yn - 1) * len(self.zs)))
        self.bound3d = np.array([0, 0, 0])
        self.fakeSkel = np.array([0, 0, 0])
        self.areas = [0 for _ in self.zs]
        self.labelcount = [[] for _ in self.zs]

    def determineDims(self, lagBounds):
        """This method is in charge of assigning Eulerian cell dimensions
        ↪   based on the average separation between an already existing
        cluster boundary"""
        if lagBounds is None:
            lagrangian = np.load('lagrangianBound3d.npy')
        else:
            lagrangian = lagBounds
        nbrs = NearestNeighbors(n_neighbors=1).fit(lagrangian)
        dist, _ = nbrs.kneighbors()
        dist = np.ndarray.flatten(dist)
        k = np.mean(dist) / self.eps
        print(' Based on the mean distance between boundary particles in the
        ↪  Lagrangian boundary, k = ' + str(round(k, 3)))
        return k

    def testSnapshot(self):
        """This method is in charge of testing the whole routine for a single
        ↪  Z snapshot"""
        i = 0
        take = np.arange(len(self.cluster[:, 2]))[self.cluster[:, 2] ==
        ↪  self.zs[i]]
        self.fakeSkel = np.vstack((self.fakeSkel, self.cluster[take[0]]))
        self.fullI = np.array([0, 0])
        self.filloccupancies(i)
        return (len(self.fullI) - 1)/((self.xn - 1)*(self.yn - 1))

    def run(self):
        """Generic run command. The output of this command is a 2D array
        ↪  containing the coordinates of all cluster boundary points.
        ↪  (bound3d)"""
        start = time.time()
        for k,z in enumerate(self.zs):
            self.forZ(k)
            print(' Cells per DBSCAN label for Z = ' + str(round(z, 3)) +
            ↪  ':')
            for key in sorted(self.labelcount[k]):
                print("     %s: %s" % (key, self.labelcount[k][key]))
        self.fakeSkel = self.fakeSkel[1:, :]

        self.plotisfull()
        if self.method is not None:
            self.bound3d = self.bound3d[1:, :]
            self.plotBoundary()
        print('Execution Time: ' + str(round(time.time() - start, 3)))


    def forZ(self, i):
        """Execution for every snapshot in Z"""
        take = np.arange(len(self.cluster[:, 2]))[self.cluster[:, 2] ==
        ↪  self.zs[i]]
        self.fakeSkel = np.vstack((self.fakeSkel, self.cluster[take[0]]))
        self.fullI = np.array([0, 0])
        self.filloccupancies(i)

        if self.method == 'skimage':
            self.skimageBound(i)
        elif self.method is not None:
            self.seeneighbors(i)

    def filloccupancies(self, k):
        """This method is in charge of sweeping along the 2D domain,
        ↪  determining whether each Eulerian cell is populated by
        cluster particles or not. This result is stored in:
        isfull: This is a 3D array of indices corresponding to the Eulerian
↪   grid (column, row, Zsnapshot) contains the label
        assigned to every Eulerian cell, which is either the DBSCAN label of
↪   the cluster particles within it, or -1
        rc and lc: These two 2D arrays both display what isfull does without
↪   employing 3D array schematics"""
        z = self.zs[k]
        areas = {}
        labelcount = {}
        tz = np.arange(len(self.reduce[:, 2]))[self.reduce[:, 2] == z]
        rz, self.reduce = self.newanddelete(tz, self.reduce)
        lz, self.redL = self.newanddelete(tz, self.redL)

        for i,x in enumerate(self.xc):
            x1 = self.x[i]
            x2 = self.x[i + 1]
            tx = np.arange(len(rz))[((rz[:, 0] > x1) & (rz[:, 0] < x2))]
            rx, rz = self.newanddelete(tx, rz)
            lx, lz = self.newanddelete(tx, lz)
            for j,y in enumerate(self.yc):
                y1 = self.y[j]
                y2 = self.y[j + 1]
                ty = np.arange(len(rx))[((rx[:, 1] > y1) & (rx[:, 1] < y2))]
                ry, rx = self.newanddelete(ty, rx)
                ly, lx = self.newanddelete(ty, lx)
                self.rc[self.longindex(i, j, k), :] = np.array([x, y, z])
                if len(ty) > 0:
                    winner = np.random.choice(ly, 1)[0]
                    self.isfull[i, j, k] = winner
                    self.lc[self.longindex(i, j, k)] = winner
                    self.fullI = np.vstack((self.fullI, np.array([i, j])))

                    if winner not in areas.keys():
                        areas[winner] = self.a
                        labelcount[winner] = 1
                    else:
                        areas[winner] += self.a
                        labelcount[winner] += 1
                else:
                    self.lc[self.longindex(i, j, k)] = -1
        self.areas[k] = areas
        self.labelcount[k] = labelcount

    def seeneighbors(self, k):
        """This method is in charge of, for every populated Eulerian cell,
        ↪  determining if one of the directly neighboring cells
        are populated by cluster particles."""
        self.fullI = self.fullI[1:, :]
        for indices in self.fullI:
            i = indices[0]
            j = indices[1]
            if j + 1 > self.yn - 2:
                right = True
            else:
                right = self.isfull[i, j + 1, k] == -1

            if j - 1 < 0:
                left = True
            else:
                left = self.isfull[i, j - 1, k] == -1

            if i - 1 < 0:
                up = True
            else:
                up = self.isfull[i - 1, j, k] == -1

            if i + 1 > self.xn - 2:
                down = True
            else:
                down = self.isfull[i + 1, j, k] == -1

            if any([up, down, left, right]):
                self.bound3d = np.vstack((self.bound3d,
                 ↪  self.rc[self.longindex(i, j, k), :]))

    def skimageBound(self, k):
        """Boundary determination employing skimage, where the input is a
        ↪  black-and-white image corresponding to the Eulerian grid"""
        from skimage import measure
        self.fullI = self.fullI[1:, :]
        canvas = np.zeros(self.isfull.shape[:2])
        for indices in self.fullI:
            i = indices[0]
            j = indices[1]
            canvas[i, j] = 1

        contours = measure.find_contours(canvas, 0)
        if type(contours) is not list:
            contours = [contours]

        for c in contours:
            for r in c:
                i = int(r[0])
                j = int(r[1])
                self.bound3d = np.vstack((self.bound3d,
                 ↪  self.rc[self.longindex(i, j, k), :]))


    def longindex(self, i, j, k):
        """Converts three indices used in the 3D array to two indices used in
        ↪  2D array equivalent"""
        r = j + (self.yn - 1)*i + (self.yn - 1)*(self.xn - 1)*k
        return r

    @staticmethod
    def newanddelete(index, array):
        """Method in charge of extracting and deleting an element from an
        ↪  array"""
        new = array[index]
        array = np.delete(array, index, axis=0)
        return new, array

    def plotisfull(self):
        """Plotting of the Eulerian grid represented in isfull"""
        cP = clusterPlot(self.rc, self.lc, self.folder)
        cP.plotAll('IsFull')

    def plotBoundary(self):
        """Plotting of the resulting cluster boundaries"""
        sP = skeletonPlot(self.bound3d, self.fakeSkel, folder=self.folder)
        sP.snapPlot(title='Eulerian Boundary')

class eulerianSkeleton:
    """This class is in charge of creating a brute skeleton defining the
    ↪  cluster's shape based on Eulerian cell centers which have,
    not only a sufficient separation from any predefined cluster boundary,
↪   but also a separation from all other skeleton particles
    within the cluster. The input to this class is:
    rc and lc: Eulerian cell centers and labels, as per generated in the
↪   previous class
    bound3d: 2D array of coordinates of cluster boundary points
    eps: Neighborhood radius of the previous DBSCAN analysis, used as a
↪   reference length
    ks: Defines, when multiplied by eps, the minimum separation between
↪   skeleton particles
    kc: Defines, when multiplied by kc, the minimum separation to hold
↪   between skeleton particles and boundary particles
    folder: folder in which to save resulting plots
    """
    def __init__(self, rc, lc, bound3d, eps, ks=0.8, kc=0.15, folder=''):
        self.rc = rc
        self.lc = lc
        self.bound3d = bound3d
        self.eps = eps
        self.dx = abs(rc[0, 1] - rc[1, 1])
        self.ds = eps*ks
        self.crash = kc*eps
        self.folder = folder

        self.zs = np.unique(self.rc[:, 2])
        self.skel = np.array([0, 0, 0])

    def run(self, title='EulerianSkeleton'):
        print('Topological Skeletonization of Clusters')
        start = time.time()
        for z in self.zs:
            print(' Z = ' + str(round(z, 3)))
            self.forZ(z)

        self.skel = self.skel[1:, :]
        print('Execution Time: ' + str(round(time.time() - start, 3)))
        self.plotSkel(title)

    def forZ(self, z):
        releC = ((self.rc[:, 2] == z) & (self.lc != -1))
        self.clusters = self.rc[releC, :]
        self.labels = self.lc[releC]

        releB = self.bound3d[:, 2] == z
        self.boundary = self.bound3d[releB, :]
        self.nbrs = NearestNeighbors(n_neighbors=1).fit(self.boundary)
        for l in np.unique(self.labels):
            self.forCluster(l)


    def forCluster(self, l, ksample=0.3):
        """This method analyzes a single DBSCAN-labelled cluster in a single
        ↪  2D snapshot in Z, and determines a set of Eulerian cell
        centers which are the farthest away possible from all boundary
↪   particles and are separated enough from each other."""
        cluster = self.clusters[self.labels == l]
        cluster = cluster[np.random.choice(np.arange(len(cluster)),
        ↪  max(int(ksample*len(cluster)), 1))]
        cluNbrs = NearestNeighbors(radius=50*self.ds).fit(cluster)
        score = np.array([])
        for i,r in enumerate(cluster):
            minD, _ = self.nbrs.kneighbors(np.reshape(r, (1, -1)))
            score = np.append(score, minD)

        taken = [np.argmax(score)] #Array defining which Eulerian cell
        ↪  centers already define the skelleton
        done = False
        it = 0
        while not done:
            it += 1
            #Every taken cell has a best ranked, far enough, not taken cell
            distances, indices = cluNbrs.radius_neighbors(cluster[taken, :])
            far = []
            for i, row in enumerate(distances):
                farenough = indices[i][row > self.ds] #Which Eulerian cell
                ↪  centers are far enough from the current skeleton
                ↪  particle?
                farvirgin = [x for x in farenough if x not in taken] #Which
                ↪  of these are not already taken?
                farvirgin = [f for f in farvirgin if score[f] > self.crash]
                ↪  #Which of these are not too close to cluster boundaries?
                if len(far) == 0:
                    far = farvirgin
                else:
                    far = [f for f in far if f in farvirgin]
                    if len(far) == 0:
                        done = True #If no candidates for skeleton particles
                        ↪  exist, stop
                        break

            if len(far) > 0:
                finalround = score[far]
                taken.append(far[np.argmax(finalround)])

            if it > 100:
                done = True

        self.skel = np.vstack((self.skel, cluster[taken, :]))

    def plotSkel(self, title='EulerianSkeleton'):
        self.plot = skeletonPlot(self.bound3d, self.skel, folder=self.folder)
        self.plot.snapPlot(title)


class optimumCellSize:
    """Class used to analyze the effect of cell size on the number of
    ↪ populated Eulerian cells"""
    def __init__(self, cluster, labels, eps, k0=0.088, k1=0.3):
        ks = np.linspace(k0, k1)
        filled = np.zeros(len(ks))
        for i,k in enumerate(ks):
            app = eulerianAnalysis(cluster, labels, eps, kx=k, ky=k)
            filled[i] = app.testSnapshot()*100

        plt.figure()
        plt.plot(ks, filled)
        plt.xlabel('Cell Size [-]')
        plt.ylabel('Percentage of Populated Cells [%]')
        plt.title('Evolution of Filled Cells with Cell Size')


