IMPERIAL COLLEGE LONDON

MENG INDIVIDUAL PROJECT

Visualising the Evolution of Aerodynamic Flows over Time using Trees

Author:
Mihai POPA

Supervisor:
Prof. Paul KELLY

Department of Computing

June 19, 2017

Abstract

Exploring the evolution of features in large-scale time-varying fields is an important problem in a wide range of science and engineering areas. One example is the area of Computational Fluid Dynamics (CFD), where the efficient analysis of time-dependent data produced by CFD solvers is essential to the understanding of fluid flows and their applicability to engineering problems. Contour trees represent a data structure often used to explore discrete fields, while ignoring the temporal dimension of the data. They are built to capture the nesting relationship between the features of one field, and have been often used as a feature extraction strategy.

In this project we present an algorithm for tracking features in time-varying datasets, based on spatial overlap and on the structures of the contour trees corresponding to the data. We integrated the developed algorithm in an interactive graphical interface, where the temporal evolution of field features can be visualized. The efficiency of the tracking tool was then demonstrated on datasets belonging to CFD simulations. Finally, we analyzed the feasibility of running the tracking algorithm in-situ, by integrating it in PyFR, a CFD solver developed at Imperial College London.

Acknowledgements

Firstly, I would like to thank to Prof. Paul Kelly for his continuous guidance and inspiration throughout this project. I am also extremely grateful to Dr. Peter Vincent for his enthusiasm in steering the project in the right direction. I would like to thank to Yoshiaki Abe for his invaluable feedback and for helping me understand many fluid dynamics concepts.

Finally, I would like to thank Oana and my parents, Dragos and Adriana, for their unconditional love and support.

Contents

Abstract **ii**

Acknowledgements **iii**

1 Introduction **1**
- 1.1 Motivation 1
- 1.2 Contributions 2
- 1.3 Report outline 3

2 Background and Related Work **4**
- 2.1 Computational Fluid Dynamics 4
- 2.2 Turbulence and vortex detection 4
- 2.3 PyFR 6
- 2.4 Data analysis 7
  - 2.4.1 Paraview 7
  - 2.4.2 The Contour Spectrum 9
- 2.5 Contour Trees 10
  - 2.5.1 Definitions and properties 10
  - 2.5.2 Construction 13
    - Building an object tree 14
- 2.6 Using Contour Trees to understand flows 15
- 2.7 Union-find data structure 17
- 2.8 Extracting isosurfaces 18
- 2.9 Tracking features 19
  - 2.9.1 Region-based tracking 20
  - 2.9.2 Tracking as a global optimization problem 23
  - 2.9.3 Based on critical points 24
  - 2.9.4 Based on contour trees 24
- 2.10 Existing work 26
- 2.11 Summary 27

3 Feature tracking **29**
- 3.1 Computing contour correspondences 30
  - 3.1.1 Contour correspondence definition 31
  - 3.1.2 The algorithm 32
    - Building a labeled object tree 33
    - Building a join tree reflecting overlaps 35
    - Labeling a contour tree from a join tree 36
    - Intersecting two labeled contour trees 37
  - 3.1.3 Implementation 38

3.2 Computing arc correspondences 41
3.3 Summary 43

4 Tool implementation 44
4.1 Functionality overview 44
4.2 Data manipulation pipeline 44
4.2.1 Preprocessing the data 45
4.2.2 Building the contour trees for feature extraction 46
4.2.3 Detecting features of interest 47
Filtering spheres 48
4.3 The graphical interface 48
4.4 Summary 49

5 Evaluation 50
5.1 Tracking features in artificial fields 50
5.1.1 Generating small scale meshes with features 50
5.1.2 Tracking moving spheres 52
5.1.3 Spheres that join and split 54
5.1.4 Identifying and tracking a banana 58
5.1.5 Tracking a banana that rotates 58
5.1.6 Plotting contours with different isovalues 62
5.1.7 Larger example with bananas and spheres 62
5.2 Tracking features in fields from aeronautical simulations . . . 66
5.2.1 The fully developed stage of the Taylor Green Vortex . 66
5.2.2 The developing stage of the Taylor Green Vortex 74
5.2.3 Selected features in the fully-developed stage of the turbulent plane Couette flow 84
5.2.4 Selected features in the fully-developed stage of the Taylor Green Vortex 89
5.3 Feature tracking metrics 95
5.4 Performance analysis 97
5.5 Comparison to other feature tracking methods 102
5.6 Tool evaluation 102
5.7 Summary 103

6 Analyzing streaks in the turbulent plane Couette flow 104
6.1 Motivation 104
6.2 Filtering vortices 106
6.2.1 Performing an orthogonal distance regression 107
6.2.2 Choosing the thresholds 108
6.3 Results 108
6.4 Summary 110

7 Conclusions 116
7.1 Reflection 116
7.2 Deliverables 116
7.3 Future Work 117
7.3.1 In-situ feature tracking 117

Scalability and hardware 117
Tracking features in distributed memory 118
Making the tracking information valuable 121
7.3.2 Dealing with data periodicity 121
7.3.3 Paraview plug-in 122
7.3.4 Designing a domain specific language for feature detection 122

A Fluid flows used **123**
A.1 Taylor Green Vortex 123
A.2 Turbulent plane Couette flow 124

Bibliography **125**

Chapter 1

Introduction

1.1 Motivation

Computational fluid dynamics (CFD) is the field aiming to understand flows of fluids and gases, by using computers in order to simulate and analyse them. The recent increase in interest for this field is largely motivated by its wide range of direct applications in industry. As an example, it supports research concerned with the analysis of air flows around airplanes and cars, targeting fuel efficiency, noise reduction, safety, or performance.

Naturally, CFD has emerged as an important research field, and multiple CFD solvers - software tools that carry out the simulations - were developed. One example is PyFR [47], a solver developed in the Vincent Lab at the Department of Aeronautics of Imperial College London. PyFR is an open-source Python framework, designed to solve the range of systems governing flows on mixed unstructured grids containing complex objects, corresponding to the needs existing in industry.

One of the challenges related to CFD is making the simulations indeed valuable. As raw data, they represent multidimensional scalar or vector fields: as an example, PyFR's output are typically 3D meshes, where each vertex has associated a velocity, a density and a pressure. For humans this kind of data is particularly difficult to visualise and understand.

Currently, visualising the data is often done by displaying cutting planes or 3D isosurfaces (areas with constant scalar value) that highlight features of the data. As an example, isosurfacing by the Q-criteria scalar is often used to observe vortical structures. Useful properties of the data can hence be discovered, but this approach comes with multiple issues:

- How can the researcher find out the isovalues (the constant values defining the isosurface) of interest? Without a prior knowledge about the domain, it can only be done by a trial and error approach. This proves to be inefficient, as rendering the desired isosurfaces can be time consuming, especially on large datasets.

- Although cutting planes and isosurfaces are able to highlight global properties of the data, they may miss certain local information valuable for the researcher.

Aiming to address these problems, Contour Trees have been used to represent the global changes in the topology of scalar fields, by capturing the nesting relationship of its isosurfaces. In this way, the Contour Tree graph represents an indexing method into the field, and can be used to identify the isovalues corresponding to interesting features of the data.

However, CFD simulations usually consist of multiple snapshots, corresponding to subsequent time steps. This adds an extra dimension, the time, to the flow understanding problem, and raises a new question: how can the researcher understand how the flow evolves over time?

A commonly used solution is to visualise global isosurfaces of the data at a fixed value across multiple time snapshots. The isovalue of interest can be chosen by trial and error, or by using contour trees. The isosurfaces will contain many features of interest, and it will be the responsibility of the researcher to spot how they evolve between snapshots. However, this approach lacks automation, and quickly becomes unfeasible for simulations containing many time steps or large fields. In this project we aim to address this problem, by developing a visualisation tool able to track features of the flow, as they evolve temporally. Exploring the evolution of features in large-scale time-varying fields is an important problem in a wide range of science and engineering areas, including CFD.

The analysis methods described so far do not assume any domain specific knowledge, and the same approaches would be used, for example, to analyse both the blood flow in vessels as well as a terrain map, as long as they are represented as scalar fields. Depending on the application, however, researchers are more interested in observing the evolution of certain specific features of the field. These features can be characterized by the means of their 3D shape or their field values. We include a proof of concept on how our tool can be extended to detect specific features in a turbulent plane Couette flow, and demonstrate the tool's ability to track these towards the later stages of the flow.

1.2 Contributions

The main goal of this project was to implement a tool supporting the understanding of time-varying scalar fields. In order to achieve this, the tool is able to detect the field features, and track them in the temporal direction. A summary of the contributions of this project is the following:

- We propose a new algorithm for tracking features in scalar fields, based on computing correspondences between contour tree arcs. This correspondence information is then used to track temporally the features defined by the arcs of the contour tree. The approach we describe is based on an existing algorithm for computing correspondences between isosurfaces [39]. We also propose an improvement for this algorithm, which enables the application of contour tree simplifications prior to the computation of the isosurface-level correspondences.

- We developed a tool that uses this tracking algorithm in order to visually observe field features as they evolve over time. The contour tree corresponding to the field is also presented, and it can be used to interactively select the tracked features.

- The tool is extended with automatic feature detection capabilities. We demonstrate how the tool is able to automatically detect sphere-like field features, as well as a specific class of vortex structures detailed at the next point.
- We demonstrate how the tool can be used to support the understanding of a specific fluid dynamics environment. To achieve this, we implemented a way to automatically identify streamwise vortex structures belonging to low-velocity streaks in a turbulent plane Couette flow, and hence showed how the Q-value analysis can be combined with other field information, such as velocity.
- We evaluate the capability of running the feature tracking algorithm at the time the simulation is performed. We highlight the relevance of this in the context of the I/O bottleneck limiting the ability to store the simulations produced by CFD solvers.

1.3 Report outline

This document is structured into 7 chapters, which detail the contributions stated above.

Chapter 2 introduces the reader to the field of computational fluid dynamics and explains how turbulence leads to the formation of vortex features in flows. We proceed by describing how scalar fields are currently visualized, and introduce the concept of Contour Tree. We then focus on describing alternative feature tracking algorithms existing in the literature.

Chapter 3 presents the feature tracking algorithm developed as part of this project, and highlights the research contributions produced by the current work. In Chapter 4, we present the capabilities of the tool developed, and focus on several implementation details. The effectiveness of the work presented in these chapters is evaluated in Chapter 5.

In Chapter 6, we produce a case study on how our tool can be used to improve over the existing literature about understanding a certain phenomena existing in wall-bounded turbulent flows.

Finally, in Chapter 7, we draw the project's conclusions and present a number of future ideas, particularly focusing on the feasibility of conducting the feature tracking analysis in-situ (at the time the simulation is performed).

Chapter 2

Background and Related Work

This chapter aims to describe the context of the current project, and to offer the knowledge background required for understanding the work and the motivation of the next chapters.

We start by giving details about relevant fluid dynamics aspects, and then proceed by introducing the concept of Contour Tree. Lastly, we describe some of the algorithms for tracking features in scalar fields, which currently exist in the literature.

2.1 Computational Fluid Dynamics

Although the huge increase of enthusiasm and research interest about CFD happened in the last fifteen years, the first contributions supporting this field are much older. The Navier - Strokes equations, representing the generalisation of Euler's work on fluid dynamics, have been published in the 19th century, and still stand as the foundation of many CFD solvers used nowadays.

The mathematical equations governing the fluid dynamics were solved using automated computing for the first time in the 1960s, as a result of the emergence of the first supercomputers. Since then, the field has continuously developed due to the progress in computer performance and the increased availability of performant hardware. However, the recent remarkable development of the field was primarily driven by the industry needs, due to its major benefits in industries like aeronautics, automotive, medical, defense or petrochemical.

A common example from aeronautics is performing CFD simulations in order to replace experiments requiring costly wind tunnels; in the petrochemical industry, CFD simulations are often used to visualise the transfer of heat or mass between environments. For engineers working in all these industries, running simulations rather than actually performing experiments is time efficient, and leads to significant cost reductions.

2.2 Turbulence and vortex detection

In fluid dynamics, turbulence occurs when a flow leaves the laminar state: it is no longer ordered and stable, but becomes irregular and chaotic. In fact, almost all fluid flow that we encounter around us is turbulent: from the airflow around cars and buildings to the flows during engine combustion.

An attempt to predict the occurrence or the lack of turbulence is represented by the Reynolds number:

$$Re = \frac{\rho U L}{\mu}$$

where ρ is the density of the fluid, U is its velocity, L is a characteristic linear dimension, and μ is the fluid viscosity.