# eu = eulerianAnalysis(np.load('allPoints.npy'),
↪ np.load('DBSCANlabels.npy'), np.load('eps.npy'), boundMethod=None,
↪ kx=0.2, ky=0.2)
# eu.run()
# bound3d = np.load('bound3d.npy')
# # rc = np.load('cellCenters.npy')
# # lc = np.load('cellLabels.npy')
# eps = np.load('eps.npy')
# rc = eu.rc
# lc = eu.lc
#
# z1 = rc[0, 2]
# z2 = rc[-1, 2]
# take = ((rc[:, 2] == z1) | (rc[:, 2] == z2))
# takeB = ((bound3d[:, 2] == z1) | (bound3d[:, 2] == z2))
# sk = eulerianSkeleton(rc[take], lc[take], bound3d[takeB], eps, 1.6)
# sk.run('test')
```

K clusterConnect.py

```
import numpy as np
from sklearn.neighbors import NearestNeighbors
import time
from skeletonPlot import skeletonPlot

# Development of data analysis tools for the topological and temporal
↪  analysis of clusters of particles in turbulent flow
# Script Description: Based on the resulting Brute Skeleton representation of
↪  several DBSCAN clusters within different levels of Z,
# this script is in charge of examining the connectivity of every brute
↪  skeleton point with all others, by determining whether
# a straight line between both skeleton points collides with a cluster
↪  boundary or not. In order to examine connectivities along
# different levels of Z, the projection of the skeleton point to the other
↪  level of Z is used instead of the original skeleton point.
# Álvaro Tomás Gil - UIUC 2020

class skeletonConnect:
    """This is the main class of the script. Based on the resulting Brute
    ↪  Skeleton representation of several DBSCAN clusters within different
    ↪  levels of Z,
    this script is in charge of examining the connectivity of every brute
↪   skeleton point with all others, by determining whether
    a straight line between both skeleton points collides with a cluster
↪   boundary or not. In order to examine connectivities along
    different levels of Z, the projection of the skeleton point to the other
↪   level of Z is used instead of the original skeleton point.
    For the inputs:
    skel: 2D array of 3D skeleton point positions
    bound3d: 2D array of 3D boundary particle positions
    eps: float, Neighborhood radius of the previous DBSCAN analysis, used as
↪   a reference length
    kb: float, Determines the neighborhood distance with which to extract
↪   relevant skeleton positions, by multiplying it with eps
    kc: float, Determines the neighborhood distance under which to assume
↪   that a trajectory between two skeleton points has collided
    with a cluster boundary, by multiplying it with eps
    ks: float, Determines the separation distance between adjacent points
↪   defining the trajectory connecting two skeleton particles,
    by multiplying it by eps
    folder: folder in which to save resulting plots"""
    def __init__(self, skel, bound3d, eps, kb=5, kc=0.1, ks=0.02, folder=''):
    self.skel = skel
    self.bound3d = bound3d
    self.eps = eps
    self.folder = folder

    self.ds = ks * eps
    self.zs = np.unique(bound3d[:, 2])
    self.dz = np.abs(self.zs[1] - self.zs[0]) / 4

    #Neighborhood of skeleton particles
    self.nbrs = NearestNeighbors(radius=kb * eps,
    ↪  algorithm='auto').fit(skel)

    #Neighborhood of boundary particles, for collisions
    self.collision = NearestNeighbors(radius=kc * eps,
    ↪  algorithm='auto').fit(bound3d)

    #For every skeleton particle, this list contains a list of connected
    ↪  skeleton particles
    self.connexions = [[] for i in range(len(self.skel))]
    self.trajs = np.array([0, 0, 0])

    def run(self):
    """Main method of the class, in charge of its execution."""
    print('Determination of Skeleton Connectivity')
    start = time.time()
    for i in range(len(self.skel)):
        connexions, trajs = self.forSkelP(i)
        if trajs is not None:
            self.connexions[i] = connexions
            self.trajs = np.vstack((self.trajs, trajs))

    self.trajs = self.trajs[1:, :]
    self.mutualizer()
    self.labeller()
    self.plotTrajs()
    print('Execution Time: ' + str(round(time.time() - start, 3)))

    def forSkelP(self, i):
    """For a single skeleton point, defined by index i, this method
    ↪  examines whether it can be connected to nearby skeleton
    particles with straight trajectories, taking into account the point's
↪   projections on the two adjacent levels of Z"""
```

```
        r = self.skel[i, :2]
        thisZ = self.skel[i, 2]
        thisZi = np.argmin(np.abs(self.zs - thisZ))

        #allZ samples adjacent Z levels as well as the Z level of skeleton
        ↪   point i
        allZ = self.zs[max(0, thisZi - 1):min(len(self.zs), thisZi + 1) + 1]
        connexions = []
        trajs = np.array([0, 0, 0])

        for z in allZ:
            r1 = np.append(r, z)
            conn, traj = self.findNeigh(r1, thisZ)
            if traj is not None:
                connexions = connexions + conn
                trajs = np.vstack((trajs, traj))

        if len(trajs.shape) == 2:
            trajs = trajs[1:, :]
        else:
            trajs = None
        return connexions, trajs

    def findNeigh(self, r, originalZ):
        """Given a skeleton point position r, this method finds the nearby
        ↪   skeleton positions which can be connected
        by means of a straight trajectory, and outputs the indices of the
↪   connected neighbors as well as the connecting
        trajectory points."""
        thisZ = r[2]
        distance, indexes = self.nbrs.radius_neighbors(np.reshape(r, (1,
        ↪   -1)))
        indexes = indexes[0]
        distance = distance[0]
        indexes = indexes[distance > 1e-10]

        take = ((self.skel[indexes, 2] >= thisZ - self.dz) &
        ↪   (self.skel[indexes, 2] <= thisZ + self.dz))
        indexes = indexes[take]
        connexions = []
        trajs = np.array([0, 0, 0])

        for i in indexes:
            r2 = self.skel[i, :]
            conn, traj = self.areConnected(r, r2, originalZ)
            if conn:
                connexions.append(i)
                trajs = np.vstack((trajs, traj[::5, :]))

        if len(trajs.shape) == 2:
            trajs = trajs[1:, :]
        else:
            trajs = None

        return connexions, trajs

    def areConnected(self, r1, r2, originalZ):
        """Given two positions r1 and r2, this method is in charge of
         ↪  determining whether a straight line between both
        points is at any point closer than kc * eps to a cluster boundary. If
↪   this is not so, both points are said to be connected,
        (connected = True), and the trajectory between both is returned. In
↪   the case in which r1 is not an actual skeleton
        point but rather a projection into an adjacent level in Z, this
↪   method returns a different trajectory than the one
        employed to study connectivities. This trajectory is the straight
↪   line between the original skeleton point at its original
        Z and the second skeleton point."""
        thisZ = r1[2]
        connected = True
        traj = self.defTraj(r1, r2)
        indexes = self.collision.radius_neighbors(traj,
         ↪  return_distance=False)
        for i, v in enumerate(traj):
            ind = indexes[i]
            take = ((self.bound3d[ind, 2] >= thisZ - self.dz) &
             ↪  (self.bound3d[ind, 2] <= thisZ + self.dz))
            ind = ind[take]

            if len(ind) > 0:
                connected = False
                break

        if connected and originalZ != thisZ:
            r1[2] = originalZ
            traj = self.defTraj(r1, r2)

        return connected, traj

    def defTraj(self, r1, r2):
        """This simple method outputs the points belonging to the straight
        ↪  trajectory between points r1 and r2"""
        numPts = np.ceil(np.linalg.norm(r1 - r2) / self.ds)
        traj = np.linspace(r1, r2, numPts)
        return traj

    def plotTrajs(self):
        """This method invokes class skeletonPlot to define a descriptive
        ↪  plot of the determined connections"""
        self.plot = skeletonPlot(self.bound3d, self.trajs, skel2=self.skel,
        ↪  labels=self.labels, folder=self.folder)
        # self.plot.messyPlot(title = 'Connectivity of Skeleton Points',
        ↪  labelS = 'Connexions')
        self.plot.snapPlot(title='Connectivity of Skeleton Points',
        ↪  labelS='Connexions')

    def mutualizer(self):
        """Based on a list of connexions, this method makes sure that if i is
        ↪  included in the connexions of j, so is j included
        in the connexions of i."""
        for i, c in enumerate(self.connexions):
            for j in c:
                if i not in self.connexions[j]:
                    self.connexions[j].append(i)

    def labeller(self):
        """Based on the defined connexions, this method is in charge of
        ↪  assigning a label to each skeleton particle, such that
        connected skeleton points have the same label, but only the lowest
↪   possible labels are assigned."""
        self.labels = [0 for i in range(len(self.skel))]
        print('        Assignment of Cluster Labels')

        for _ in range(1):
            for i, c in enumerate(self.connexions):
                if self.labels[i] == 0:
                    self.labels[i] = max(self.labels) + 1
                # print('SP: ' + str(i) + '. Connexions: ' + str(c) + '. Li:
                ↪  ' + str(self.labels[i]))
                for j in c:
                    # print('SP: ' + str(j) + '. Lj: ' + str(self.labels[j]))
                    l = self.labels[j]
                    current = self.labels[i]
                    if l == 0:
                        self.labels[j] = current
                    elif l < current and l > 0:
                        self.mergeLabels(l, current)
                    elif l > current:
                        self.mergeLabels(current, l)
            un, counts = np.unique(self.labels, return_counts=True)
            res = [(un[i], counts[i]) for i in range(len(un))]
            print('        Cluster Labels and Frequencies: ', res)

    def mergeLabels(self, winner, loser):
        """In the case in which two already labelled skeleton points are
        ↪  found to be connected, this method determines
        which label should be propagated (winner), and which label (loser)
↪   should be substituted."""
        # print('Winner: ' + str(winner) + '. Loser: ' + str(loser))
        for i, l in enumerate(self.labels):
            if l == loser:
                self.labels[i] = winner

        self.new = self.labels.copy()
        for i, l in enumerate(np.unique(self.labels)):
            for j, k in enumerate(self.labels):
                if k == l:
                    self.new[j] = i

        self.labels = self.new.copy()
```

L relabeller.py

```
import numpy as np
from sklearn.neighbors import NearestNeighbors
import matplotlib.pyplot as plt
from clusterPlot import clusterPlot
import time

# Development of data analysis tools for the topological and temporal
↪   analysis of clusters of particles in turbulent flow
# Script Description: Based on a previous DBSCAN cluster labeling for every
↪   2D snapshot in Z, and a set of labeled skeleton
# particles, which takes into account their connectivity across different 2D
↪   snapshots in Z, this script is in charge of
# replacing the old DBSCAN label assigned to every cluster particle by the
↪   label of the skeleton particles within each cluster.
# For every skeleton particle, the closest cluster particle is determined,
↪   and its DBSCAN label is extracted. Then, all
```

```
# cluster particles sharing such DBSCAN label have their cluster label set to
↪  the label of the skeleton particle. This script is
# also in charge of determining the area in each 2D snapshot in Z which
↪  belongs to each of the new skeleton particle labels.
# Álvaro Tomás Gil - UIUC 2020

class relabeller:
        """Based on a previous DBSCAN cluster labeling for every 2D snapshot
         ↪  in Z, and a set of labeled skeleton
        particles, which takes into account their connectivity across
↪  different 2D snapshots in Z, this script is in charge of
        replacing the old DBSCAN label assigned to every cluster particle by
↪  the label of the skeleton particles within each cluster.
        For every skeleton particle, the closest cluster particle is
↪  determined, and its DBSCAN label is extracted. Then, all
        cluster particles sharing such DBSCAN label have their cluster label
↪  set to the label of the skeleton particle. This script is
        also in charge of determining the area in each 2D snapshot in Z which
↪  belongs to each of the new skeleton particle labels. Inputs:
        data: 2D array containing all particle locations projected into a set
↪  of 2D planes for different values of Z
        oldlab: Array containing the DBSCAN labels assigned to each particle
↪  of data. As many elements as data. Label -1 corresponds to non-cluster
↪  particles
        skel: 2D array containing the positions of all skeleton particles
↪  describing the cluster's interior
        newlab: Array of labels assigned to every skeleton particles. As many
↪  elements as skel.
        oldAreas: List of dicts containing as many elements as Z levels.
↪  Every dict has, for a DBSCAN label as a key, the assined area as a value
        folder: folder in which to save resulting plots"""
        def __init__(self, data, oldlab, skel, newlab, oldAreas, folder=''):
                self.data = data
                self.skel = skel
                self.newlab = newlab
                self.oldAreas = oldAreas
                self.folder = folder

                self.zs = np.unique(data[:, 2])
                self.dz = np.abs(self.zs[1] - self.zs[0])/4
                self.dZ = np.abs(self.zs[1] - self.zs[0])
                self.clusters = data[oldlab != -1]
                self.oldlab = oldlab[oldlab != -1]
                self.nextL = max(self.newlab)
```

```
            self.new = np.zeros(len(self.oldlab)) #0 for _ in
            ↪  range(len(self.oldlab))]
            self.nbrs = NearestNeighbors(n_neighbors = 1, algorithm =
            ↪  'auto').fit(self.clusters)

    def run(self):
            print('Relabelling of Cluster Particles')
            start = time.time()
            for i, z in enumerate(self.zs):
                    print('        Z = ' + str(round(z, 4)) + ' -
                    ↪  Percentage Relabelled: ' +
                    ↪  str(round(len(np.nonzero(self.new)[0])*100/len(self.new),
                    ↪  3)) + '%.')
                    self.forZsnap(z)

            self.verify()
            print('Execution Time: ' + str(round(time.time() - start,
            ↪  3)))
            self.labelPlot()
            self.computeVolume()


    def forZsnap(self, thisZ):
            """Method in charge of isolating cluster and skeleton
            ↪  particles relevant to the current 2D snapshot in Z and
            ↪  executing the procedure"""
            #relevantS indexes skel with skeleton particles within the Z
            ↪  level described by thisZ
            self.relevantS = np.arange(len(self.skel))[((self.skel[:, 2]
            ↪  >= thisZ - self.dz) & (self.skel[:, 2] <= thisZ +
            ↪  self.dz))]
            self.spneigh = NearestNeighbors(n_neighbors=15,
            ↪  algorithm='auto').fit(self.skel[self.relevantS])
            # relevantC indexes clusters with cluster particles within
            ↪  the Z level described by thisZ
            self.relevantC =
            ↪  np.arange(len(self.clusters))[((self.clusters[:, 2] >=
            ↪  thisZ - self.dz) & (self.clusters[:, 2] <= thisZ +
            ↪  self.dz))]
            for i in self.relevantS:
                    self.forSkelPoint(i)
            self.assignZeros()


    def forSkelPoint(self, i):
```

```
                """Method in charge of extracting a skeleton particle, its
                ↪  label, its closest cluster particle, and the label of the
                ↪  latter. Then,
                the method calls updateLabels"""
                sp = self.skel[i, :]
                closest = self.nbrs.kneighbors(np.reshape(sp, (1, -1)),
                ↪  return_distance = False)
                loserL = self.oldlab[closest[0][0]]
                winnerL = self.newlab[i]

                self.updateLabels(winnerL, loserL)

        def updateLabels(self, winner, loser):
                """Method in charge of examining all cluster particles with
                ↪  label 'loser' and  assigning label 'winner' to them"""
                take = [(self.new[i] == 0 and self.oldlab[i] == loser) for i
                ↪  in self.relevantC]
                intersect = [(self.new[i] != 0 and self.oldlab[i] == loser
                ↪  and self.new[i] != winner) for i in self.relevantC]
                toChange = self.relevantC[take] #these cluster particles
                ↪  havent had their labels substituted by one of an SP
                ↪  before
                toJudge = self.relevantC[intersect] #these have, but such
                ↪  label is different than the one currently imposed

                for i in toChange:
                        self.new[i] = winner

                for i in toJudge: #What if a cluster particle has already had
                ↪  its label updated?
                        self.new[i] = self.decideIntersect(winner,
                         ↪  self.new[i], i)


        def decideIntersect(self, l1, l2, i):
                """This method is in charge of resolving a conflict in new
                ↪  label assignment. If a cluster particle is to be
                ↪  relabelled
                with a skeleton particle label when it has already been
↪  assigned another skeleton particle label, this cluster particle
                is to be assigned the skeleton particle label of the skeleton
↪  particle which is closest of the two."""
                r = self.clusters[i, :]
                SP1 = [j for j in self.relevantS if self.newlab[j] == l1]
                SP2 = [j for j in self.relevantS if self.newlab[j] == l2]
                dist, closest = self.spneigh.kneighbors(np.reshape(r, (1,
                ↪  -1)))
                # closest indexes relevantS, not skel!
                # relevantS, and thus SP1 and SP2 do index skel
                winner = 0
                for sp, d in enumerate(dist[0]):
                        if self.relevantS[closest[0][sp]] in SP1:
                                winner = l1
                        elif self.relevantS[closest[0][sp]] in SP2:
                                winner = l2

                        if winner > 0:
                                break

                if winner == 0:
                        print('        Intersection between SP labels ' +
                        ↪  str(l1) + ' and ' + str(l2) + ' yielded no
                        ↪  result.')
                        sp = self.relevantS[closest[0][sp]]
                        winner = self.newlab[sp]
                        print('        Winner is then SP label ' +
                        ↪  str(winner))
                return winner

        def assignZeros(self):
                """Since it may be the case that smaller clusters are not
                ↪  assigned a skeleton particle, this method is in charge of
                generating a new skeleton particle label exclusively for
↪  these clusters."""
                abandoned = [i for i in self.relevantC if self.new[i] == 0]
                labels = np.unique([self.oldlab[i] for i in abandoned])
                for l in labels:
                        self.nextL += 1
                        assign = [i for i in abandoned if self.oldlab[i] ==
                        ↪  l]
                        for j in assign:
                                self.new[j] = self.nextL

        def verify(self):
                """Once the relabeling procedure is finished, this method is
                ↪  in charge of verifying if all skeleton particle labels
                coincide with the labels of the closest cluster particles."""
                self.agree = [False for i in range(len(self.skel))]
                for i,sp in enumerate(self.skel):
                        closest = self.nbrs.kneighbors(np.reshape(sp, (1,
                        ↪  -1)), return_distance = False)
                if self.new[closest[0][0]] == self.newlab[i]:
                        self.agree[i] = True
                else:
                        print('        SP ' + str(i) + ' at ' +
                        ↪  str(sp) + ' with label ' +
                        ↪  str(self.newlab[i]) + ' has a cluster
                        ↪  label of ' +
                        ↪  str(self.new[closest[0][0]]))

        print('        From the ' + str(len(self.skel)) + ' skeleton
        ↪  particles present, ' + str(sum(self.agree)) + ' (' +
        ↪  str(round(sum(self.agree)*100/len(self.skel), 3)) + '%)
        ↪  coincide with their relabelled cluster label.')


def labelPlot(self):
        """Plotting of relabelled clusters."""
        self.plot = clusterPlot(self.clusters, self.new, self.folder)
        self.plot.plotAll('Relabelled Clusters')

def computeVolume(self):
        """Based on previously areas per new label, the algorithm
        ↪  estimates the cluster's volume based on a linear
        ↪  interpolation."""
        self.createIndex()
        self.volumes = [0 for _ in np.unique(self.new)]

        for i, z in enumerate(self.zs[:-1]):
                areas1 = self.newAreas[i + 1]
                areas0 = self.newAreas[i]
                for k in areas0.keys():
                        if k in areas1.keys():
                                self.volumes[k - 1] += 0.5 *
                                ↪  (areas0[k] + areas1[k]) * self.dZ
                        else:
                                self.volumes[k - 1] += 0.5 *
                                ↪  areas0[k] * self.dZ

                for k in areas1.keys():
                        if k not in areas0.keys():
                                self.volumes[k - 1] += 0.5 *
                                ↪  areas1[k] * self.dZ

        self.volumePlot()

def createIndex(self):
```

```
                """For every initial DBSCAN label, this method determines to
                ↪  which new skeleton particle label this label has been
                ↪  converted.
                This method is also in charge of comparing old and new
↪  labelling of cluster particles, and based on the areas per 2D snapshot in
↪  Z
                assigned to each old cluster label, compute the areas per 2D
↪  snapshot assigned to each new cluster label. """
                self.new = self.new.astype(int)
                self.newAreas = [0 for _ in self.zs]
                for i,z in enumerate(self.zs):
                        oldAreas = self.oldAreas[i]
                        areadict = {}
                        self.relevantC = np.arange(len(self.clusters))[
                                ((self.clusters[:, 2] >= z - self.dz) &
                                ↪  (self.clusters[:, 2] <= z + self.dz))]
                        maxNew = np.max(self.new[self.relevantC])
                        maxOld = np.max(self.oldlab[self.relevantC])
                        index = np.zeros((maxOld + 1, maxNew))
                        for j, l1, l2 in zip(self.relevantC,
                        ↪  self.oldlab[self.relevantC],
                        ↪  self.new[self.relevantC]):
                                index[l1, l2 - 1] += 1

                        #For every old label, divide each of the transitions
                        ↪  to each skeleton particle label by the total in
                        ↪  the old label
                        for j, row in enumerate(index):
                                if np.sum(row) == 0:
                                        print('e')
                                index[j] = row/np.sum(row)

                        #For every new label, compute the associated area
                        ↪  based on the old estimated areas
                        for j, col in enumerate(np.transpose(index)):
                                total = 0
                                for k, fraction in enumerate(col):
                                        if k in oldAreas.keys():
                                                total += oldAreas[k]*fraction