Turbulence is associated with high values of the Reynolds number: it occurs when the value of Re exceeds a certain scalar threshold, called the *critical Reynolds number*. The value of the threshold depends on the flow; for example, in the plane Couette flow it is 2400. Analysing the formula, we can see that the occurrence of turbulence in fluid flows is favored by velocity: a fast flow increases the chance of turbulence. On the other hand, higher viscosity delays the occurrence of turbulence in the fluid.

A result of turbulence is the existence of vortices. Interestingly, although vortices seem to represent a very intuitive concept in nature, there is no universally accepted definition for it [13]. They are often described as a "rapidly spinning, circular or spiral flow of fluid around a central axis". This project is concerned with analysing vortices and vortex structures in flows produced by CFD solvers, so it is useful to present some of the common ways of measuring them:

- **Vorticity**, describing the local spinning motion of a continuum near a fixed point. It is expressed by the formula $\vec{\omega} = \nabla \times \vec{u}$, where ∇ is the del operator and $\vec{u}$ is the flow velocity vector. The presence of vortices is associated to high vorticity values, but in fact this is not always true: vorticity may also be high in parallel shear flows where no vortices are present [13]

- The **Q-criterion**, a local measure of the excess of rotation rate relative to the strain rate [11], able to distinguish shear-like flows. It makes use of the following decomposition of the velocity gradient tensor $\nabla\vec{v}$, in the sum of a symmetric and an anti-symmetric matrix [13]:

  $$\nabla\vec{v} = S + \Omega$$

  where $S = \frac{1}{2}[\nabla\vec{v} + \nabla\vec{v}^T]$ is the rate-of-strain tensor, and $\Omega = \frac{1}{2}[\nabla\vec{v} - \nabla\vec{v}^T]$ is the vorticity tensor.
  The Q-criterion is then expressed as:

  $$Q = \frac{1}{2}[|\Omega|^2 - |S|^2]$$

  Positive values of Q indicate the presence of vortices, and higher values indicate highly expressed vorticity, often in the core of a vortex

- The **$\boldsymbol{\lambda_2}$-criterion**, which identifies vortices to be local points with:

  $$\lambda_2(S^2 + \Omega^2) < 0$$

  where $\lambda_2(A)$ denotes the intermediate eigenvalue of a symmetric tensor A [11]. Although accurate, the λ_2-criterion is not as used as the Q-criterion in practice, because of its higher computational cost - computing the eigenvalues of a matrix is expensive.

2.3 PyFR

The current project aims to develop a new way of visualising PyFR simulations along multiple time steps, and to study the feasibility of integrating this at the time when simulations are performed. In this section, we present an overview of how data is generated by PyFR and how this can be analysed using Paraview [2].

At its core, PyFR needs to solve a number of partial differential equations (PDEs) governing fluid flows: the Euler equations for inviscid compressible flow and the Navier - Strokes equations for viscous compressible flow. In order to achieve this, a mesh structure is constructed by splitting the domain in small regions with simple geometry, such as triangles or quadrilaterals in 2D, and tetrahedra or pyramids in 3D.

In order to solve the equations describing the system, PyFR uses an approach based on Flux Reconstruction, idea first introduced in [15], which will only be presented briefly here due to its complexity. Every mesh element is translated to a standard element in a different space, where the solutions of the equations are computed. The equation describing the flow system is the following one:

$$\frac{\partial u_\alpha}{\partial t} + \nabla \cdot f_\alpha = 0$$

where u_α is a conserved quantity, and f_α is the flux of this conserved quantity.

Using a time stepping scheme such as RK4, one can simulate the evolution of the flux iteratively. For a particular time step, the output is represented by polynomials for each cell of the mesh, that describe the scalar or vectorial values of the grid in an accurate and compressed manner. These polynomials are stored in .pyfrs files, while the file format for the mesh description is .pyfrm. Table 2.1 presents the key functionality of PyFR, according to [47].

Although using polynomials to describe values in cells is a good representation for accuracy and compactness, the data is difficult to be analysed in this format. This is why PyFR has an export function, which converts .pyfrm and .pyfrs files to standard unstructured meshes. These have values associated to each vertex, while the values within cells can be linearly interpolated; this means that they are typically less precise than the initial format. One format following this representation is .vtu, which was also used for the input data of the current project.

TABLE 2.1: Key functionality of PyFR

Dimensions	2D, 3D
Elements	Triangles, Quadrilaterals, Hexahedra
Spatial orders	Arbitrary
Time steppers	Euler, RK4, DOPRI5
Precisions	Single, Double
Platforms	CPUs via C/OpenMP, NVIDIA GPUs via CUDA
Communication	MPI
Governing systems	Euler, Compressible Navier–Stokes

2.4 Data analysis

2.4.1 Paraview

Paraview [2] is a data analysis and visualization tool often used in understanding the flows produced by CFD solvers. It was also frequently used by us during the project work, in order to assess the correctitude of our algorithms and of our developed tool.

The .vtu files produced by PyFR are loaded into Paraview. These unstructured meshes correspond to fields where each point has typically associated three properties: a vectorial velocity, a scalar pressure and a scalar density. These are instantaneous measures of the properties at the time step the field snapshot corresponds to. It can be noticed that none of the output properties are direct measures used for detecting vortices. However, vorticity measures can be computed using this information. For example, we are able to use the gradients of the velocity property in order to compute the instantaneous Q-criterion. This can be then used in order to visualise interesting vortical features of the data.

For a snapshot of the Taylor Green Vortex simulation, a naive surface Q-criterion visualisation of the resulting flow is shown in figure 2.1, clearly unusable for any kind of analysis. On the other hand, figure 2.2 shows the isosurface for a fixed isovalue of 0.3, generated using the built-in contour filter. We can see how many vortex structures are now easily identified. This motivates the usage of isosurfaces in the process of understanding CFD data and 3D scalar or vectorial fields in general.

We have seen a basic data analysis workflow used to understand one snapshot of a fluid flow. This can be useful, but comes with a number of issues that make it far from ideal, especially when multiple time steps are analysed:

- There is no way to extract local features, as only global isosurfaces can be generated. This can be slow when the analysed field is large.
- Understanding how a particular feature evolved between snapshots is done by eye.

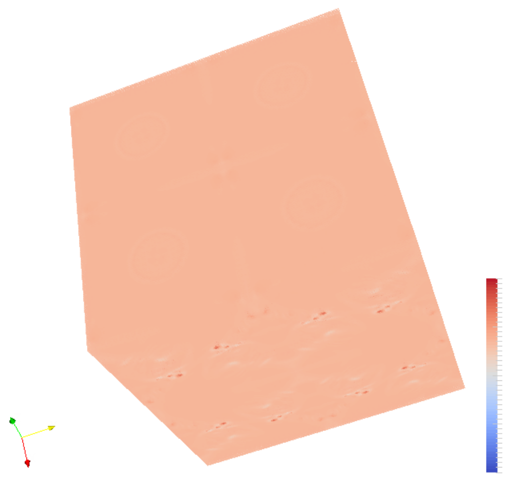


FIGURE 2.1: Shallow wireframe view of the Q-criterion for a Taylor Green simulation

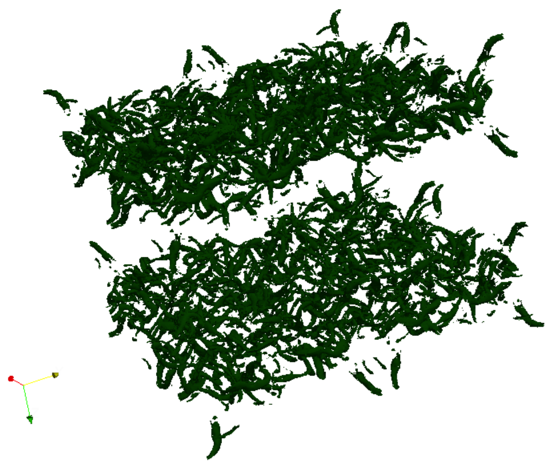
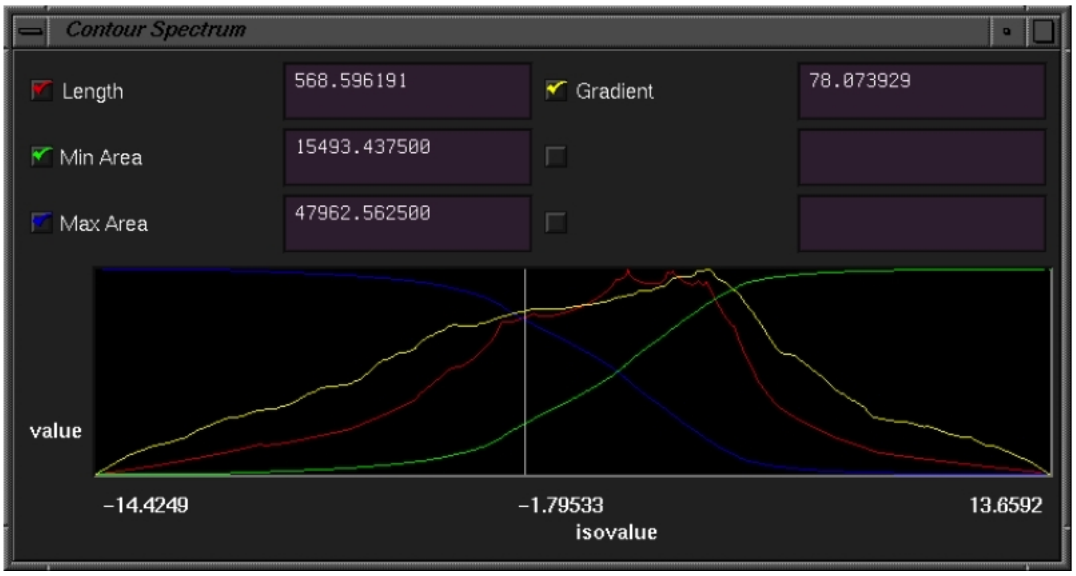


FIGURE 2.2: Vortices visibile in Paraview at Q-criterion iso-value 0.3 for a Taylor Green simulation



✓ Length	568.596191	✓ Gradient	78.073929
✓ Min Area	15493.437500	☐	
✓ Max Area	47962.562500	☐	

FIGURE 2.3: The Contour Spectrum interface showing the scalar field statistics

- The above two combined make the approach inefficient and prone to error, especially if the visualised flow has a high density of interacting features.

2.4.2 The Contour Spectrum

The concept of Contour Spectrum was introduced by Bajaj et al. in [4], aiming to improve the experience of understanding isocontours. In order to aid identifying isovalues of potential interest, the contour spectrum consists of a range of useful information displayed to the user, including statistics for each isovalue:

- the length or area of the corresponding contours
- the volume of the contours
- a metric based on the slope or the gradient of the function

As mentioned in the introduction, in the absence of domain specific knowledge, the exploratory process of choosing isovalues of interest is an iterative process, expensive for the researcher; the work of Bajaj et al. comes to address this problem. Their efficient way of computing the above mentioned information is integrated in an user interface, exemplified in Figure 2.3. We can see how the statistical values are plotted against the isovalue, and how the values are updated interactively for the selected isovalue.

The contour spectrum was later extended in multiple pieces of work. For example, in [10] Carr et al. suggest an interface that they call “flexible" isosurfaces, where several individual contours with different isovalues can be displayed and manipulated at once. A level set becomes a particular case of

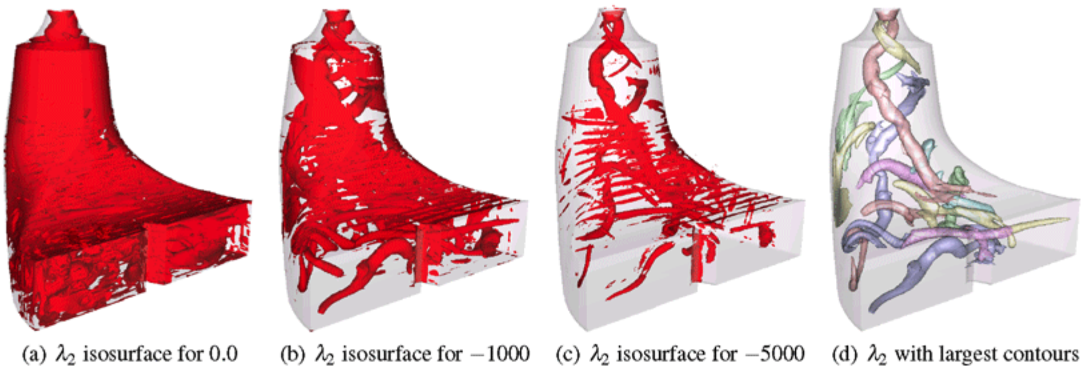

(a) λ_2 isosurface for 0.0
(b) λ_2 isosurface for −1000
(c) λ_2 isosurface for −5000
(d) λ_2 with largest contours

FIGURE 2.4: λ_2-isosurfaces for different isovalues [35]

the “flexible" isosurfaces, as it contains all contours corresponding to a fixed isovalue. In [35], Schneider et al. demonstrate the value of only visualising the largest contours, one of the information of the contour spectrum. Figure 2.4 contains three renderings of isosurfaces with particular isovalues (0, -1000 and -5000 respectively), as well as a rendering of the largest contours (the scalar value associated to the field is the λ_2-criterion, used for measuring vortices). We can clearly see how the latter is able to reveal all features, while the first three are cluttered and lose important information.