                                areadict[j + 1] = total

                        self.newAreas[i] = areadict

        def volumePlot(self, top=10):
                """This method is simply in charge of plotting a bar plot
                ↪  comparing cluster volumes"""
                fig = plt.figure()
                fig.set_size_inches(18.5, 9.5)
                ax = fig.add_subplot(111)
                label = ['Cluster ' + str(i) for i in range(1,
                ↪  len(self.volumes) + 1)]

                volumesC = np.sort(self.volumes)[::-1][:top]
                sortI = np.argsort(self.volumes)[::-1][:top]
                label = [label[i] for i in sortI]

                cmap = plt.get_cmap('plasma')
                c = cmap(volumesC)

                ax.bar(range(top), volumesC, tick_label=label, width=0.5,
                ↪  color=c)
                ax.tick_params(labelsize=18)
                plt.ylabel('Volume [m^3]', fontsize=18)
                plt.title('Volume per Cluster', fontsize=22)
                plt.savefig(self.folder + 'Volume per Cluster')
                plt.close('all')
```

M temporalTracking.py

```
import numpy as np
from sklearn.neighbors import NearestNeighbors
from temporalPlot import temporalPlot
import matplotlib.pyplot as plt

# Development of data analysis tools for the topological and temporal
↪  analysis of clusters of particles in turbulent flow
# Script Description: This script is designed to analyze an already-processed
↪  pair of sets of particle positions, for adjacent instants of time.
# The carried out analysis is basically focused on comparing neighboring
↪  topologies for skeleton points of both time instants.
# For each skeleton point, the distance to the closest boundary particle in
↪  every direction is measured, and then each skeleton
# point of the first time instant is paired with the skeleton point of the
↪  second time frame which has the most similar set
# of measured distances and which is within an area of expected translation.
↪  The main output of this analysis is an array
```

```
# of transitions which, for every cluster label of the first time frame,
↪  takes into account the cluster labels which they
# theoretically have adopted in the second time frame.
# Álvaro Tomás Gil - UIUC 2020

class temporalTracker:
    """This class is designed to analyze an already-processed pair of sets of
    ↪  particle positions, for adjacent instants of time.
    The carried out analysis is basically focused on comparing neighboring
↪   topologies for skeleton points of both time instants.
    For each skeleton point, the distance to the closest boundary particle in
↪   every direction is measured, and then each skeleton
    point of the first time instant is paired with the skeleton point of the
↪   second time frame which has the most similar set
    of measured distances and which is within an area of expected
↪   translation. The main output of this analysis is an array
    of transitions which, for every cluster label of the first time frame,
↪   takes into account the cluster labels which they
    theoretically have adopted in the second time frame.
    For the inputs:
    pairs: List of two ints, containing the time instants to compare
    v: Float representing the channel flow bulk velocity associated to the
↪   datasets
    urms: Fluctuating particle velocity, proportional to the flow bulk
↪   velocity
    ksens: Float, which when multiplied with the distance that a particle at
↪   v travels between both time instants, defines
    the neighborhood radius with which to carry out the distance measurements
↪   for each skeleton point
    kfocus: Float, which when multiplied with the distance that a particle at
↪   v travels between both time instants, describes
    the area of interest in which to look for skeleton points in the second
↪   time frame which have similar distance measures
    dirs: Number of distance measurements to carry out for each skeleton
↪   point
    root: Address from which to retrieve time instant data"""
    def __init__(self, pair, v=7.7 * 0.15e-3 / 100, urms=0.1, ksens=3,
    ↪  kfocus=1, dirs=10, root='./Data/v78/t_'):
        self.v = v
        self.pair = pair

        self.add = [root + str(p) + '/' for p in pair]
        self.d = v * (pair[1] - pair[0])
        eps = np.mean([np.load(a + 'eps.npy') for a in self.add])
        self.sensorRadius = ksens * eps
        self.regionRadius = [urms * self.d, kfocus]

        self.sk = [np.load(a + 'skeletonize.npy') for a in self.add]
        self.bound = [np.load(a + 'bound3d.npy') for a in self.add]
        self.labels = [np.load(a + 'SKlabels.npy') for a in self.add]
        self.volumes = [np.load(a + 'DBSCANVolumes.npy') for a in self.add]
        self.nbrs = [NearestNeighbors(radius=self.sensorRadius,
        ↪  algorithm='auto').fit(b) for b in self.bound]
        self.zs = np.unique(self.sk[0][:, 2])
        self.connections = np.array([0, 0, 0])
        self.connAngles = []

        #Transitions has as many rows as labels in the old time instant, and
        ↪  as many columns as labels in the new time
        # instant. By taking the i-th row of the dataset, the j-th column
        ↪  represents how many skeleton points from the
        # cluster label i in the old time frame are paired with a cluster
        ↪  label j in the new time instant
        self.transitions = np.zeros((len(np.unique(self.labels[0])),
        ↪  len(np.unique(self.labels[1])) + 1))
        self.paired = np.hstack((np.reshape(self.labels[0], (-1, 1)),
        ↪  np.zeros((len(self.labels[0]), 1))))
        self.pairedString = [['t = ' + str(self.pair[0]) + ' - Cluster ' +
        ↪  str(int(i)), 0] for i in self.labels[0]]

        self.dirs = np.array([0, 0, 0])
        self.corresponding = np.array([])
        for theta in np.linspace(0, 2 * np.pi, dirs):
            x = np.cos(theta)
            y = np.sin(theta)
            z = 0
            self.dirs = np.vstack((self.dirs, np.array([x, y, z])))
            self.corresponding = np.append(self.corresponding, theta)
        self.dirs = self.dirs[1:, :]

    def run(self):
        self.proximities = [self.proximities2D(self.sk[i], i) for i in
        ↪  range(2)]

        self.aveD = np.mean(self.proximities[0], axis=1)  # Mean distance to
        ↪  neighboring boundary particles
        self.regionRadii = self.regionRadius[0] + self.regionRadius[1] *
        ↪  self.aveD
        self.focus = NearestNeighbors(radius=np.max(self.regionRadii),
        ↪  algorithm='auto').fit(self.sk[1])
        for i, sk in enumerate(self.sk[0]):
            self.forSkelPoint(i)

        print('Out of ' + str(len(self.sk[0])) + ' skeleton points of the
        ↪   first time frame, ' + str(sum(self.transitions[:, 0])) + ' (' +
        ↪   str(
            round(sum(self.transitions[:, 0]) * 100 / len(self.sk[0]),
                  3)) + ' %) are not paired with another skeleton point in
                  ↪   the second time frame')

        print('Connecting Vectors are on average oriented ' +
        ↪   str(round(np.mean(self.connAngles) * 180 / np.pi, 3)) + ' degrees
        ↪   wrt the X axis')

        for l in np.unique(self.labels[0]):
            members = sum(self.labels[0] == l)
            topNew = np.argsort(self.transitions[l - 1, :])[::-1][:5]
            counts = np.sort(self.transitions[l - 1, :])[::-1][:5]
            print('For the ' + str(members) + ' skeleton points in the first
            ↪   time instant with label ' + str(l) + ': ')
            for i in range(5):
                if counts[i] > 0:
                    if topNew[i] == 0:
                        add = ' ' + str(counts[i]) + ' ( ' +
                        ↪   str(round(counts[i] * 100 / members, 3)) + ' %)
                        ↪   are not paired with any skeleton point from the
                        ↪   next time instant'
                    else:
                        add = ' ' + str(counts[i]) + ' ( ' +
                        ↪   str(round(counts[i] * 100/ members, 3)) + ' %)
                        ↪   are paired in the next time instant with cluster
                        ↪   label ' + str(topNew[i])
                    print(add)

        self.plotResults()

    def proximities2D(self, skel, i):
        """This method measures, for every skeleton point in skel, the
        ↪   distances to the closest boundary particle in each
        direction. These distances in each direction are returned via
↪   proximities, which is an array witg as many rows
        as skel and as many columns as dirs. For the input:
        skel: 2D array of 3D skeleton positions
        i: Int dictating which time instant to take into account"""
        distAll, indiAll = self.nbrs[i].radius_neighbors(skel)
        proximities = np.zeros((1, len(self.dirs)))
        for j, x in enumerate(skel):
            dist = distAll[j]
            if len(dist) > 0:
                indi = indiAll[j]
                vecs = self.bound[i][indi, :] - x

                dots = np.matmul(vecs, np.transpose(self.dirs))  # As many
                ↪  rows as len(vecs), as many columns as len(dirs)
                maxDot = np.argmax(dots, axis=1)  # as many elements as vecs,
                ↪  indexes dirs
                covered = []  # indexes vecs, same len as dirs
                for k in range(len(self.dirs)):
                    intheway = np.ndarray.flatten(np.argwhere(maxDot == k))
                    ↪  # indexes vecs
                    if len(intheway) == 1:
                        covered.append(intheway[0])
                    elif len(intheway) > 1:
                        d = dist[intheway]
                        take = np.argmin(d)
                        covered.append(intheway[take])
                    else:
                        covered.append(-1)

                dist = np.append(dist, self.sensorRadius)
                distances = dist[covered]
            else:
                distances = self.sensorRadius * np.ones((1, len(self.dirs)))
            proximities = np.vstack((proximities, distances))
        return proximities[1:]


    def forSkelPoint(self, i, dt=0.15):
        """For the skeleton point of the first time frame specified by index
        ↪  i, this method looks for the neighboring
        skeleton points of the next time frame close to the points expected
↪  position, and extracts the skeleton point of
        the next time frame which has the closest measures of proximitiy
↪  distances, as well as its cluster label. dt is
        the actual time step in ms"""
        r = self.sk[0][i, :]
        label = self.labels[0][i]
        distances = self.proximities[0][i, :]
        rexpected = r + self.d * np.array([1, 0, 0])

        possible = self.focus.radius_neighbors(np.reshape(rexpected, (1,
        ↪  -1)), return_distance=False, radius=self.regionRadii[i])
        possible = [p for p in possible[0] if self.sk[1][p, 2] == r[2]]
        toCompare = self.proximities[1][possible, :]

        deviations = np.sum((toCompare - distances)**2, axis=1)
        if len(deviations) == 0:
            newLabel = 0
        else:
            closest = possible[np.argmin(deviations)]
            newLabel = self.labels[1][closest]
            connection = np.linspace(r, self.sk[1][closest, :], 100)
            self.connections = np.vstack((self.connections, connection))
            angle = np.arctan2(connection[-1, 1] - connection[0, 1],
            ↪  connection[-1, 0] - connection[0, 0])
            self.connAngles.append(angle)
        self.transitions[label - 1, newLabel] += 1
        self.paired[i, 1] = newLabel
        if newLabel == 0:
            self.pairedString[i][1] = 't = ' + str(self.pair[1] * dt / 100) +
            ↪  ' ms - Unpaired'
        else:
            self.pairedString[i][1] = 't = ' + str(self.pair[1] * dt / 100) +
            ↪  ' ms - Cluster ' + str(newLabel)

    def plotResults(self):
        self.plot = temporalPlot(self.sk, self.bound, self.labels,
        ↪  self.connections, self.add[1])
        self.plot.snapPlot()
        self.plot.sankeyPlot(self.pairedString)

class temporalTrackerGlobal:
    """This class simply invokes temporalTracker for more than one pair of
    ↪  time instants, and is able to track topology
    evolutions over a greater extent of time.
    For the inputs:
    pairs: 2D list where each element is a pair of time instants to examine
    dt: Time step for 100 iterations"""
    def __init__(self, pairs, dt=0.15, root='./Data/v78/t_'):
        self.pairs = pairs
        self.trackers = [0 for _ in pairs]
        self.transitions = [0 for _ in pairs]
        self.volumes = []
        self.times = []
        for i, p in enumerate(pairs):
            tt = temporalTracker(p, root=root)
            tt.run()
            self.trackers[i] = tt
            self.transitions[i] = tt.transitions
            self.volumes.append([t for t in tt.volumes[0]])
            self.times.append(p[0] * dt / 100)
            if i==len(pairs) - 1:
                self.volumes.append([t for t in tt.volumes[1]])
                self.times.append(p[1] * dt / 100)

    def trackCluster(self, c=1):
        """This method tracks a certain cluster c, by observing the cluster
        ↪  to which its majority of skeleton points
        transform into, and plotting the number of skeleton points and volume
↪   of such cluster at each time instant"""
        volumes = np.zeros(len(self.pairs) + 1)
        skel = np.zeros(len(self.pairs) + 1)
        cluster = c

        for i, p in enumerate(self.pairs):
            volumes[i] = self.volumes[i][cluster - 1]
            skel[i] = np.sum(self.transitions[i][cluster - 1, :])
            cluster = np.argmax(self.transitions[i][cluster - 1, 1:]) + 1
            if i == len(self.pairs) - 1:
                volumes[i + 1] = self.volumes[i + 1][cluster - 1]
                skel[i + 1] = sum(self.trackers[i].labels[1] == cluster)

        fig, ax1 = plt.subplots()
        fig.set_size_inches(16, 11)
        color = 'tab:red'
        ax1.set_xlabel('Time [ms]', fontsize=18)
        plt.xticks(fontsize=15)
        plt.grid()
        ax1.set_ylabel('Cluster Volume [m^3]', color=color, fontsize=18)
        ax1.plot(self.times, volumes, color=color)
        ax1.tick_params(axis='y', labelcolor=color, labelsize=15)
        ax1.set_title('Evolution of Volume and Number of Skeleton Points of
        ↪  Cluster ' + str(c), fontsize=20)

        ax2 = ax1.twinx()
        color = 'tab:blue'
        ax2.set_ylabel('Member Skeleton Points [-]', color=color,
        ↪  fontsize=18)
        ax2.plot(self.times, skel, color=color)
        ax2.tick_params(axis='y', labelcolor=color, labelsize=15)
        fig.tight_layout()

# pairs = [[0, 100], [100, 200], [200, 300], [300, 400], [400, 500]]
pairs = [[0, 500], [500, 1000], [1000, 1500], [1500, 2000]]
```

```
tt = temporalTrackerGlobal(pairs, root='./Data/t_')
tt.trackCluster(1)
plt.show()
```

N temporalPlot.py

```
import numpy as np
from matplotlib import cm
import matplotlib.pyplot as plt
import imageio


# Development of data analysis tools for the topological and temporal
↪  analysis of clusters of particles in turbulent flow
# Script Description: This script is in charge of plotting by means of a GIF
↪  file, how a set of classified clusters evolve
# between two time instants. In order to do so, for every snapshot in Z, the
↪  cluster boundaries and skeleton points are plotted
# together, and the pairing between skeleton points is also displayed.
# Álvaro Tomás Gil - UIUC 2020

class temporalPlot:
    """This class is in charge of plotting by means of a GIF file, how a set
     ↪  of classified clusters evolve
    between two time instants. In order to do so, for every snapshot in Z,
↪   the cluster boundaries and skeleton points are plotted
    together, and the pairing between skeleton points is also displayed.
    For the inputs:
    skel: List of two elements, where each element is a 2D array of 3D
↪   positions of skeleton points for each of the time
    instants compared.
    bound: List of two elements, where each element is a 2D array of 3D
↪   positions of cluster boundary points for each of the time
    instants compared.
    labels: List of two elements, where each element is a list of cluster
↪   label for the skeleton points of each of the time
    instants compared.
    connections: 2D array of 3D points defining connections between skeleton
↪   points of the first time frame with the second one
    folder: folder in which to save resulting plots"""
    def __init__(self, skel, bound, labels, connections, folder=''):
        self.skel = skel
        self.bound = bound
        self.labels = labels
        self.folder = folder
        self.connections = connections

        self.zs = np.unique(skel[0][:, 2])

        # Combinations of marker and marker boundary colors are made in order
        ↪  to increase the possibilities of marker types
        self.skelUniqueLabels = [0, 0]
        self.skelColors = [0, 0]
        self.skelColorsB = [0, 0]
        self.boundC = [0, 0]
        self.cm = [cm.get_cmap('winter'), cm.get_cmap('autumn')]
        for i in range(2):
            self.skelUniqueLabels[i] = np.unique(self.labels[i])
            strength = np.linspace(0, 0.8, len(self.skelUniqueLabels[i]))
            np.random.shuffle(strength)
            self.skelColors[i] = [self.cm[i](each) for each in strength]
            np.random.shuffle(strength)
            self.skelColorsB[i] = [self.cm[i](each) for each in strength]
            normalized = (self.bound[i][:, 2] - np.min(self.zs)) /
            ↪  (np.ptp(self.zs))
            self.boundC[i] = self.cm[i](normalized)


    def snapPlot(self, title='Temporal Tracking of Particle Clusters'):
        """This method plots each two-dimensional domain separately, but
        ↪  joins all of them in a GIF animation which allows
        a 3D evolution of them to be visualized."""

        def update(choose):
            fig, ax = plt.subplots()
            fig.set_size_inches(18.5, 9.5)

            self.plotInstance(choose, ax)
            ax.legend()
            ax.set_title('Z = ' + '{0:03f}'.format(choose), fontsize=24)
            ax.set_xlabel('x [m]', fontsize=18)
            ax.set_ylabel('y [m]', fontsize=18)
            ax.tick_params(axis='both', which='major', labelsize=15)
            plt.axis('equal')
            ax.set_xlim(np.min(self.bound[0][:, 0]), np.max(self.bound[0][:,
            ↪  0]))
            ax.set_ylim(np.min(self.bound[0][:, 1]), np.max(self.bound[0][:,
            ↪  1]))
            fig.canvas.draw()
            image = np.frombuffer(fig.canvas.tostring_rgb(), dtype='uint8')
            image = image.reshape(fig.canvas.get_width_height()[::-1] + (3,))
            plt.close()
            return image

        kwargs_write = {'fps': 1.0, 'quantizer': 'nq'}
        imageio.mimsave(self.folder + title + '.gif', [update(i) for i in
        ↪   self.zs], fps=2)

    def plotInstance(self, choose, ax):
        if len(self.zs) >= 2:
            dz = (self.zs[1] - self.zs[0]) / 4
        else:
            dz = 0.001 * self.zs[0]

        takeC = self.connections[:, 2] == choose
        ax.scatter(self.connections[takeC, 0], self.connections[takeC, 1],
        ↪   s=5, marker='D')

        for i in range(2):
            if i == 0:
                label = 'Old Skeleton - Cluster '
                labelB = 'Old Cluster Boundary'
            else:
                label = 'New Skeleton - Cluster '
                labelB = 'New Cluster Boundary'
            takeB = self.bound[i][:, 2] == choose
            takeS = ((self.skel[i][:, 2] >= choose - dz) & (self.skel[i][:,
            ↪   2] <= choose + dz))
            self.plotLabels(ax, self.skel[i][takeS, :],
            ↪   self.labels[i][takeS], self.skelColors[i],
            ↪   self.skelColorsB[i], label, marker='P')
            ax.scatter(self.bound[i][takeB, 0], self.bound[i][takeB, 1],
            ↪   c=self.boundC[i][takeB], s=1, label=labelB)


    @staticmethod
    def plotLabels(ax, data, labels, colors, colorsB, label, marker='.',
    ↪   size=75):
        unique_labels = np.unique(labels)

        for k, col, colB in zip(unique_labels, colors, colorsB):
            class_member_mask = (labels == k)
            xy = data[class_member_mask]
            if len(xy) > 0:
                if label is not None:
                    ax.scatter(xy[:, 0], xy[:, 1],
                     ↪  c=np.reshape(np.array(col), (1, -1)),
                           edgecolors=np.reshape(np.array(colB), (1, -1)),
                            ↪  s=size, marker=marker,
                           label=label + str(k))
                else:
                    ax.scatter(xy[:, 0], xy[:, 1],
                     ↪  c=np.reshape(np.array(col), (1, -1)),
                               edgecolors=np.reshape(np.array(colB), (1,
                                ↪  -1)), s=size, marker=marker, label='')

    def sankeyPlot(self, paired):
        from pySankey import sankey
        import pandas as pd
        df = pd.DataFrame(data=paired, columns=['Old Cluster', 'New
        ↪  Cluster'])
        sankey.sankey(df['Old Cluster'], df['New Cluster'], 'Cluster
        ↪  Evolution through Movement of Skeleton Points', fontsize=15,
        ↪  aspect=2, figure_name=self.folder + 'Sankey Diagram of Cluster
        ↪  Evolution')



        # from matplotlib.sankey import Sankey
        #
        # if num == 1:
        #     transitions = [transitions]
        #
        # previous = np.ones((1, np.size(transitions, axis=1)))
        # for i,t in enumerate(transitions):
        #
        #     ranking = np.argsort(np.sum(t, axis=1))[::-1][:topClusters]
        #     for j,r in enumerate(ranking):
        #         fig = plt.figure()
        #         ax = fig.add_subplot(111, xticks=[], yticks=[])
        #         ax.set_title('Time Step ' + str(i) + ' - Cluster ' +
        ↪  str(r), fontsize=24)
        #         sankey = Sankey(ax=ax)
        #         total = sum(t[r, :])
        #         flows = []
        #         labels = []
        #         for k, f in enumerate(previous[:, j + 1]):
        #             if i == 0:
        #                 flows.append(f)
```

```
#             else:
#                 flows.append(f / total)
#             labels.append('Old Cluster ' + str(k + 1))
#
#         for k, f in enumerate(t[r, :]):
#             flows.append(- f / total)
#             if k == 0:
#                 labels.append('Unpaired')
#             else:
#                 labels.append('New Cluster ' + str(k))
#
#         sankey.add(flows=flows, orientations=[0 for _ in flows],
↪   labels=labels)
#         sankey.finish()
```

O voronoiCluster.py

```
import numpy as np
from scipy.spatial import Voronoi, ConvexHull
from sklearn.neighbors import NearestNeighbors
import matplotlib.pyplot as plt
import time
from voronoiPlot import voronoiPlot