The approaches described in this subsection aim to make flow features easier to identify. Although motivated by the global structure of the flows, they are purely mathematical and the engineer or researcher analysing the data remains responsible with choosing isovalues that reveal the indeed interesting features.

2.5 Contour Trees

2.5.1 Definitions and properties

In this section we aim to introduce the reader mathematically to the concept of Contour Tree (CT). Most of the definitions and ideas presented here follow the work described in [9].

The input to the problem is a set of n points in the multidimensional space R^d, each having a corresponding scalar value. In order to extend this set of points to a scalar field, which by definition has a value associated to each point in R^d, we can define a simplicial mesh with the vertex set being the n points given, and a function f to interpolate values within each simplex. f is a linear interpolation function, $f : R^d \to R$, assigning a value to each point in the domain according to the scalar values in the vertices defining the simplex it belongs to.

We continue by defining a level set: for a specific constant value y (an isovalue), it represents the set of points in the domain where the function f takes the value y:

$$\{p \in R^d | f(p) = y\}$$

These sets are generally not connected, forming multiple connected components called isolines in 2D and isosurfaces in 3D. In data of arbitrary dimension, the term contour is typically used for naming them: so isolines and isosurfaces are particular cases of contours. A level set can consist of an arbitrary number of contours, including 0.

We are interested in the evolution of level sets as the isovalue y is varied, aspect studied in detail by the field of Morse theory. In the Morse theory, points at which the topology of the level sets change are called critical points, and a fundamental assumption is that these occur at distinct values in order to avoid certain pathological cases. In practice, this can be enforced by adding small random scalar perturbations to the initial input data, in an attempt to guarantee uniqueness [9]. At critical points it was shown that the set of possible topological changes is limited to six events:

- A new component is created at a local minimum.
- An existing component is destroyed at a local maximum.
- Multiple components are joined into a new component at a saddle point.
- An existing component is split into multiple components at a saddle point.
- A genus change for an existing component occurs at a saddle point.
- A combination of the above three can occur at highly degenerate multi-saddle points.

Considering that the components mentioned above are actually contours, we can see the CT as a graph that reflects how contours interact with each other: how they appear, join, split and disappear. Note that a genus change does not affect the tree. Therefore, the CT is constructed to respect the following properties:

- Each leaf of the contour tree corresponds to the creation or deletion of a contour, happening at a local minimum or local maximum of the parameter, respectively.
- Each interior node of the contour tree represents the joining or splitting of multiple contours.
- An edge marks the continuous existence of a contour for all the isovalues in the interval determined by the ends of the edge.

An useful example for visualising this is included in [9] and here we present a slightly modified version of it. Figure 2.5 shows the development of the level sets corresponding to a value y as it is increased. We have $y1 < y2 < ... < y6$. Figure 2.6 contains the corresponding contour tree. Starting with the global maximum (from $y6$ towards $y1$), 4 connected components appear in sequence. This is reflected by the 4 leaf nodes 7, 8, 9, and 10 at the top of the contour tree. As the isovalue y decreases, the two contours at the bottom

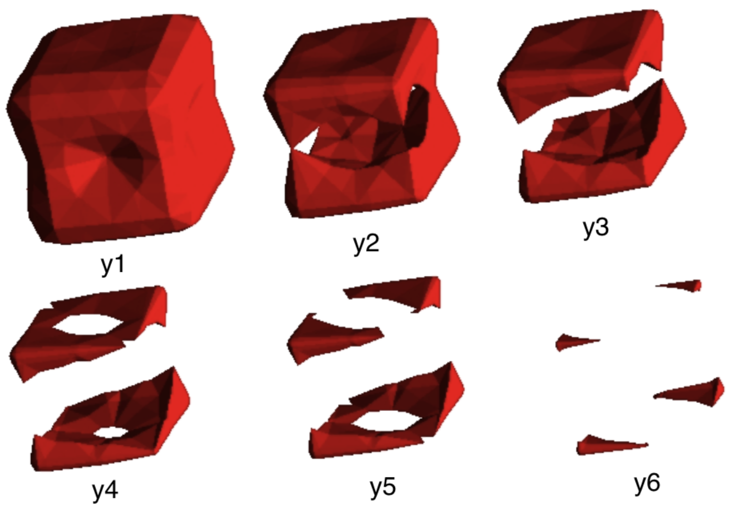


FIGURE 2.5: 3D level sets of y as y increases [9]

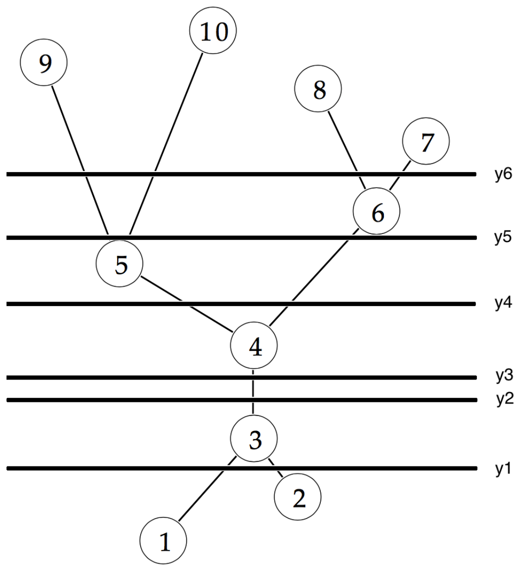


FIGURE 2.6: Contour tree for figure 2.5

join, leading to the creation of node 6 in the contour tree. At the next critical point, corresponding to an isovalue in the interval $(y4, y5)$, the two contours at the bottom also join, leading to the creation of node 5. Gradually, these two components unify, as shown at $y2$, and remain connected until they split in nodes 1 and 2, following the contraction of the inward contour and the expansion of the outward contour.

Looking further at the contour tree in Figure 2.6, we can make an interesting observation about its layout, by assuming that it follows a 2D coordinate system: the y coordinate of the nodes correspond to the values of their associated critical points in the mesh. A consequence of this is that if we intersect the tree representation with a horizontal line $y = c$, the intersection points will correspond to the connected contours with isovalue c, and their union will form the level set corresponding to this value.

2.5.2 Construction

A commonly used solution for constructing contour trees is due to Carr et al. in [9]. Regardless the dimensionality of the input space, the proposed algorithm runs in time $\mathcal{O}(n \log n + N\alpha(N))$, using $\mathcal{O}(n + N)$ memory. Here n is the number of vertices defining the mesh, N is the number of simplices, and α represents the inverse of the Ackermann function.

The algorithm begins by constructing a *join* tree and a *split* tree, which are going to be merged in order to obtain the contour tree. While an intersection point of a horizontal line with an edge of the contour tree represents a connected contour with the property $\{p \in R^d | f(p) = y\}$, the intersection with an edge of the join tree will have the form $\{p \in R^d | f(p) \geq y\}$. Therefore, as we move the isovalue from $+\infty$ to $-\infty$, the set of active elements will contain ranges of contours instead of contours: we will call them *objects*. The upper objects will grow monotonically: if points p_1 and p_2 belong to the same object at isovalue y, they will not be disconnected for any isovalue $y' \leq y$. This follows from the monotonic definition of objects. Therefore, since upper objects cannot split, we can observe that join trees will only have information about how contours appear, merge and disappear, but no information about splits. Oppositely, the split tree will only have information about how contours appear, split and disappear, reflecting the evolution of *lower objects* of the form $\{p \in R^d | f(p) \leq y\}$.

It can be seen that the join and the split trees of a field are symmetrical by definition. In fact, the algorithm is usually implemented by applying the same contour merging logic twice, once traversing the vertices of the scalar field in decreasing order to build the join tree, and once in increasing order to build the split tree. The two structures are trees, as the child of any node will correspond to a critical vertex with strictly lower or strictly higher value (for join trees and split trees respectively). In addition, both trees will have a root corresponding to an unique upper or lower object that contains all the points of the field.

Once the join and split trees are built, the contour tree is constructed by repeatedly choosing and removing a non-root leaf from the join or from the

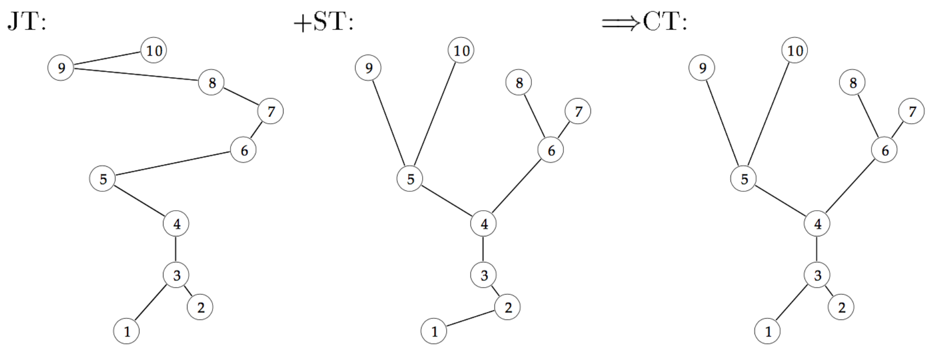


FIGURE 2.7: The join and split trees merge to form the contour tree [9]

split tree. Suppose leaf v is extracted from the join tree. v and its adjacent edge will be added to the contour tree. Also, there are two cases:

- v is a leaf in the split tree as well, and it will be simply removed.
- v is a degree 2 node in the split tree. In this case, the two nodes connected to v need to be connected by a new edge, and v will be then removed.

We can see that after each step the sets of vertices in the two trees will be invariably equivalent. Inductively, the contour tree will be successfully constructed this way. An example of contour tree construction can be seen in Figure 2.7, taken from the original paper.

For the purpose of this project, we are particularly interested in the process of building the join and split trees. Therefore, the rest of the subsection will focus on explaining this further.

Building an object tree

In order to build the object trees, the idea is to use an *union-find* data structure, detailed in section 2.7, that will reflect the currently active upper or lower objects while the isovalue is varied: if two mesh vertices belong to the same set, then they belong to the same object.

The stages of the algorithm building the upper tree are:

1. Sort the vertices of the mesh in decreasing order.
2. Traverse the vertices in order. For the current vertex v, add v in the union-find structure, as a singleton. Then update the join tree and the upper objects as follows, by looking at the mesh neighbours of v:
    - If no neighbour was initialised in the union-find set (so no neighbour was already processed), v is a local maximum. A new leaf corresponding to it is added in the join tree, meaning that a new upper object was created.

- If a (potentially complete) subset of our neighbours were processed, but all of them belong to the same set, v is just a new vertex in this upper object, and we add v to the set. Note that the upper objects the neighbours belong to are obtained by querying the union-find data structure. The join tree is not modified in this case.
- If some neighbours were processed, and they belong to two or more upper objects, v is a join node. Their corresponding union-find are merged, and v is also added to the set. A new join node, corresponding to v, is added to the tree, and it is connected to the parent nodes representing the upper objects that have just been merged.

We can see that simulating the interactions between upper objects using disjoint sets is a relatively easy process, which represents the reasoning behind not building the contour tree in one traversal - splitting objects would be computationally expensive. As mentioned above, the split and the join trees are symmetrical, and in order to build the split tree, we only have to modify step 1: sort the vertices in increasing rather than decreasing order.

2.6 Using Contour Trees to understand flows

Contour Trees are often constructed using scalars corresponding to vorticity, such as the Q-criterion. Since high values of Q are associated to the presence of vorticity, the upper leaves of the resulting contour tree will correspond to the appearance of vortex structures. In addition, a branch of the contour tree will represent a continuous class of isocontours that do not interact with others, and can be associated to a particular feature. The inner tree nodes will then correspond to the merge or to the split of features when the isovalue is decreased.

We have seen that by only using Paraview it is impossible to generate local contours associated to features, but only global contours that contain a multitude of structures. The contour trees are able to aid the dynamic extraction of local features from the field, using the fact that they represent an index into the mesh: each contour tree branch is associated to the mesh region containing the branch contours. This means that now we can only contour the region containing the flow feature of interest. This idea will be detailed further in 4.2.2.

Contour trees can also be used to identify isovalues of interest. Since each node has a Q-criterion value assigned, a researcher can often only look at the values of the vertices connected by a branch, and determine if the branch contours are of interest. This improves over the trial and error approach of guessing isovalues that contain useful structures. Taking this idea further, the contour spectrum paper [4] suggests that each arc of the contour tree can be annotated with the statistics corresponding to the contours on the arc. In this way, we can observe local properties of the features with good precision.

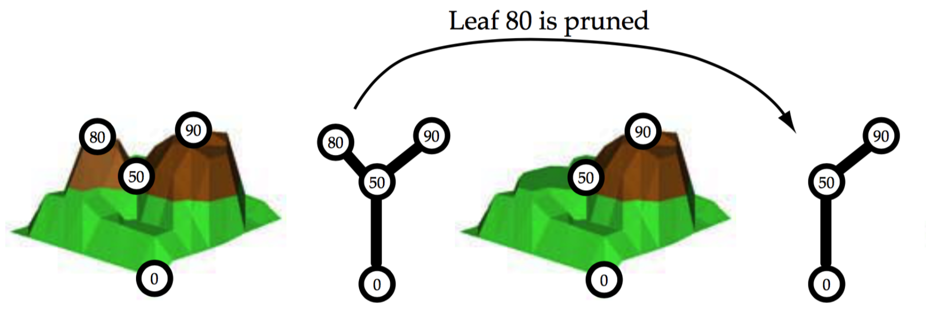


FIGURE 2.8: Leaf 80 is pruned from the contour tree. [8]

Simplifying contour trees

Contour trees computed for data from real applications are very sensitive to noise, and they typically contain a lot of local maxima and minima. These can be attributed to noise, or just to the creation or disappearance of very small features, which are typically not of interest for scientists. In order to represent useful structures for data understanding, and to speed up later analysis, contour trees are often simplified as the next step after their construction. The general idea is to repeatedly prune leaves of the tree, according to various criteria, until the remaining tree structure becomes meaningfully simpler. Note that pruning a leaf of the tree corresponds to a saddle point cancellation, as observed in Figure 2.8. Here an upper leaf is pruned, so a local maxima is flattened.

In [8], Carr et. al enumerate a number of local geometrical measures that can be used as pruning criteria, by defining the priority of removing a leaf arc from the contour tree:

- Volume. This can be approximated by the number of vertices on the arc, an easy to compute metric.
- Hypervolume. This represents the integral of function $g(\mathbf{v}) = f(\mathbf{v}) - f(\mathbf{h})$ over the region defined by the arc, where $f(\mathbf{v})$ is the value function of the mesh, and $\mathbf{h}$ is the parent vertex of the arc to be pruned.
- Persistence. This represents the difference between the arc's maximum and minimum values. Its relevance is motivated by the lack of correlation between the volume of the edge and its value range: large volumes do not necessarily span large ranges of values.

Repeatedly pruning leaves is often complemented by collapsing degree 2 nodes [8], connected to one node with larger and one node with smaller value. This is exemplified in Figure 2.9, where vertex 50 is reduced. This operation is simplifying the contour tree, but does not produce any modification in the original scalar field.

The pruning process will always be able to prune the highest priority leaf. Therefore, the tree will be reduced to one node containing the entire field

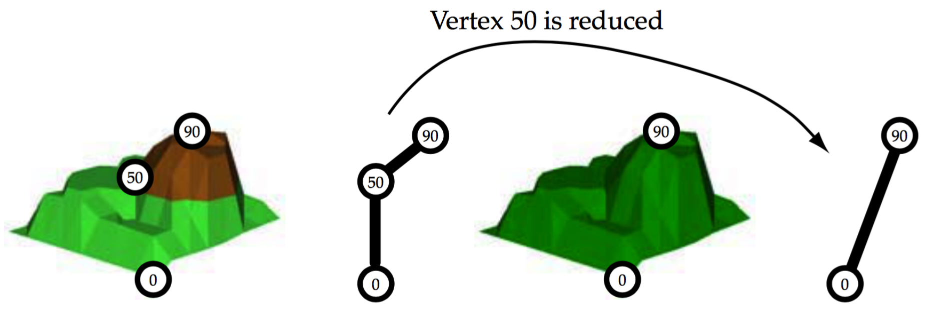


FIGURE 2.9: Node 50 is reduced from the contour tree. [8]

unless it is stopped earlier. There are two commonly used simple criteria to decide when to stop the pruning process:

- The maximum pruning priority decreases under a fixed threshold. For example, we may want to remove all leaves with a volume smaller than n, where n is chosen according to the desired aggressiveness and to the size of the field.
- The contour tree has no more than n nodes. This criteria is useful for producing fixed size trees.

Note that the effect of pruning the contour tree can be also be applied back to the scalar field data. This can be done by flattening the values of voxels in pruned vertices. This way, we can also see contour trees as an intermediate representation able to guide the process of simplifying a scalar field.

2.7 Union-find data structure

The union-find (or disjoint-set) data structure is able to keep track of elements partitioned into any number of disjoint sets, supporting two kind of operations:

- Union(S_1, S_2): the disjoint sets S_1 and S_2 are merged.
- Find(x): find the subset S that a particular element x belongs to.

In most implementations, each disjoint set is identified by one of its elements, called the set *representative*. This way, the Union operation will take as arguments two set representatives, while Find will return the representative of the set containing x. Therefore, in order to determine whether x and y belong to the same disjoint set, the condition Find(x) == Find(y) is checked - the two sets need to have the same representative.

The efficient way to perform the operations is to represent the disjoint sets as trees. The representative of a set will be the root of its tree. Each element holds a pointer to its parent, while the representative's pointer will just point to itself. Now a Find(x) query will be answered by repeatedly following the pointer to the parent until we find the representative. For a Union(S_1, S_2) operation, we can set the parent of S_2 to be S_1. For m Union and Find operations and n set elements, the complexity of this approach will be $O(m+n^2)$, the worst case being represented by the following order for unions, when the resulting tree is highly unbalanced: Union($n-1$, n), Union($n-2$, $n-1$), ..., Union(0, 1). Finding that the representative of all elements is 0 will require $0+1+...+(n-1)$ steps, which is of quadratic order.

An optimisation is to consider the sizes of the disjoint sets being connected during the Union operation, and to always keep the larger set's representative as the representative of the union. In this way, the height of the tree cannot be larger than $log(n)$, and the overall complexity is reduced to $O(m+nlog(n))$.

An improvement is now to *compress paths* whenever the Find(x) operation is called. Specifically, all the nodes on the path from x to the representative r will have their parent set to r. This way, the tree is flattened and any further $Find$ operation for these nodes will reach the current representative in one step.

Another improvement is to assign a rank to every set, and during an Union to always keep the representative to be the one with higher previous rank. In addition, if the two ranks are equal, the rank of the new representative will be increased by 1, meaning that the maximum depth of the tree was increased. Therefore, the rank is a measure for the depth of the trees, which does not take into account the path compression.

These two improvements together were proven to reduce the worst case complexity to $O(\alpha n)$, where α is the inverse of the Ackermann function. The proof is however quite involved, and hence omitted here.

2.8 Extracting isosurfaces

This section presents a number of techniques for identifying isosurfaces: regions of the scalar fields that have a constant specific value. Although the tool developed as part of the project makes use of the Visualization Toolkit (VTK) library [36] for this task, it is useful to briefly introduce some techniques potentially used behind the scenes.

Substantial research effort has been put into solving this problem, due to its importance in visualising grids and scalar fields. The Marching Cubes algorithm [25] works by considering every cell (simplex in our case) of the field and identifying the ones defining the isosurface. This becomes quickly inefficient for large datasets, and several improvements were developed. Some are based on space sub-division for cell classification, such as the Octrees method [46]. Others are based on seed sets.

A seed set is a set of mesh points with the following property: any possible connected component of any contour in the mesh contains at least one seed. If a seed set is known, the computation of isosurfaces can be done with a simple propagation algorithm based on locality. We are guaranteed

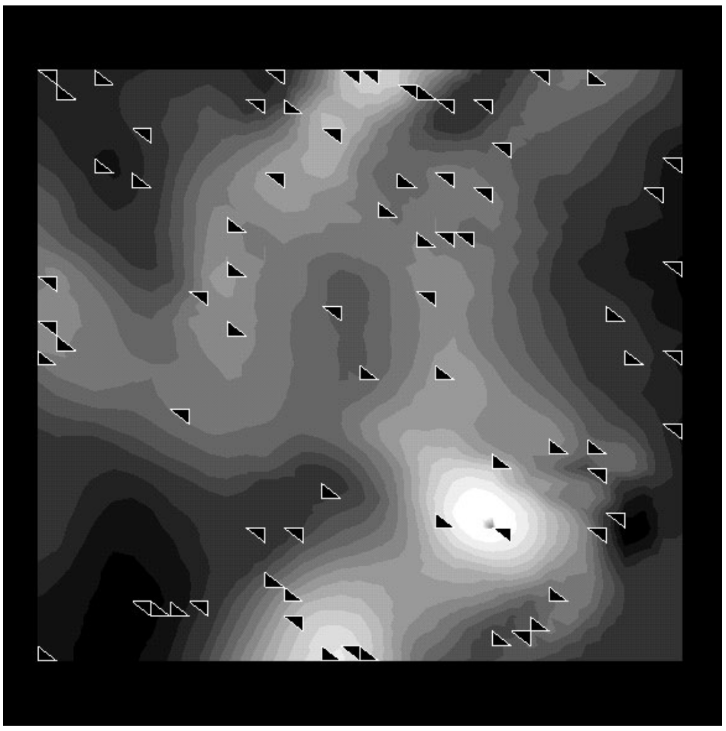


FIGURE 2.10: Seed set for a slice of wind speed data [21]

to reach every level set component due to the mentioned property of the set. Intuitively, extracting isosurfaces is efficient when the seed set is small.

[21] proposes an algorithm based on contour trees for computing minimal seed sets, requiring quadratic time and memory. They also propose an approximation of this algorithm, generating a seed set guaranteed to be no more than twice larger than the optimal one. This requires only linear memory and is much more usable in practice. Also, the performance difference while extracting isosurfaces proved to be at most insignificant. The algorithms start from the following idea: if we construct a bipartite graph where the nodes on one side are the mesh points, and the nodes on the right side correspond to contours, the problem is reduced to finding a minimal (or small enough) subset of the nodes on the left that dominates all contours on the right. A node is dominated if it is connected to at least one node belonging to the subset. Figure 2.10, taken from [21], shows the layout of a seed set in a slice of 3D data corresponding to wind speed. In VTK, extracting isosurfaces is done with a seed based approach, improved in order to counter problems with datasets containing voids and through-holes [26].

2.9 Tracking features

We have seen so far that displaying contours is a common way of visualising a scalar field produced by CFD solvers. However, in reality the simulations consist of multiple scalar fields, corresponding to a sequence of several time steps. Naturally, the question that arises is how can we analyse the entire sequence efficiently? Snapshots at particular time steps can be analysed as previously seen in this section. However, this does not offer much context on the flow evolution in time: it is difficult to observe patterns and visually follow regions of interest.

We are interested in identifying correspondence between flow features from a time step to the next one. Mathematically, a flow feature is defined to be a set of vertices, defining a volume in the mesh. However, semantically, the definition of flow features and the way to extract them depend on the application and on the origin of the flow. In our CFD aeronautics analysis, they often correspond to vortices: sets of adjacent vertices with high Q-criterion.

The problem of tracking features over time, the correspondence problem, represents an active field of study, and multiple approaches were developed. The first algorithms only make use of the spatial overlap between features, whereas many of the recently developed approaches rely on the topology knowledge offered by contour trees. In the rest of this section, we are going to discuss some of the most important algorithms for feature tracking.

2.9.1 Region-based tracking

In [34], Samtaney et al. provide an algorithm for tracking features (objects) in 2D or 3D, based on some of their attributes such as the centroid and volume. The paper assumes that the features are "thresholded clusters": adjacent mesh vertices with their scalar value exceeding a threshold. After the objects are identified, their evolution over time is subject to:

- Continuation, when an object continues from step t_i to t_{i+1}. Note that continuation accepts object rotation, translation, and (sufficiently small) size changes.
- Bifurcation, when an object splits into two or more objects at t_{i+1}.
- Amalgamation, when two or more objects merge at t_{i+1}.
- Creation, when a new feature is created.
- Dissipation, when a feature disappears.

In order to decide continuation, the correspondence criteria between two objects O_A^i and O_B^{i+1} at consecutive time steps is represented by two conditions:

$$|O_A^i \cap O_B^{i+1}| > |O_A^i \cap O_G^{i+1}|$$

where $O_G^{i+1} \neq O_B^{i+1}$ for all objects O_G^{i+1} extracted at time step t_{i+1}. O_X represents the size of the object X

$$|O_A^i \cap O_B^{i+1}| > T_{over}$$

where T_{over} is a threshold on the size of intersection. With other words, an object continues between two time steps if the overlap is maximum, and exceeds a certain threshold.