# Development of data analysis tools for the topological and temporal
↪  analysis of clusters of particles in turbulent flow
# Script Description: This script is designed perform the 3D particle cluster
↪  analysis of a dataset of particle positions
# employing Voronoi tesselations. First, the dataset of positions is cropped
↪  to a specified set of dimensions. Then,
# the Voronoi cells corresponding to the particles are determined, and those
↪  of a volume corresponding to a cluster particle
# will be isolated. Then the connectivity of these Voronoi cells will be
↪  studied, in order to determine global cluster volumes.
# Lastly, this script allows for a topological validation of a set of
↪  skeleton points, by examining what percentage of these
# skeleton points exist in regions which this Voronoi analysis classifies as
↪  belonging to particle clusters.
# Álvaro Tomás Gil - UIUC 2020

class VoronoiCluster:
    """This is the main class of the script, in charge of performing the 3D
    ↪  particle cluster analysis of a dataset of particle positions
    employing Voronoi tesselations. First, the dataset of positions is
↪   cropped to a specified set of dimensions. Then,
    the Voronoi cells corresponding to the particles are determined, and
↪   those of a volume corresponding to a cluster particle
    will be isolated. Then the connectivity of these Voronoi cells will be
↪   studied, in order to determine global cluster volumes. For the input:
    data: 2D array of 3D positions of all particles
    filename: If data is None, VTK file from which to load the data
    ranges: Limits of the domain in the form [[xmin, xmax], [ymin, ymax],
↪   [zmin, zmax]]"""
    def __init__(self, ranges=[[0.108, 0.162], [0.008, 0.032], [0.013,
    ↪   0.028]], data=None, filename='prt_TG_ductVe8_780000.vtk'):
        print('Clustering Analysis of Turbulent Particle-laden Flow with
        ↪   Voronoi Volumes')
        self.ranges = np.array(ranges)
        self.data = data
        self.times = [0, 0]

        if data is None:
            self.loadVTK(filename)


    def loadVTK(self, filename):
        """This method is in charge of loading the VTK file in order to
        ↪   obtain an un-projected set of particle positions"""
        import vtk
        print('Extracting Dataset')
        start = time.time()
        reader = vtk.vtkPolyDataReader()
        reader.SetFileName(filename)
        reader.Update()
        polydata = reader.GetOutput()
        n = polydata.GetNumberOfPoints()
        self.data = np.array([0, 0, 0])

        for i in range(0, n, 1):
            vraw = list(polydata.GetPoint(i))
            inRange = np.all([vraw[0] > self.ranges[0,0], vraw[0] <
            ↪   self.ranges[0,1], vraw[1] > self.ranges[1,0], vraw[1] <
            ↪   self.ranges[1,1], vraw[2] > self.ranges[2,0], vraw[2] <
            ↪   self.ranges[2,1]])
            if inRange:
                self.data = np.vstack((self.data, np.array(vraw)))
                if i % 50000 == 0:
```

```
                print(' Out of the ' + str(n) + ' particles in the
                ↪  dataset, ' + str(i) + ' (' + str(round(i*100/n, 3)) +
                ↪  ' %) have been processed, and ' + str(len(self.data)
                ↪  - 1) + ' have been stored.')

        self.data = self.data[1:, :]
        rangeStr = '_x[' + str(self.ranges[0,0]) + ',' +
        ↪  str(self.ranges[0,1]) + ']_y[' + str(self.ranges[1,0]) + ',' +
        ↪  str(self.ranges[1,1]) + ']_z[' + str(self.ranges[1,0]) + ',' +
        ↪  str(self.ranges[1,1]) + '].npy'
        np.save('VoronoiData' + rangeStr, self.data)
        print('Elapsed Time: ' + str(round(time.time() - start, 3)))

    def volumePDF(self, maxVar=-1, bins=75, threshold=1):
        """This method is in charge of carrying out the Voronoi Tessellation
        ↪  of the supplied data, and obtaining a PDF
        of the resulting normalized Voronoi volumes. It also compares this
↪   PDF with the one that would result from
        a set of Poisson distributed points."""
        print('Cluster Identification Based on Voronoi Volumes')
        start = time.time()
        self.vor = Voronoi(self.data)
        self.volumes, self.nonB = self.voronoiVolumes(self.vor)
        self.nonBI = np.arange(0, len(self.vor.point_region))[self.nonB]
        self.volumes_sorted = np.sort(self.volumes)
        self.oldOrder = np.argsort(self.volumes)

        if maxVar > 0:
            means = [np.mean(self.volumes_sorted)]
            varMean = []
            topV = -1
            #Discard some very big Voronoi cells which unnecessarily alter
            ↪  the mean volume. Stop once the mean volume does
            #not vary more than maxVar with an elimination of these large
            ↪  cells. Deactivate this part with maxVar= < 0
            for i in range(250):
                volumes = self.volumes_sorted[:-(i + 1)]
                means.append(np.mean(volumes))
                varM = (means[-1] - means[-2])/means[-2]
                varMean.append(varM)
                if np.abs(varM) < maxVar and topV == -1:
                    topV = -(i + 1)
            self.oldOrder = self.oldOrder[:topV]
            self.volumes_sorted = self.volumes_sorted[:topV]

        self.V = self.volumes_sorted/np.mean(self.volumes_sorted)
        self.bins = np.logspace(np.log(np.min(self.V)),
        ↪   np.log(np.max(self.V)), bins)

        self.PDF, _ = np.histogram(self.V, bins=self.bins, density=True)
        self.bins = (self.bins[1:] + self.bins[:-1]) / 2

        self.RandomPDF = self.PoissonPDF(self.bins)
        self.intersectPDFs(threshold=threshold)
        self.assignLabels()
        self.times[0] = time.time() - start
        print('Elapsed Time: ' + str(round(time.time() - start, 3)))


    def voronoiVolumes(self, vor):
        """Given a Voronoi Object, this method is in charge of obtaining the
        ↪   volume of each Voronoi Cell, and classifying it
        as a boundary cell if one of its vertex indices is -1 or if any of
↪   its vertices is outside the domain of interest"""
        volumes = np.array([])
        data = vor.points
        limits = [[np.min(data[:, 0]), np.max(data[:, 0])], [np.min(data[:,
        ↪   1]), np.max(data[:, 1])], [np.min(data[:, 2]), np.max(data[:,
        ↪   2])]]
        nonB = [False for _ in data]
        for i, region in enumerate(vor.point_region):
            indices = vor.regions[region]
            if -1 not in indices:
                v = vor.vertices[indices]
                isWithin = self.checkVertices(v, limits)
                if isWithin:
                    volumes = np.append(volumes, ConvexHull(v).volume)
                    nonB[i] = True
        return volumes, nonB

    @staticmethod
    def checkVertices(vertices, limits):
        """Given a set of Voronoi Vertices, this simple methods checks if all
        ↪   of them are maintained within the range
        of the form [[xmin, xmax], [ymin, ymax], [zmin, zmax]] expressed in
↪   limits"""
        isWithin = True
        for i,v in enumerate(vertices):
            x = v[0]
            y = v[1]
            z = v[2]
            if x < limits[0][0] or x > limits[0][1]:
                isWithin = False
                break
            if y < limits[1][0] or y > limits[1][1]:
                isWithin = False
                break
            if z < limits[2][0] or z > limits[2][1]:
                isWithin = False
                break
        return isWithin

    @staticmethod
    def PoissonPDF(v):
        """Given a set of normalized Voronoi volumes, this method computes
        ↪  the corresponding PDF, as per Ferenc et al. 1992"""
        from scipy.special import gamma

        a = 3.24174
        b = 3.24269
        c = 1.26861
        g = gamma(a / c)
        k1 = c * b ** (a / c) / g
        pdf = k1 * np.power(v, (a - 1)) * np.exp(- b * np.power(v, c))
        return pdf

    def intersectPDFs(self, threshold=1):
        """This method determines at which normalized Voronoi volumes do the
        ↪  Random PDF and the obtained PDF intersect"""
        diff = np.abs(self.PDF - self.RandomPDF)
        half = np.argmax(self.RandomPDF)
        start = np.nonzero(self.PDF > 0.5*np.max(self.PDF))[0][0]
        end = np.nonzero(self.RandomPDF[half:] <
        ↪  0.5*np.max(self.RandomPDF))[0][0] + half

        if start == 0 and half == 0:
            self.cut1 = 0
        else:
            self.cut1 = np.argmin(diff[start:half]) + start
        self.V1 = self.bins[self.cut1] * threshold

        self.cut2 = np.argmin(diff[half:end]) + half
        self.V2 = self.bins[self.cut2]

    def assignLabels(self):
        """This obtains a list of indexes of points which can be labeled as
        ↪  cluster particles."""
        clusters = np.arange(0, len(self.V))[self.V < self.V1] #indexes
         ↪  self.V, volumes_sorted, and oldOrder
        self.clusterV = self.volumes_sorted[clusters]
        clusters = self.oldOrder[clusters] #indexes volumes
        self.clusters = self.nonBI[clusters] #indexes self.vor and self.data
        self.easyLabel = np.zeros(len(self.data))
        self.easyLabel[self.clusters] = 1
        print('Out of ' + str(len(self.data)) + ' particles, ' +
         ↪  str(len(self.clusters)) + ' (' +
         ↪  str(round(len(self.clusters)*100/len(self.data), 3)) +' %) are
         ↪  labelled as cluster particles.')

    def optimumBins(self, b0=100, b1=10000, n=100):
        """This method tracks the evolution of the first intersection between
         ↪  PDFs with the number of bins in the PDF"""
        self.intersections = []
        for i in np.linspace(b0, b1, n):
            self.volumePDF(bins=i)
            self.intersections.append(self.V1)

        plt.figure()
        plt.plot(np.linspace(b0, b1, n), self.intersections)
        plt.xlabel('Number of Bins [-]')
        plt.ylabel('Normed Voronoi Volume of Intersection [-]')
        plt.title('Evolution of Intersection Volume with Number of Bins')

    def connectClusterCells(self):
        print('Connectivities of Cluster Cells')
        start = time.time()
        self.connect = connectClusters(self.vor, self.clusters)
        self.connect.run()

        self.labels = self.connect.labels
        self.unique_labels = np.unique(self.labels)
        clusters = [c for c in self.unique_labels if c != -1]
        self.volumesC = [0 for _ in clusters]
        clustLabels = [self.labels[i] for i in self.clusters]
        for i,l in enumerate(clustLabels):
            self.volumesC[l] += self.clusterV[i]

        self.volumePlot()
        self.times[1] = time.time() - start
        print('Elapsed Time: ' + str(round(time.time() - start, 3)))
```

```
def plotVolumePDFs(self, topV=3, noSecond=True):
    """This method plots the obtained PDF and the PDF of the randomly
    ↪  distributed case together with their intersection
    points."""
    take = self.bins < topV
    fig = plt.figure()
    plt.plot(self.bins[take], self.PDF[take], label='Preferential
    ↪  Distribution')
    plt.plot(self.bins[take], self.RandomPDF[take], label='Random
    ↪  Distribution')
    plt.plot(self.V1*np.ones(50), np.linspace(0, self.PDF[self.cut1]),
    ↪  '--', label='First Intersection - V = ' + str(round(self.V1, 2)))
    if not noSecond:
        plt.plot(self.V2 * np.ones(50), np.linspace(0,
        ↪  self.PDF[self.cut2]), '--', label='Second Intersection - V =
        ↪  ' + str(round(self.V2, 2)))
    plt.xlim([0, topV])
    plt.title('Voronoi Cell Volume PDF')
    plt.xlabel('Normed Volume [-]')
    plt.ylabel('PDF [-]')
    plt.legend()

def plotClusters(self):
    """Plots all particles, sorting them into cluster or non-cluster
    ↪  particles according to the Voronoi classification"""
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    fig.set_size_inches(18.5, 9.5)
    ax.set_title('Identification of Cluster Particles with Voronoi
    ↪  Volumes', fontsize=22)
    ax.set_xlabel('x [m]', fontsize=18)
    ax.set_ylabel('y [m]', fontsize=18)
    ax.set_zlabel('z [m]', fontsize=18)

    strength = np.linspace(0, 0.8, len(self.unique_labels))
    np.random.shuffle(strength)
    colors = [plt.cm.nipy_spectral(each) for each in strength]
    np.random.shuffle(strength)
    colorsB = [plt.cm.nipy_spectral(each) for each in strength]

    for k, col, colB in zip(self.unique_labels, colors, colorsB):
        a = 1
        s = 3
        if k == -1:
            # Black used for noise.
            col = [1, 0, 0]
```