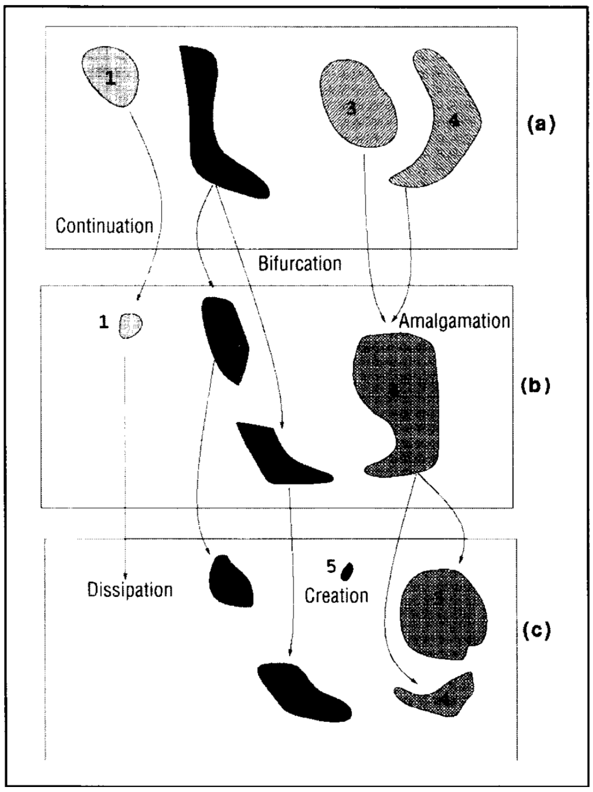


FIGURE 2.11: Tracking interactions: continuation (1), creation (2), dissipation (3), bifurcation (4), and amalgamation (5) [34]

Bifurcation and amalgamation take into consideration other properties: the mass and the volume. Details on how these are calculated are presented in the original paper. Note that they are equivalent operations: the amalgamation can be considered a bifurcation between t_{i+1} and t_i. Deciding the occurrence of bifurcation and amalgamation requires considering combinations of two or more objects from time steps t_{i+1} and t_i respectively. The number of combinations is exponentially large, but this can be mediated with certain observations that limit the testing: for example, these two interactions only occur within neighbours.

Creation happens when an object cannot be associated to any object in the previous time step, while dissipation occurs when no match is found in the subsequent snapshot. All the tracking interactions are exemplified in Figure 2.11, taken from the original paper [34].

However, being based on centroids and volumes, the approach proposed by Samtaney et al. has its limitations. For example, errors can occur when two different regions have the same center and volume, "such as a torus and an object inside the hole" [37].

Silver et al. [37] addressed this problem by using overlap tracking to refine the idea described above. The concepts of continuation, bifurcation, amalgamation, creation and dissipation are equivalently defined, but their correspondence criteria are changed. As an example, object O_A^i is now the

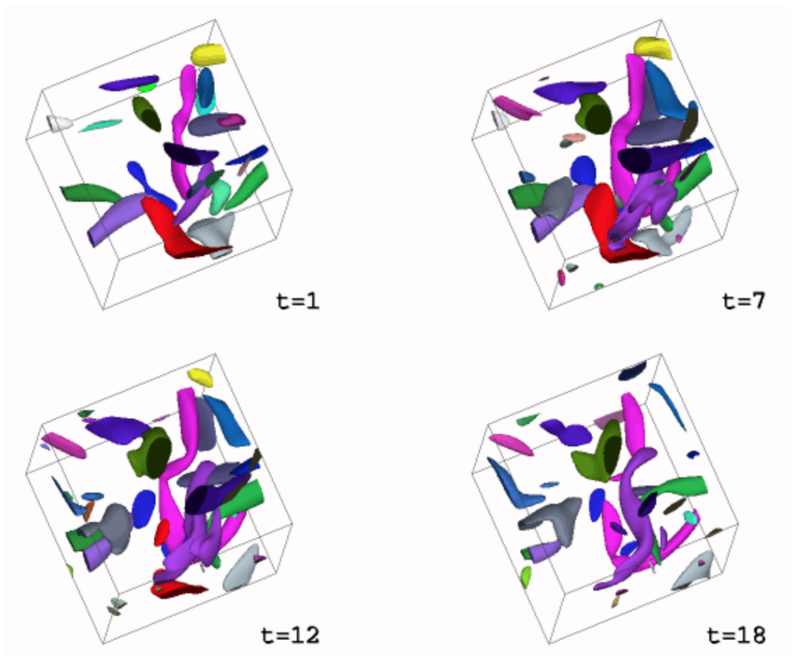


FIGURE 2.12: Overlap based tracking in a CFD dataset [37]

continuation of object O_B^{i+1} if and only if the following conditions are satisfied:

$$|O_A^i \cap O_B^{i+1}| > 0$$

$$\frac{max(|O_A^i \setminus O_B^{i+1}|, |O_B^{i+1} \setminus O_A^i|)}{max(|O_A^i|, |O_B^{i+1}|)} < T_{under}$$

where T_{under} is again a specified threshold. Note that continuation is now conditioned by overlapping. Octrees [33] were used for efficiently computing the above values for the relevant pairs of objects in different time steps.

Figure 2.12, taken from the original paper, demonstrates the successful tracking of features in a CFD simulation dataset.

In [28], Muelder and Ma propose the idea to track features by trying to predict their positions over time. This is motivated by the fact that, when dealing with data originating from real applications, features often follow predictable paths. Depending on the approach used, the prediction strategy will look at how a feature changed position across a number of past time snapshots. Consequently, a standard overlap-based tracking algorithm is used for the first time steps, until enough past information is available to enable prediction. Once the continuation of a feature in the next snapshot is predicted, they are using a proactive approach to detect the actual corresponding feature: rather than firstly extracting the features, and then trying to match them to the available predictions, they use the predictions as seeds used to extract the features of the next time step. This is done using breadth-first search algorithms on the mesh structure, in order to grow or shrink the predicted feature, transforming it in an actual feature. Note that they are defining features to be connected regions with scalar values larger than a

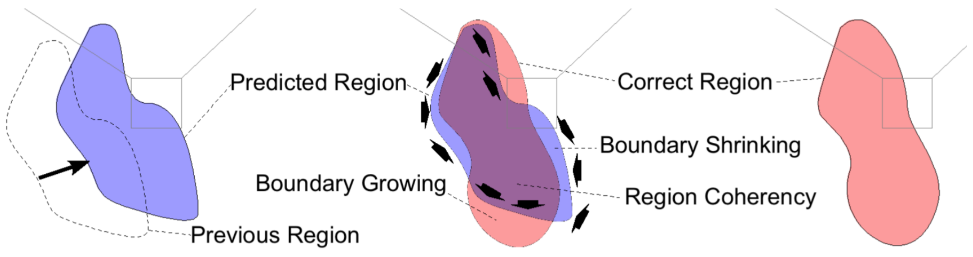


FIGURE 2.13: Adjusting the predicted boundaries to the ones of a feature [28]

fixed threshold. Figure 2.13, taken from the original paper, shows why both shrinking and growing the predicted boundaries is required in order to obtain a feature.

Another overlap-based approach for tracking is introduced in [19] by Ji et. al. They propose to use 4D isosurfacing on the mesh where one dimension is time, and the other three correspond to the structure of the mesh. Therefore, each mesh vertex will now have some coordinates (x, y, z, t), and the value of the vertex (x, y, z) in the t^{th} field. The idea is to consider isovolumes of this field, which span multiple time steps. Since these are connected, they manage to obtain tracking information for the features contained in the isovolumes. The merge and bifurcation events are detected by looking at the critical points inside the regions. However, this approach is not very applicable in practice: the addition of the extra dimension generates huge fields for real data, making the algorithm slow and memory demanding.

2.9.2 Tracking as a global optimization problem

The approaches mentioned so far represent local tracking techniques, where features are matched independently, by exclusively using local information. In [18], Ji and Shen translate the problem of computing feature correspondences to a global optimization problem. Their target is to obtain better matching accuracy, by considering the global context of the field, with the improvements being significant when tracking small objects, or when the temporal sampling of the underlying data is low relatively to its velocity. Indeed, the fundamental principle of most region-based methods is that two features cannot correspond unless they overlap spatially; if the temporal sampling is insufficient, this precondition may not hold, as demonstrated by Figure 2.14 taken from the original paper.

Ji and Shen propose to use the Earth Mover's Distance (EMD) as a cost function between features from different snapshots. EMD is a metric originally used in statistics which reflects the minimal amount of work required to translate one distribution into another. Intuitively, given two spatial distributions, one can be seen as the mass of Earth spread in space, while the other represents the holes in the same space. Obtaining the minimal EMD is now translating to the minimal effort required to fill the holes with mass. EMD can be solved polynomially using the Hungarian algorithm [29].

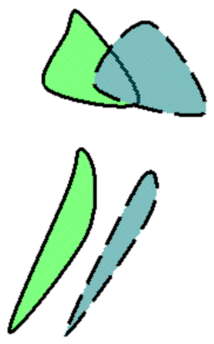


FIGURE 2.14: Possible effect of insufficient sampling. Overlap-based algorithms will fail to track the bottom feature [19]

In order to compute the cost of matching two features, these are first decomposed into the cells they contain, which represent the two distributions in the EMD problem. Then EMD is computed in terms of the Euclidean distance between cells. Therefore the problem is now reduced to computing a minimal cost matching between two sets of features, when a cost function defined on pairs of features is available. Since the algorithm must consider all possible evolutionary events of features (including merges and bifurcations), the search space of this optimization problem can become huge. To address this, the paper also shows how a branch and bound [22] method improves the runtime of the algorithm.

2.9.3 Based on critical points

A different idea is to track features using tracking algorithms for critical points. The general concept was introduced for the first time by Theisel and Seidel in [41]. They proceed by representing the dynamic behaviour of features as the stream lines of a higher dimensional vector field. They suggest a way to construct this vector field (the feature flow field) and make use of existing stream lines integration methods in order to perform the feature tracking in fields. The idea is briefly exemplified in Figure 2.15, where it can be seen how the behaviour of feature v is tracked by tracing the stream lines of the field from the feature; however, the supporting mathematical concepts are quite involved and hence omitted here.

The work of Theisel and Seidel is an interesting theoretical approach, but a direct implementation of it is very sensitive to noise, since it heavily relies on numerical methods such as derivatives or differential equations. Weinkauf et al. improve the applicability of the idea on data coming from real applications, by proposing a more stable feature flow field in [44].

2.9.4 Based on contour trees

In [39], Sohn et al. combine overlap based tracking with contour trees. They are interested in the evolution of contours at fixed isovalues over time. This

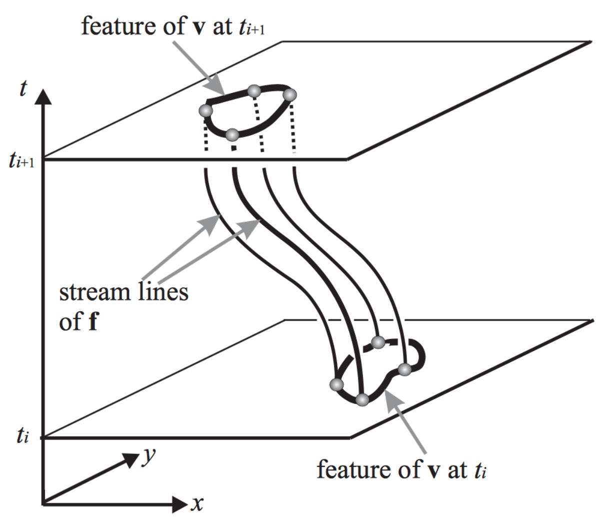


FIGURE 2.15: Feature tracking using feature flow fields. [41]

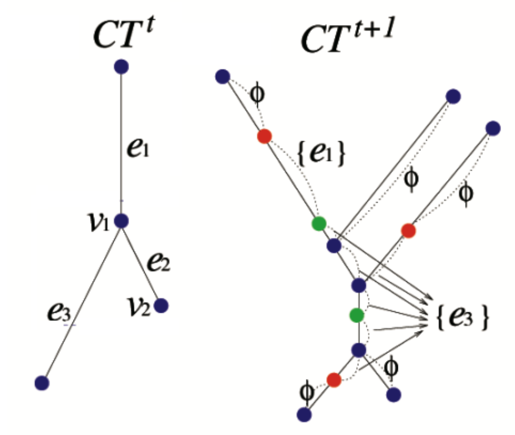


FIGURE 2.16: Contours correspondences between two trees. [39]

is done by detecting overlapping pairs of contours from consecutive time steps: this way they can define ranges of values for which edges from two contour trees overlap. Figure 2.16 exemplifies the contours correspondence calculation for two consecutive contour trees. A point labeled with edge set E at isovalue w on an edge of the second contour tree means that the contour evolves from the contours at w on edges in E from the first time step.

They also suggest the following idea: varying time is similar to varying the isovalue within one time step. With other words, tracking the points in time where sets in consecutive time steps start and stop to overlap is not different from tracking the points (critical isovalues) where level sets split and join in one scalar field. They concretize this idea in a structure called the *Topology Change Graph* (TCG), which highlights the topology changes of time-varying isosurfaces. Using the TCG, they are immediately able to visualise the evolution of individual time-varying isosurfaces as they merge, split, create or disappear.

All the approaches mentioned so far depend on the definition of a feature being invariable. However, this may not be always acceptable: for instance, in a mesh where features are simply regions with high scalar value, we may want to adjust the threshold defining the features. In applications with large datasets, recomputing the feature correspondences with every feature definition change is likely to be unfeasible.

In [45], Widanagamaachchi et al. address this missing dimension. They propose a new tracking graph, augmented with certain meta information, which can be used to quickly recompute the correspondences between edges and contours for distinct and dynamically defined features. Therefore, while the approach described in [39] requires complete recomputation of the correspondence information every time the way we define the features change, this can now be avoided. The main observation is that if there are many edge correspondences between subtrees S_A^i and S_B^{i+1} in consecutive time steps, the features defined by the two subtrees are likely to correspond at some point, depending on their definition.

We can see how the algorithms described in this subsection rely on the hierarchical structure of the data, as described by the contour or join trees. By using this nesting relationship between regions, tracking information is computed for the entire field. In contrast, by only using the approaches assuming flat features seen above, this is likely to be a more expensive task.

In [32], Saikia and Weinkauf propose the idea to use the knowledge of all available time steps in order to decide the correspondences between any two consecutive ones. They therefore assess the improvement brought by having a global, rather than a local temporal context when deciding correspondences.

They are extracting the features of interest only using the join trees of the fields. This is motivated by the fact that in many applications the contour tree is dominated by the join tree, scientists being more interested in only observing how features appear and join. They then proceed by building a directed acyclic graph that contains all the features of all time steps, where features of consecutive time steps are connected by arcs with values representing the cost of matching them. These costs are obtained considering both spatial overlap and the similarity between regions' data. Starting from a given region, they are then applying the Dijkstra algorithm in order to find the shortest path through the graph built as described above, which corresponds to the evolution of the region in time.

2.10 Existing work

The current project partially reuses some of the code written by Daniel Simig for his undergraduate thesis [38]. During his project, he developed a tool able to extract and visualize the features of the data using contour trees. His results relevant to this project are the following:

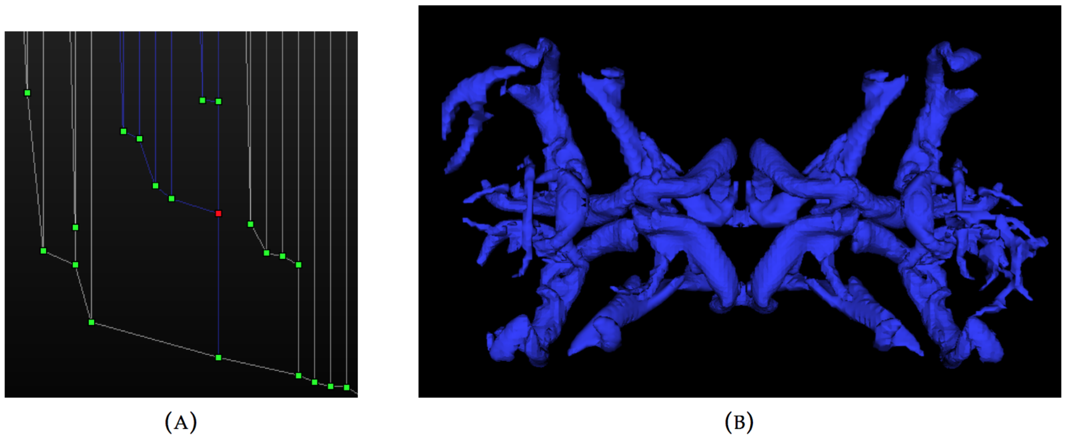


FIGURE 2.17: A selected (red) node and the contour corresponding to its subtree [38]

1. Creating a C++ API for computing contour trees, over the *libtourtre* library [24]. libtourtre represents an open-source implementation, written in C, of the algorithm described in 2.5.2.

2. The implementation of a number of reductions for contour trees, as described in 2.6. There were two reductions primarily used throughout the project:

   - Pruning by volume, to remove insignificant features
   - Pruning by value, in order to compact arcs corresponding to negative values in the data. With regards to the Q-criteria, the negative values correspond to noise and have no physical meaning.

3. Building an interactive GUI that uses the simplified contour tree as an index in a time step of a flow. The user is able to choose an edge of the contour tree and visualise one isosurface on that edge, as seen in Figure 2.17, taken from the original thesis. It represents a local contour, rather than a global one, exemplifying the dynamic feature extraction idea presented earlier in this chapter. This is achieved by generating the contour only on the mesh region corresponding to the arc and its subtree, which are reproduced using the vertex to arc mapping associated to the contour tree. The VTK library is used for extracting the isosurface of the region, using a so-called VTK contour filter.

2.11 Summary

In this chapter, we presented general Computational Fluid Dynamics knowledge which will be particularly relevant for understanding Chapter 6, where we use our tool in the context of a real problem existing in the field, as well as for the understanding of certain parts of Chapter 5 where the achievements of the current project are evaluated. We then introduced contour trees, and described an efficient and commonly used algorithm for constructing them, which supports the understanding of the feature tracking algorithm described in Chapter 3. We then presented several feature tracking techniques existing in the literature, which we evaluate against our algorithm in Section 5.5.

Chapter 3

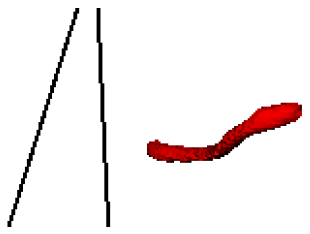
Feature tracking

The main focus of the project was to implement an algorithm for deciding correspondence of features between two scalar fields, having in mind its applicability to time snapshots of fluid simulations.

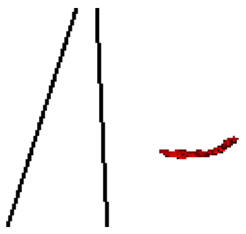
As discussed in Section 2.9, there is no fixed way to define the features. However, we decided to use contour trees to extract them, such that any feature will correspond to an arc of the contour tree. The feature can then be visualized by contouring the field subregion defined by that arc and its subtree at any isovalue belonging to the arc. An example of vortex structure belonging to the Taylor Green Vortex simulation is included in Figure 3.1, where its local local context is contoured at two different isovalues (of course, both within the scalar range of the nodes connected by the arc). We can see that the contour appearance is slightly different, but they indeed represent the same feature. The tracking algorithm should therefore focus on the correspondence between arcs of two contour trees.

In addition, we would like the algorithm to be able to deal effectively with noisy data sets, so it should support the integration with contour tree simplification procedures. Moreover, since the process of storing the data produced by CFD solvers has recently become a major bottleneck in the field, the in-situ capabilities of the algorithms for understanding and visualising the simulations become extremely relevant. Therefore, our approach should be fundamentally applicable at the time the simulation is performed.

In the rest of the chapter, we are going to describe the feature tracking algorithm developed as part of the project. The research contributions associated to it are the following ones:



(A) Isosurface defining a feature at 1.2



(B) Isosurface defining a feature at 2.17

FIGURE 3.1: Two isosurfaces belonging to the same contour tree arc, taken at different isovalues.

- We propose a new algorithm for feature tracking, based on computing arc correspondences between contour trees. This is achieved by extending the approach described by Sohn et al. in [39] with a way of converting contour correspondences to correspondences between arcs.
- We improve the contour correspondences computation algorithm described by Sohn et al. to make it applicable to contour trees that have been previously simplified.
- We demonstrate the efficiency of tracking features in a wide range of scenarios, while particularly focusing on the ability of tracking vortex structures in turbulent fluid flow simulations. This work is presented in Chapter 5.
- We assess the feasibility of running the algorithm at the time the fluid simulations are performed, work included in Section 7.3.1.

3.1 Computing contour correspondences

Remember that each point on a contour tree arc corresponds to a local isosurface. Given two scalar fields, we want to know for each local isosurface of the second field the (possibly empty) set of local isosurfaces from the other snapshot that evolve into it. With respect to the contour trees of the two fields, we therefore want to label each point of an arc of the second tree with the correspondent set of arcs from the first one. Intuitively, this is a difficult task, since the domain of the isovalues is continuous. However, the fact that we approximate the spatial overlap between isosurfaces using the number of shared mesh vertices means that the correspondence information for an arc can only change at mesh vertices, therefore a discrete domain. Moreover, in the case of data coming from real applications the number of arc regions labeled differently is going to be much smaller compared to the number of vertices, due to the expected coherence within the data [39], as we will see later in section 5.3.

Figure 3.2 shows an example of correspondences computed between two contour trees, CT^t and CT^{t+1}. Here, we can see how segments of edges of CT^{t+1} are labeled with edges from CT^t. If a point on an arc of CT^{t+1} is labeled with an edge, the contour defined by that point evolves from the contour at the same value on that specific edge of CT^t. The layout of the trees follows a linear vertical scale, corresponding to the scalar values of the nodes. On the other hand, the horizontal positions of the nodes are arbitrary.

More formally, for each edge $e_k^{t+1} \in CT^{t+1}$, we are going to output a division of it in several regions of contours of the form
$\{((HI(e_k^{t+1}), x_1), E_1), ((x_1, x_2), E_2), ...((x_r, LO(e_k^{t+1})), E_{r+1})\}$, where $HI(e)$ and $LO(e)$ denote the two contour tree nodes connected by e. For any (x_k, x_{k+1}) region of contours, the set of correspondent edges E_{k+1} will have the form $e_{p_1}^t, e_{p_2}^t ... \in CT^t$, such that any contour in the region (x_k, x_{k+1}) evolves from the set of contours at the same isovalue on edges in E_{k+1}.

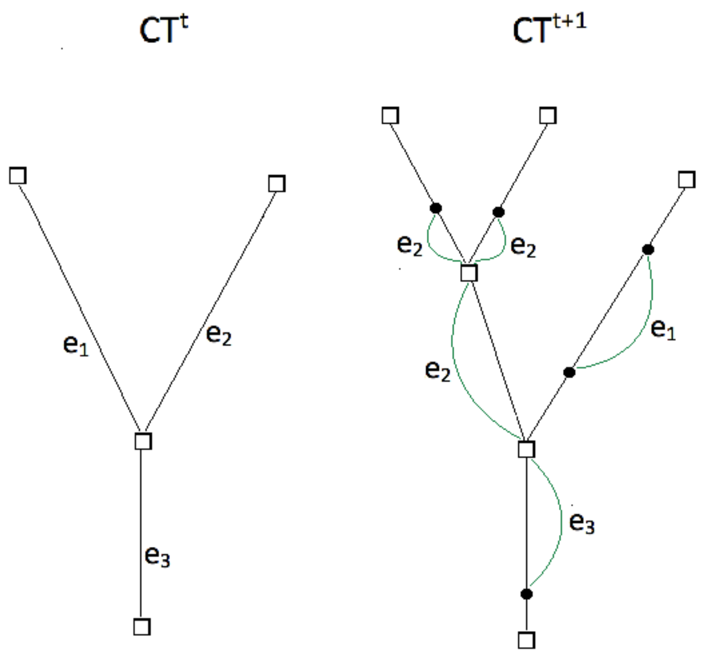


FIGURE 3.2: Contour correspondences computation example between two trees.

3.1.1 Contour correspondence definition

We proceed by giving a formal definition for the correspondence between two contours.

In 2.5.2, we have seen the definition of upper objects and lower objects for a fixed isovalue y: connected components of mesh vertices having value $\geq y$ or $\leq y$, respectively. Therefore, for a fixed y at a timestep t, we can define two sets: $\{UP_1^t, ..., UP_n^t\}$ and $\{LO_1^t, ..., LO_m^t\}$ containing the upper and lower objects, respectively. Naturally, any contour will be on the border of one upper object and on the border of one lower object, objects determined by the threshold equal to the contour isovalue.

We are now able to define the correspondence between two contours C_k^t and $C_{k'}^{t+1}$ belonging to time steps t and $t+1$. For C_k^t and $C_{k'}^{t+1}$, let's assume their upper and lower objects are UP_x^t, LO_y^t, $UP_{x'}^{t+1}$ and $LO_{y'}^{t+1}$. We say that $C_{k'}^{t+1}$ corresponds to C_k^t if and only if there is a *significant* overlap both between UP_x^t and $UP_{x'}^{t+1}$ and between LO_y^t and $LO_{y'}^{t+1}$.