```
                a = 0.3
                s = 1

            class_member_mask = (self.labels == k)
            xy = self.data[class_member_mask]
            if len(xy) > 0:
                ax.scatter(xy[:, 0], xy[:, 1], xy[:, 2],
                 ↪  c=np.reshape(np.array(col), (1, -1)),
                           edgecolors=np.reshape(np.array(colB), (1, -1)),
                            ↪  alpha=a, s=s, label='Cluster ' + str(k))


    def plotVolumeContours(self):
        """Plots all particles, coloring them as a function of their
         ↪  associated Voronoi cell volume"""
        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3d')
        fig.set_size_inches(18.5, 9.5)
        ax.set_title('Particle Positions Colored by Voronoi Volumes',
         ↪  fontsize=22)
        ax.set_xlabel('x [m]', fontsize=18)
        ax.set_ylabel('y [m]', fontsize=18)
        ax.set_zlabel('z [m]', fontsize=18)
        pos = ax.scatter(self.data[self.nonB, 0], self.data[self.nonB, 1],
         ↪  self.data[self.nonB, 2], s=10, c=self.volumes, cmap='plasma')
        cbar = fig.colorbar(pos, ax=ax)
        cbar.ax.tick_params(labelsize=15)

    def plotVoronoiCell(self, cells):
        """Plots a single Voronoi cell, with its Voronoi vertices as well. To
         ↪  gain perspective wrt to the rest of the points,
        the limits of the plots are set according to the limits of all the
↪   point positions."""
        for i in cells:
            #i indexes volumes
            i = self.nonBI[i] #now i indexes vor.point_region

            vI = self.vor.regions[self.vor.point_region[i]]
            v = self.vor.vertices[vI, :]
            r = v

            fig = plt.figure()
            ax = fig.add_subplot(111, projection='3d')
            fig.set_size_inches(18.5, 9.5)
            ax.set_title('Voronoi Cell of Particle ' + str(i))
            ax.set_xlabel('x [m]')
            ax.set_ylabel('y [m]')
            ax.set_zlabel('z [m]')
            ax.scatter(r[:, 0], r[:, 1], r[:, 2], s=5, alpha=0.5, label='Cell
            ↪  Boundaries')
            ax.scatter(self.data[i, 0], self.data[i, 1], self.data[i, 2],
            ↪  s=25, label='Cell Center')
            ax.set_xlim3d(np.min(self.data[:, 0]), np.max(self.data[:, 0]))
            ax.set_ylim3d(np.min(self.data[:, 1]), np.max(self.data[:, 1]))
            ax.set_zlim3d(np.min(self.data[:, 2]), np.max(self.data[:, 2]))
            # limits = np.vstack((np.array([np.max(self.data[:, 0]),
            ↪  np.max(self.data[:, 1]), np.max(self.data[:, 2])]),
            ↪  np.array([np.min(self.data[:, 0]), np.min(self.data[:, 1]),
            ↪  np.min(self.data[:, 2])])))
            # ax.scatter(limits[:, 0], limits[:, 1], limits[:, 2], s=1)
            ax.legend()

    def volumePlot(self, top=10):
        """This method is simply in charge of plotting a bar plot comparing
        ↪  cluster volumes"""
        fig = plt.figure()
        fig.set_size_inches(18.5, 9.5)
        ax = fig.add_subplot(111)
        label = ['Cluster ' + str(i) for i in range(1, len(self.volumesC) +
        ↪  1)]

        volumesC = np.sort(self.volumesC)[::-1][:top]
        sortI = np.argsort(self.volumesC)[::-1][:top]
        label = [label[i] for i in sortI]

        cmap = plt.get_cmap('plasma')
        c = cmap(volumesC)

        ax.bar(range(top), volumesC, tick_label=label, width=0.5, color=c)
        ax.tick_params(labelsize=18)
        plt.ylabel('Volume [m^3]', fontsize=18)
        plt.title('Volume per Cluster', fontsize=22)
        plt.savefig('Voronoi Volumes per Cluster')

    def timePiePlot(self, pctM=0.04):
        """This method simply generates a plot of the time consumption
        ↪  associated to each step of the analysis."""
        names = ['Voronoi Tesselation', 'Connectivity of Cluster Cells']
        dict = {}
        for i,j in zip(names, self.times):
            dict[i] = j
        total = sum(dict.values())
        title = 'Time Consumption per Step of Voronoi Analysis  - Total [s]=
        ↪  ' + str(round(total, 3))
        labels = []
        values = []
        for v in dict.keys():
            if dict[v] / total > pctM:
                labels.append(v + ' - ' + str(round(dict[v], 3)) + ' (' +
                ↪  str(round(dict[v] * 100 / total, 2)) + ' %)')
            else:
                labels.append(v)
            values.append(dict[v] / total)
        labdis = 1.07
        cmap = plt.get_cmap("plasma")
        c = np.arange(len(dict.keys())) / len(dict.keys())
        colors = cmap(c)

        fig = plt.figure()
        fig.set_size_inches(12, 7)
        plt.title(title, fontsize=22)
        plt.pie(dict.values(), labels=labels, shadow=True, startangle=0,
        ↪  labeldistance=labdis, colors=colors)
        plt.axis('equal')  # Equal aspect ratio ensures that pie is drawn as
        ↪  a circle.


class connectClusters:
    """This class connects Voronoi cells labeled as cluster cells into larger
    ↪  clusters, by applying a cluster label to
    each Voronoi cell. For the input:
    vor: Voronoi object from scipy.spatial of a dataset of particle positions
    clusters: List of indices indexing the dataset employed in vor,
↪   referencing non-boundary Voronoi cells which are cluster cells"""
    def __init__(self, vor, clusters):
        self.vor = vor
        self.clusters = clusters.astype(int)

        self.N = len(clusters)
        self.isCluster = [i in self.clusters for i in
        ↪  range(len(self.vor.point_region))]
        self.labels = [-1 for _ in self.vor.point_region]
        self.maxLabel = 0
        self.pairs = vor.ridge_points
        self.taken = []
        self.it = 0
    def run(self):
        """Main method of the class"""
        for i,p in enumerate(self.pairs):
            self.forPointPair(i)
            if i % 100000 == 0:
                print('Percentage Processed: ' + str(round(i * 100 /
                 ↪  len(self.pairs), 3)) + '. Existing Cluster Labels: ',
                 ↪  len(np.unique(self.labels)))

    def forPointPair(self, i):
        """For a Voronoi ridge specified by i, this method processes the
         ↪  adjacent Voronoi cell centers, assigning
        the corresponding cluster label to each of them."""
        areCluster = [self.isCluster[j] for j in self.pairs[i]]

        if sum(areCluster) > 1:
            #If at least two neighboring cells are cluster cells, four
             ↪  possible cases exist: 1. none of them have been previously
            #labeled and thus a new cluster label has to be defined, 2. all
             ↪  have been labeled with the same cluster label
            #and as a result nothing is to be done, 3. only few of them has
             ↪  been labeled with a cluster label which is
            #then propagated to the other cells, 4. or several have been
             ↪  assigned different cluster labels, and thus the older
            #cluster label has to be propagated.

            labels = [self.labels[j] for j in self.pairs[i]]
            already = [j != -1 for j in labels]
            if sum(already) == 0: #None of the cell centers have been
             ↪  assigned a cluster label
                for j,p in enumerate(self.pairs[i]):
                    if areCluster[j]:
                        self.labels[p] = self.maxLabel
                self.maxLabel += 1
            else: #At least one of the cell centers has been assigned a
             ↪  cluster label
                contesting = [j for j in labels if j != -1]
                toAssign = min(contesting)
                for j,p in enumerate(self.pairs[i]):
                    if areCluster[j]:
                        if labels[j] == -1:
                            self.labels[p] = toAssign
                        elif labels[j] != toAssign:
                            self.propagateLabel(toAssign, labels[j])
                self.maxLabel = np.max(self.labels) + 1
    def propagateLabel(self, l1, l2):
        """This method solves a conflict of labels by propagating the older
        ↪  (lower) label to the Voronoi cells labeled with the
        newer label"""

        if l1 != l2:
            winner = min(l1, l2)
            loser = max(l1, l2)
            loserN = 0
            superiorN = 0
            for i,l in enumerate(self.labels):
                if l == loser:
                    loserN += 1
                    self.labels[i] = winner
                if l > loser:
                    superiorN += 1
                    self.labels[i] = l - 1

            # print('Loser Label is ' + str(loser) + ' . With ' + str(loserN)
            ↪  + ' associated cells. Winner label is ' + str(winner))

class  VoronoiValidation:
    """This class carries out the validation of a set of skeleton points to
    ↪  which a cluster label has been assigned,
    by comparing these labels to the ones which result of performing a 3D
↪   clustering analysis with Voronoi. For each
    skeleton point, this class extracts its closest Voronoi cell, as well as
↪   the cluster label assigned to such cell. Then,
    for each skeleton cluster label, one can examine what percentage of its
↪   skeleton points has been misclassified.
    For the inputs:
    data: 2D array of 3D positions of Voronoi cell centers, or essentially
↪   particle positions
    vorLabels: List of same length as data, assigning a Voronoi cluster label
↪   to each cell center
    skel: 2D array of 3D positions of skeleton points
    skelLabels: List of same length as skel, assigning a cluster label to
↪   each skeleton point
    expelExtreme: boolean determining whether to expel skeleton particles
↪   from the upper and lower levels of Z from the analysis"""
    def __init__(self, data, vorLabels, skel, skelLabels,
    ↪  expelExtremes=True):
        self.data = data
        self.vorLabels = [v + 1 for v in vorLabels]
        self.skel = skel
        self.skelLabels = [s - 1 for s in skelLabels]
        if expelExtremes:
            maxZ = np.max(self.skel[:, 2])
            minZ = np.min(self.skel[:, 2])
            expel = [i for i in range(len(self.skel)) if self.skel[i, 2] ==
            ↪  maxZ or self.skel[i, 2] == minZ]
            self.skel = np.delete(self.skel, expel, axis=0)
            self.skelLabels = np.delete(self.skelLabels, expel, axis=0)

        self.nbrs = NearestNeighbors(n_neighbors=1).fit(self.data)
        self.uniqueVor = np.unique(self.vorLabels)
        self.uniqueSkel = np.unique(self.skelLabels)
        self.memberships = np.zeros((len(self.uniqueSkel),
        ↪  len(self.uniqueVor)))
        self.isCorrect = [1 for _ in self.skel]

    def run(self):
        """Main method of the class, in charge of examining skeleton cluster
        ↪  label and presenting results"""
        for l in self.uniqueSkel:
            mask = np.arange(len(self.skel))[self.skelLabels == l]
            counts = self.findNearest(mask)
            self.memberships[l] = counts