In order to assess the significance of overlap between two upper (or lower) objects, the following condition is used:

$$\frac{|UP_x^t \cap UP_{x'}^{t+1}|}{min(|UP_x^t|, |UP_{x'}^{t+1}|)} > T_{over}$$

where T_{over} is a specified threshold which filters out undesirable matchings and can be changed depending on the application. The robustness of upper and lower objects against the changes occurring in consecutive fields is therefore used to track the evolution of contours.

We notice that the overlap degree is relative to the smaller object among the two. As an example, this means that if the first object is contained in the second one, the overlap will always be significant regardless the difference between their dimensions. On the other hand, the algorithm is not able to identify correspondences if the upper objects do not share any vertex, a typical limitation for overlap based tracking algorithms. However, we considered this an acceptable limitation, having in mind that the tracking algorithm would be eventually running in-situ, where the temporal sampling can be assumed to be sufficient.

The T_{over} threshold is a confidence scalar, which should be adjusted according to the underlying data. Specifically, there are two basic aspects often taken into consideration:

- The temporal sampling of the underlying data. When the sampling is lower, the threshold should be sufficiently low.
- The velocity of the features of the data. When the velocity is higher, the threshold should be again sufficiently low.

On the other hand, choosing a too low threshold can lead to the report of false match positives.

3.1.2 The algorithm

We have seen that the correspondence criteria consists of two independent halves: assessing the significance of the overlap between upper objects and between lower objects, respectively. The algorithm therefore consists of computing contour correspondences according to upper and lower objects separately, in two different passes, and then intersecting this information in order to obtain the correct contour correspondences. The process of computing the upper objects correspondences for two contour trees CT^t and CT^{t+1} can be summarized as follows:

- Label each edge in the join tree of CT^t with a set of edges of CT^t.
- Label each edge in the join tree of CT^{t+1} with a set of edges of CT^t. This is done according to the overlaps between the upper objects of the two time steps, during their growth.
- Label each edge in the join tree of CT^{t+1} with a set of edges of CT^{t+1}. Then use this, together with the information from the previous step, to obtain a partial contour correspondence information between the two initial contour trees, only by looking at the overlap significance between upper objects.

The lower objects correspondences will be computed similarly, using the fundamental symmetry between upper and lower objects. For the rest of the subsection, we are going to focus on the process of obtaining the upper objects correspondences, and at the end explain the intersection process between upper and lower objects correspondences, in order to obtain the contour correspondences.

Building a labeled object tree

The first operation presented is building an join tree for one of the fields, using the process described in Subsection 2.5.2. The additional aspect is that, during construction, the edges of the join tree will be labeled with the corresponding ones of the contour tree. An example of building a labeled join tree from its corresponding contour tree can be seen in Figure 3.3. We can see how an edge of the join tree can have more than one label, corresponding to splits happening in the contour tree.

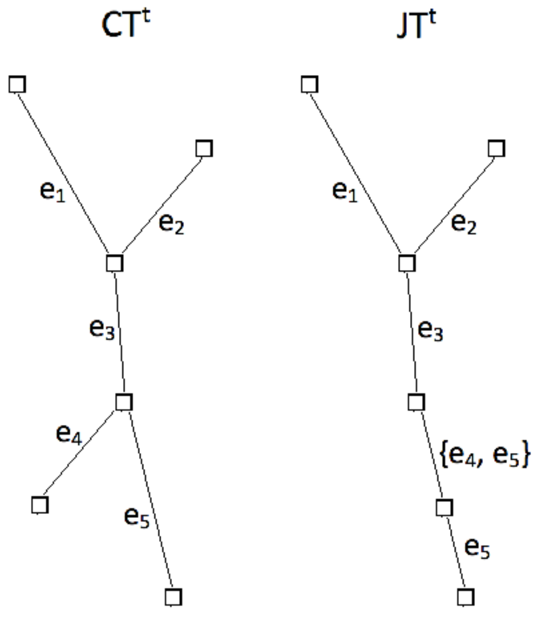


FIGURE 3.3: A labeled join tree and its corresponding contour tree.

We are going to initialize an union-find (disjoint-set) data structure that will contain the active upper objects while sweeping the isovalue from ∞ to $-\infty$. While traversing the mesh vertices in decreasing order by their scalar value, for a vertex v we have again the cases encountered while building the join tree:

- No neighbour of v has been processed yet. Therefore, we are at a global maximum, so:
  - A new set, corresponding to a new upper object, is added in the union-find structure
  - A new node n is added to the tree. Its label set will be $L(n) = S(CTNode(v))$, where $CTNode(x)$ represents the node corresponding to x in the contour tree, while $S(n)$ denotes the set of son edges of node n. Note that $CTNode(v)$ is well defined, since v is a global maximum, so there is a node (an upper leaf) corresponding to v in the contour tree.
- v is a join:

  - A new join tree node n is created, connected to the k parent nodes that correspond to the objects being joined. The new node will inherit their labels, but the ones of the direct parent edges will be replaced by the label of the join edge in the contour tree (the child edge of v). Therefore, the label set is:

    $$L(n) = \bigcup_{i=0}^{k-1} L(T(X_i)) - P(CTNode(v)) + S(CTNode(v))$$

    where $X_{0..k-1}$ are the objects being merged, $T(X)$ denotes the newest join tree node added to object X, and $P(n)$ denotes the parent edges of the contour tree node n. Note that $\bigcup_{i=0}^{k-1} L(T(X_i)) = P(CTNode(v))$ holds unless one of the merged objects suffered a split earlier in the tree
  - The sets corresponding to the joined objects are merged.

- v is a split:
  - v is added in the disjoint set of the object that it belongs to. There is one such object, since in a well-behaved tree v is not an upper leaf as well
  - A new join tree node n is created, connected to the newest node in the object X of v, with the label set:

    $$L(n) = L(T(X)) - P(CTNode(v)) + S(CTNode(v))$$

- v is a lower leaf:
  - v is added in the disjoint tree object it belongs to
  - A new node is created in the join tree, connected to the newest node of the object v belongs to, with the label set:

    $$L(n) = L(T(X)) - P(CTNode(v))$$

- v is a regular vertex. In this case, we only have to update the object containing v.

In the above explanation, labels were assigned to nodes, while previously they were corresponding to edges. The two notations are however equivalent, since all the join tree nodes have at most one child arc (although potentially more parents): the labels of an arc are therefore those in its parent node.

By following the above process, we can see how additions to the label sets correspond to the creation of new contour classes, while removals from the label sets correspond to the termination of contour classes as we sweep the scalar isovalue from top to bottom. The splits are reflected in the join tree by the existence of degree 2 nodes, where the label set is changed.

Building a join tree reflecting overlaps

Using the algorithm described in the previous subsection, we are now in the position to build a join tree for the first time step, JT^t, having the nodes labeled with edges from CT^t. In this subsection, we aim to build a join tree for the second time step, JT^{t+1}, whose nodes should also be labeled with edges of CT^t in order to reflect significant overlaps between upper objects.

Similarly to the previous subsection, we are going to keep track of how upper objects evolve as we decrease the isovalue. Here we will store the objects of the two time steps in two distinct union-find structures, and update them as we traverse the vertices in interleaved decreasing order by their value. In total, we are going to process vertices twice the number of vertices in a mesh (as both fields are built on the same mesh), and take action according to the type of the current event and whether it comes from time step t or $t + 1$.

When processing a vertex from the second time step, we are going to add a new node in JT^{t+1}, similarly to the previous section. This ensures that all nodes in CT^{t+1} will have a correspondent in the join tree, as expected. The label set of the node will be the union of the label sets of all parents for the moment, but updated as described in the following paragraphs.

Apart from keeping track of how the objects evolve, we will also need to keep track of pairs of objects, from different time steps, that overlap. Therefore, when processing a vertex v from any of the two timesteps, after the upper objects are updated, we are also interested in updating the collisions between objects. Apart from knowing the pairs of objects that overlap, we will also know the volume (approximated by the number of shared vertices) of the overlapping region and whether the overlap is considered significant or not. This information enables us to check the significant overlapping condition. Therefore, after the objects are updated appropriately using the join tree construction rules, for a vertex v we have the following cases:

- v is an upper leaf in either contour tree, and v has already been included in an object of the other time step. This means that the vertex v has just created an object that will share one vertex (v itself) with the older object containing v in the other time step, so a new collision is added.
- v is not a non-upper leaf critical node. We are going to sum up all the collisions of the objects merged at this point (just one unless v is a join) saying that the collision with object X is significant if at least one parent had a collision with X previously marked as significant.
- v is a regular node. We only increase by 1 the size of the object v belongs to.

We will now proceed by traversing all the collisions of the object currently containing v (there will be exactly one object), and potentially create new nodes in JT^{t+1} if the significance of overlaps is changing. If the collision $(UP^t_k, UP^{t+1}_{k'})$ became significant, we are going to create a new node in JT_{t+1}, child of the last node created for object $UP^{t+1}_{k'}$, which will have the labels of the parent and also the labels of the JT^t edges the last vertex of UP^t_k belongs to. Similarly, if the collision becomes insignificant instead, we create a new node removing those labels.

Figure 3.4 shows an example where this operation is performed. The red circles correspond to points where the significance of the overlap between upper objects changes, and their position depends on the underlying fields. We can also notice that JT^{t+1} is built such that it represents a valid join tree for CT^{t+1}.

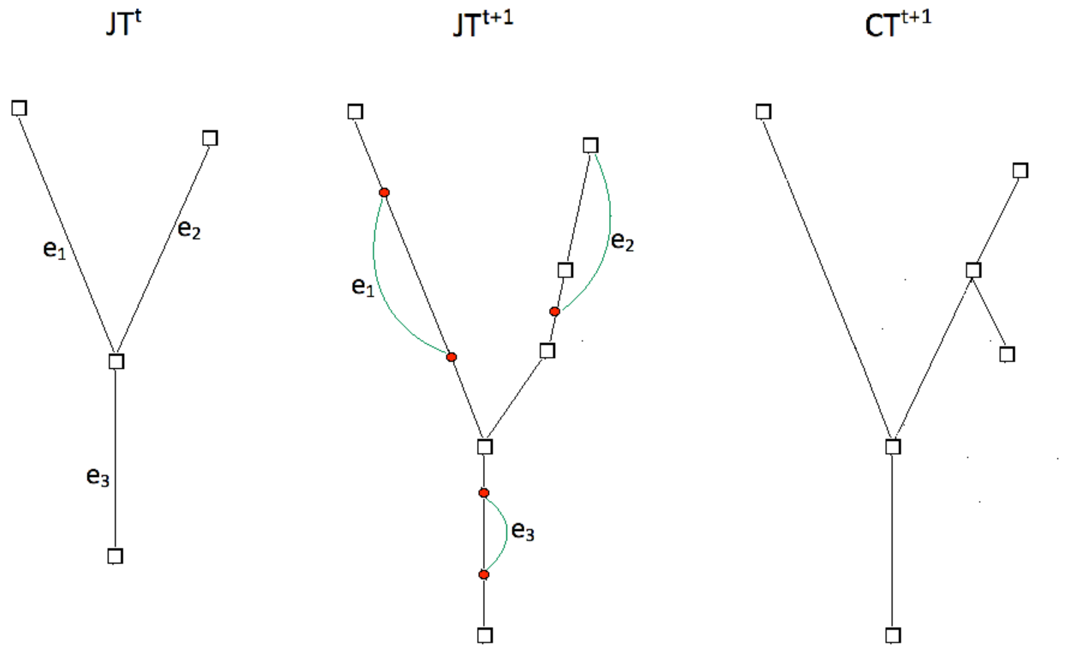


FIGURE 3.4: A join tree reflecting collisions between upper objects.

Labeling a contour tree from a join tree

In the last two subsections, we have seen the two operations that enable the construction of two join trees for the second field, one labeled with edges of the first contour tree and one labeled with those of the second contour tree. At this point, we proceed by introducing a simple way to combine these two join trees in order to generate the first half of correspondence information for edges of CT^{t+1}, the half considering only the upper objects.

The idea is that if a point on a join tree arc at isovalue x has a label e^t in the first join tree, and a label e^{t+1} in the second join tree, the isosurface at x on edge e^{t+1} of CT^{t+1} will evolve from the isosurface at x on edge e^t of CT^t.