        #self.memberships is an array of as many rows as skeleton labels and
        ↪  as many columns as Voronoi cluster labels,
        #where the i-th row shows for all skeleton points of cluster label i,
        ↪  how many belong to each of the Voronoi
        #cluster labels. More precisely, the j-th column of the i-th row of
        ↪  this array shows how many skeleton points
        #of cluster label i have a closest Voronoi cell center of label j.

        print('Out of ' + str(len(self.skel)) + ' skeleton points, ' +
        ↪  str(sum(self.memberships[:, 0])) + ' (' +
        ↪  str(round(sum(self.memberships[:, 0]) * 100/len(self.skel), 3)) +
        ↪  ' %) appear in areas classified as void areas by Voronoi')

        for l in self.uniqueSkel:
            members = sum(self.skelLabels == l)
            topVor = np.argsort(self.memberships[l])[::-1][:5] - 1
            counts = np.sort(self.memberships[l])[::-1][:5]
            print('For the ' + str(members) + ' skeleton points with label '
            ↪  + str(l) + ': ')
            for i in range(5):
                if counts[i] > 0:
                    if topVor[i] == -1:
```

```
                    add = ' ' + str(counts[i]) + ' ( ' +
                     ↪  str(round(counts[i] * 100 / members, 3)) + ' %)
                     ↪  are not associated with a Voronoi cluster cell'
                else:
                    add = ' ' + str(counts[i]) + ' ( ' +
                     ↪  str(round(counts[i] * 100/ members, 3)) + ' %)
                     ↪  belong to the Voronoi Cluster with label ' +
                     ↪  str(topVor[i])
                print(add)

        self.plotResults()

    def findNearest(self, i):
        """For a list i of indexes of skeleton point positions, this method
         ↪  examines the closest Voronoi cel center to each
        skeleton point, and based on this counts how many of the skeleton
↪   points belong to each Voronoi label. Note that
        memberships is a vector where its i-th element shows how many of the
↪   skeleton positions have a closest Voronoi
        cell of label i."""
        skel = self.skel[i, :]
        closest = self.nbrs.kneighbors(skel, return_distance=False)
        memberships = np.zeros(len(self.uniqueVor))
        for j, c in enumerate(closest):
            c = c[0]
            nearLabel = self.vorLabels[c]
            memberships[nearLabel] += 1
            if nearLabel == 0:
                self.isCorrect[i[j]] = 0
        return memberships

    def plotResults(self):
        """This method plots the skeleton particles labeled according to
         ↪  whether their closest Voronoi cell is classified
        as a cluster cell or not."""

        clusters = self.data[[i for i in range(len(self.data)) if
         ↪  self.vorLabels[i] != 0], :]
        vorLabels = [self.vorLabels[i] for i in range(len(self.data)) if
         ↪  self.vorLabels[i] != 0]

        self.plot = voronoiPlot(clusters, self.skel, self.skelLabels,
         ↪  self.isCorrect, vorLabels)
        self.plot.snapPlot()

# data = np.load('VoronoiValidation_Data.npy')
# tr = VoronoiCluster(data)
# tr.volumePDF()
# np.save('VoronoiValidation_Clusters.npy', tr.clusters)
# np.save('VoronoiValidation_ClusterVolumes.npy', tr.clusterV)
# tr.plotVolumePDFs()
#
# tr.connectClusterCells()
# tr.plotClusters()
# tr.timePiePlot()
# np.save('VoronoiValidation_ClusterLabels.npy', tr.connect.labels)
# np.save('VoronoiValidation_VolumeperCluster.npy', tr.volumesC)
#
# skel = np.load('skeletonize.npy')
# skelLabels = np.load('SKlabels.npy')
# vv = VoronoiValidation(data, tr.connect.labels, skel, skelLabels)
# vv.run()
```

P voronoiPlot.py

```
import numpy as np
from matplotlib import cm
import matplotlib.pyplot as plt
import imageio


# Development of data analysis tools for the topological and temporal
↪  analysis of clusters of particles in turbulent flow
# Script Description: This script is in charge of plotting by means of a GIF
↪  file, the constellation of skeleton points
# interior to each two-dimensional cluster along with the Voronoi cell
↪  centers classified as cluster centers. On the one
# hand, skeleton positions are labeled according to their three-dimensional
↪  cluster, as well as according to whether their
# closest Voronoi cell is that of a cluster cell or not. On the other hand,
↪  Voronoi cell centers are labeled according to
# the cluster labeled assigned to them.
# Álvaro Tomás Gil - UIUC 2020

class voronoiPlot:
    """This class is in charge of plotting by means of a GIF file, the
     ↪  constellation of skeleton points
    interior to each two-dimensional cluster along with the Voronoi cell
↪  centers classified as cluster centers. On the one
```

```
    hand, skeleton positions are labeled according to their three-dimensional
↪   cluster, as well as according to whether their
    closest Voronoi cell is that of a cluster cell or not. On the other hand,
↪   Voronoi cell centers are labeled according to
    the cluster labeled assigned to them.
    For the inputs:
    data: 2D array of 3D positions corresponding to Voronoi cluster cell
↪   centers
    skel: 2D array of 3D positions of skeleton points.
    skelLabels: list of the same length as skel, assigning a cluster label to
↪   each skeleton point
    skelProx: list of the same length as skel, assigning a 1 to skeleton
↪   points whose closest Voronoi cell center is that
    of a cluster cell
    vorLabels: list of the same length as data, assigning a cluster label to
↪   each Voronoi cell center
    folder: folder in which to save resulting plots"""
    def __init__(self, data, skel, skelLabels, skelProx, vorLabels,
    ↪   folder=''):
        self.skel = skel
        self.skelLabels = skelLabels
        self.skelProx = skelProx
        self.vorLabels = vorLabels
        self.folder = folder

        self.zs = np.unique(skel[:, 2])

        #Project Voronoi cell centers onto the finite number of levels of Z
        self.data = np.array([0, 0, 0])
        self.vorLabels = []
        dz = abs(self.zs[1] - self.zs[0]) / 2

        for i, z in enumerate(self.zs):
            take = ((data[:, 2] < z + dz) & (data[:, 2] > z - dz))
            this = data[take, :2]
            this = np.hstack((this, z * np.ones((sum(take), 1))))
            self.data = np.vstack((self.data, this))

            newLabels = [vorLabels[i] for i in range(len(data)) if take[i]]
            self.vorLabels = self.vorLabels + newLabels
        self.data = self.data[1:, :]

        # Combinations of marker and marker boundary colors are made in order
        ↪   to increase the possibilities of marker types
        self.skelUniqueLabels = np.unique(self.skelLabels)
        strength = np.linspace(0, 0.8, len(self.skelUniqueLabels))
        np.random.shuffle(strength)
        self.skelColors = [plt.cm.autumn(each) for each in strength]
        np.random.shuffle(strength)
        self.skelColorsB = [plt.cm.autumn(each) for each in strength]

        #Marker colors for Voronoi cell centers
        self.vorUniqueLabels = np.unique(self.vorLabels)
        strength = np.linspace(0, 1, len(self.vorUniqueLabels))
        np.random.shuffle(strength)
        self.vorColors = [plt.cm.winter(each) for each in strength]

    def snapPlot(self, title='Topological Coincidence Between Skeleton Points
    ↪  and Voronoi Cell Centers'):
        """This method plots each two-dimensional domain separately, but
        ↪  joins all of them in a GIF animation which allows
        a 3D evolution of them to be visualized."""

        def update(choose):
            fig, ax = plt.subplots()
            fig.set_size_inches(18.5, 9.5)

            self.plotInstance(choose, ax)
            ax.legend()
            ax.set_title('Z = ' + '{0:03f}'.format(choose), fontsize=24)
            ax.set_xlabel('x [m]', fontsize=18)
            ax.set_ylabel('y [m]', fontsize=18)
            ax.tick_params(axis='both', which='major', labelsize=15)
            plt.axis('equal')
            ax.set_xlim(np.min(self.data[:, 0]), np.max(self.data[:, 0]))
            ax.set_ylim(np.min(self.data[:, 1]), np.max(self.data[:, 1]))

            fig.canvas.draw()
            image = np.frombuffer(fig.canvas.tostring_rgb(), dtype='uint8')
            image = image.reshape(fig.canvas.get_width_height()[::-1] + (3,))
            plt.close()
            return image

        kwargs_write = {'fps': 1.0, 'quantizer': 'nq'}
        imageio.mimsave(self.folder + title + '.gif', [update(i) for i in
        ↪  self.zs], fps=2)

    def plotInstance(self, choose, ax):
        if len(self.zs) >= 2:
            dz = (self.zs[1] - self.zs[0]) / 4
        else:
            dz = 0.001 * self.zs[0]
        take = self.data[:, 2] == choose
        takeS = ((self.skel[:, 2] >= choose - dz) & (self.skel[:, 2] <=
         ↪  choose + dz))

        skelCluster = [takeS[i] and self.skelProx[i] == 1 for i in
         ↪  range(len(takeS))]
        skelVoid = [takeS[i] and self.skelProx[i] == 0 for i in
         ↪  range(len(takeS))]
        data = self.data[take, :]
        skelClusterLabels = [self.skelLabels[i] for i in
         ↪  range(len(self.skelLabels)) if skelCluster[i]]
        skelVoidLabels = [self.skelLabels[i] for i in
         ↪  range(len(self.skelLabels)) if skelVoid[i]]
        vorLabels = [self.vorLabels[i] for i in range(len(self.vorLabels)) if
         ↪  take[i]]

        self.plotLabels(ax, data, vorLabels, self.vorColors, self.vorColors,
         ↪  None, size=15)
        self.plotLabels(ax, self.skel[skelCluster, :], skelClusterLabels,
         ↪  self.skelColors, self.skelColorsB, 'Coinciding Skeleton Points of
         ↪  Cluster ', marker='^')
        self.plotLabels(ax, self.skel[skelVoid, :], skelVoidLabels,
         ↪  self.skelColors, self.skelColorsB, 'Non-Coinciding Skeleton
         ↪  Points of Cluster ', marker='v')

    @staticmethod
    def plotLabels(ax, data, labels, colors, colorsB, label, marker='.',
     ↪  size=100):
        unique_labels = np.unique(labels)

        for k, col, colB in zip(unique_labels, colors, colorsB):
            class_member_mask = (labels == k)
            xy = data[class_member_mask]
            if len(xy) > 0:
                if label is not None:
                    ax.scatter(xy[:, 0], xy[:, 1],
                     ↪  c=np.reshape(np.array(col), (1, -1)),
                           edgecolors=np.reshape(np.array(colB), (1, -1)),
                            ↪  s=size, marker=marker,
                           label=label + str(k))
                else:
                    ax.scatter(xy[:, 0], xy[:, 1],
                     ↪  c=np.reshape(np.array(col), (1, -1)),
                               edgecolors=np.reshape(np.array(colB), (1,
                                ↪  -1)), s=size, marker=marker, label='')
```