More formally, assume the two join trees are JT_1^{t+1} and JT_2^{t+1}, being labeled with edges of CT^t and CT^{t+1}, respectively. Consider edge j of JT_1^{t+1}, having the label set $E_1 = \{e_1^{t+1}, e_2^{t+1}, ...\}$. From the construction of JT_2^{t+1}, we are guaranteed that $HI(j)$ and $LO(j)$ will also correspond to two nodes in it, say n_1 and n_2. We are now able to decompose (n_1, n_2) in regions by following the son edge from n_1 until we hit n_2, obtaining the regions $\{((n_1, n_{k_1}), E_{2_1}), ((n_{k_1}, n_{k_1}), E_{2_2}), ...((n_{k_m}, n_2), E_{2_{m+2}})\}$. Now each region $((n_{k_p}, n_{k_{p+1}}), E_{2_{p+1}})$ will represent, on every edge in E_1, a consecutive region of contours between the two scalar values of n_{k_p} and $n_{k_{p+1}}$ that evolve from $E_{2_{p+1}}$. Remember that $E_{2_{p+1}}$ is a set of labels assigned to labels in the first contour tree. Therefore, at this point we managed to label (regions of) edges of the second contour tree with edges of the first one, having only looked at half of the overlap condition: only considering upper objects.

An example is included in Figure 3.5. Here, for the arc labeled $\{e_3^{t+1}, e_4^{t+1}\}$ in JT_1^{t+1}, there are two non-empty regions with different labels (i.e. $\{e_2^t, e_3^t\}$ and $\{e_2^t\}$) on the chain between the same nodes in JT_2^{t+1}. Both e_3^{t+1} and e_4^{t+1} edges of CT^{t+1} will be labeled with the two regions in JT_2^{t+1}, so with edges of CT^t.

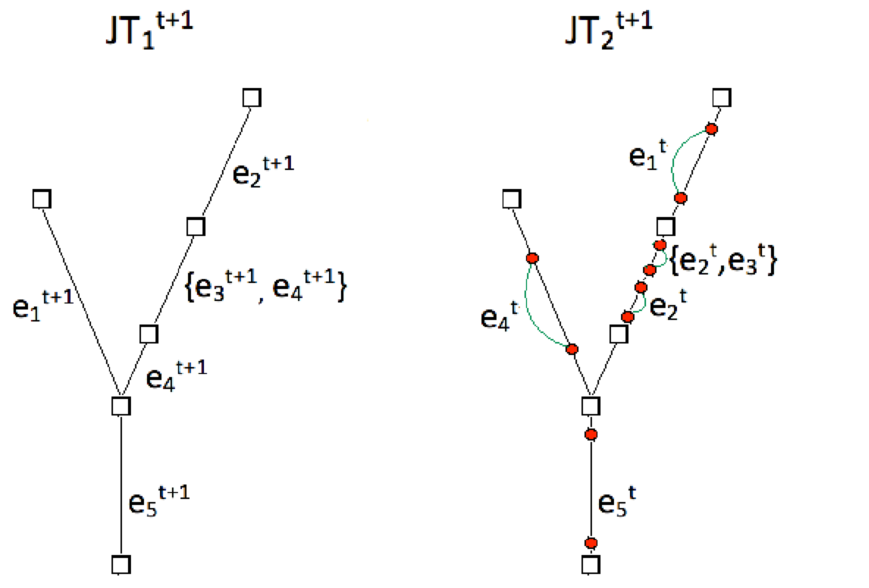


FIGURE 3.5: Two join trees, labeled with edges from different time steps contour trees.

Intersecting two labeled contour trees

The final step of the algorithm is represented by intersecting the label sets computed for each edge of the second contour tree during the two passes: once considering upper objects and once considering lower objects.

For every contour tree edge (n_1, n_2) of CT^{t+1}, we will have two divisions into regions: $\{(n_1, n'_1), (n'_1, n'_2), ..., (n'_{m_1}, n_2)\}$ and $\{(n_1, n''_1), (n''_1, n''_2), ..., (n''_{m_2}, n_2)\}$. We are going to merge these regions, ending up with $m_1 + m_2 + 1$ regions, assuming their borders do not coincide.

An example is included in Figure 3.6, which focuses on the process of intersecting one pair of arcs, represented by the top two horizontal lines. We can see how the label sets of overlapping regions are intersected and a new labeling is obtained for the edge.

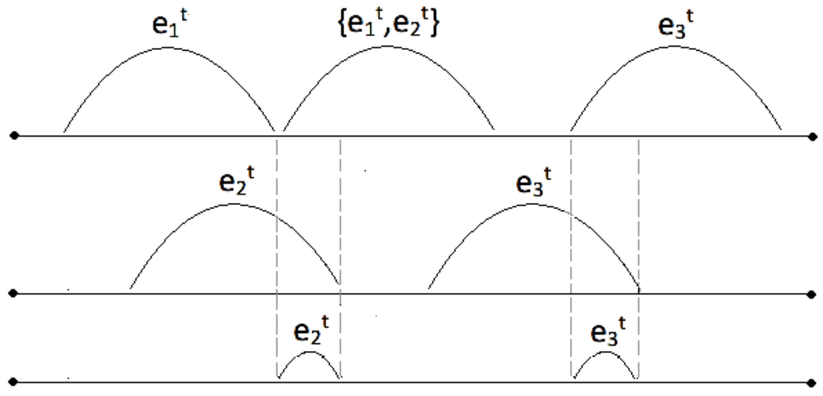


FIGURE 3.6: The operation of intersecting the labels of two contour tree arcs.

3.1.3 Implementation

In order to compute the tracking information between the features of two scalar fields, our implementation consists of the following sequence of events:

1. Build the two contour trees of the scalar fields, achieved using the *libtourtre* library.
2. Simplify the two contour trees, as described in Section 2.6. Then update the scalar values associated to pruned regions of the fields, in order to reflect the contour trees simplification.
3. Sort the mesh vertices according to the updated scalars.
4. Construct two join trees, JT^t for CT^t and JT_1^{t+1} for CT^{t+1}, with labeled edges, using our own implementation of the step described in 3.1.2.
5. Construct another join tree, JT_2^{t+1} for CT^{t+1}, having its edges labeled with edges of CT^t, using our implementation of the step described in 3.1.2.
6. Label the contours of CT^{t+1} with edges of CT^t, as described in 3.1.2, to obtain $CTUpper^{t+1}$.
7. Reverse the traverse order of the mesh vertices, and repeat steps 4-6 to obtain $CTLower^{t+1}$.
8. Intersect the contour labeling of $CTUpper^{t+1}$ and $CTLower^{t+1}$, as described in 3.1.2, in order to obtain the actual contour correspondence information, reflecting significant overlaps between both upper and lower objects.

Note that the join and split trees are computed twice for each field: once in step 2 (done by *libtourtre*) and once in step 4. Their total recomputation in step 4 is required because of the contour tree simplifications happening in step 2.

Moreover, in our implementation for the construction of upper object trees in steps 4 and 5 we had to accord special consideration to the pruned regions. This was because these trees produced using the processes described in 3.1.2 and 3.1.2 were expected to have nodes for the same critical vertices as the pruned version of the contour tree. Therefore, in addition to producing upper object trees compatible to the pruned field, we also required them to be compatible to the pruned contour tree. Otherwise, the rest of the operations for computing the contour correspondences would not have been well defined. When we flatten the field according to the pruning, the scalar of a pruned mesh vertex will be set to the value of the hierarchically closest node remaining in the tree. This means that we will now have connected mesh regions with the same value. Among these, only one voxel will have a correspondent node in the contour tree, and we would ideally like the same to happen in the upper objects trees. Since we sort the vertices by value prior to building trees, the behaviour of the sort function will influence the tree structure corresponding to the mesh region, as explained in Figure 3.7. The Figure contains a mesh laid out according to the vertices values, with nodes 2, 3 and 4 having the same scalar. Different valid ordered sequences can lead to the creation of different contour trees. $CT1$ is produced if the nodes are sorted in the order 1, 5, 2, 4, 3, while $CT2$ is produced by the ordering 1, 5, 3, 2, 4.

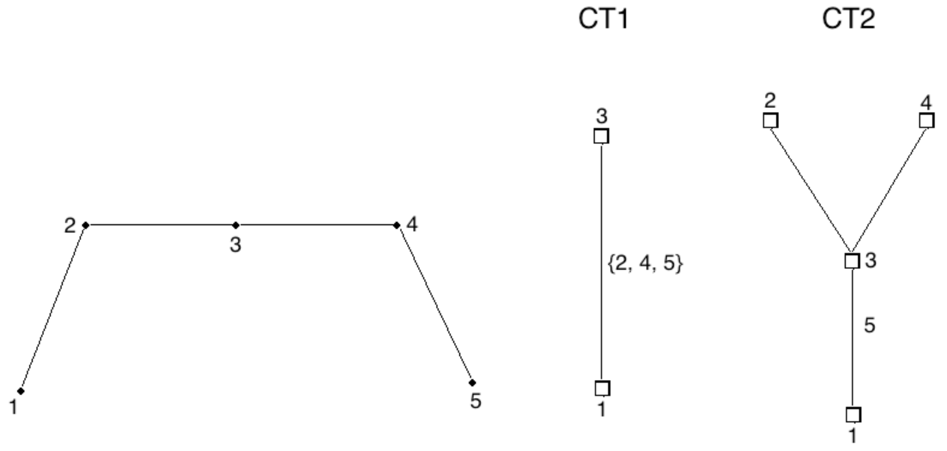


FIGURE 3.7: Possible contour trees built from a mesh with equal values, as a result of pruning.

By considering additional ordering criteria that would make the sort output predictable, we realized that, in general, there is no characterization for the orderings that would produce the expected upper object trees by just following the construction algorithms as described earlier in the chapter.

The first alternative we considered was to compute a new contour tree, according to the new join and split trees. Naturally, the contour tree would be compatible with the upper object trees, and it would reflect the pruned scalar field. However, we are unable to control its structure, and partially lose the value obtained by pruning. As an example, depending on the ordering used, the subtree corresponding to an upper leaf in the first pruned contour tree could now consist of a more complicated structure with upper leaves and joins, all happening at the isovalue of the pruned region. We aimed to avoid this kind of artificial clutter in the structure of the contour tree.

We decided to adapt the algorithms for building upper object trees to treat the pruned nodes differently. This way, during the flattening process, we stored for each pruned voxel the contour tree node it was collapsed into (the node that also defined its post-pruning value). While traversing the ordered mesh vertices in order to build the join and split trees, we use this information as follows:

- When we reach a vertex v, where other vertices were pruned, add these as well to the object v belongs to.
- If the current vertex was pruned, ignore it and continue with the next vertex.
- The management of the collisions, done in step 5, is slightly changed: when vertex v is processed, it may now be the case that more than one collision are now created or updated. This is because multiple vertices are added, at once, which may belong to different objects in the other time step.

Steps 4 and 5 require the usage of union-find data structures. They were implemented as tree forests, with the *path compression* and *union by rank* heuristics, as described in 2.7. This way, the complexity of the operations involving the union-find structures in steps 4 and 5 is $O(v\alpha(v))$, where v is the number of field voxels.

In step 5, updating the collisions between objects after a vertex is processed is one of the expensive parts of the algorithm. Therefore, we carefully considered how the collisions should be stored. We decided to store them in two hashmaps, one indexed by the objects in the first time step, and one indexed by the ones in the second time step. The downside of this is that the information is duplicated, so each update has to be performed twice. On the other hand, this enables to iterate efficiently through all the overlaps of an object, regardless what time step it belongs to.

The time complexity of step 1, the construction of the contour trees, is $\mathcal{O}(v \log v + e\alpha(e))$, with v being the number of mesh voxels, and e the number of mesh cells. The step 2 is performed in time linear to the size of the mesh, while the sorting in step 3 has complexity $\mathcal{O}(v \log v)$. Step 4, the construction of the upper object trees, has complexity $\mathcal{O}(e\alpha(e))$. Step 5, where we also have to keep track of the collisions between objects, has complexity $\mathcal{O}((v+e)(\alpha(e)+n))$, where n denotes the number of contour tree nodes. Step 6 has complexity $\mathcal{O}(nv)$, as v is the upper bound for the number of regions with different labels, while n is the limit for the size of the label sets. The intersection between the two labeled contour trees, in step 7, has again complexity $\mathcal{O}(nv)$ since the regions are already sorted prior to computing the intersections.

3.2 Computing arc correspondences

Once the correspondences between contours are computed, the next step is to aggregate these into arc correspondences. This is a non trivial task, as a consequence of the following:

- The unsteady nature of the scalar values in fields originating from real or simulated environments. This means that an arc of a contour tree will not span the same scalar range as its correspondent from the next contour tree.
- The effects of the simplification process on the two trees. Pruning by volume, as an example, can lead to the sudden appearance of arcs corresponding to significantly large objects, that had been pruned in the previous snapshots. Simplifying to achieve a fixed contour tree size can lead to even more unpredictable results: when the pruning priorities of nodes are close, we may see important difference in the pruned trees across time steps, even when the actual changes in the values of the field are small.

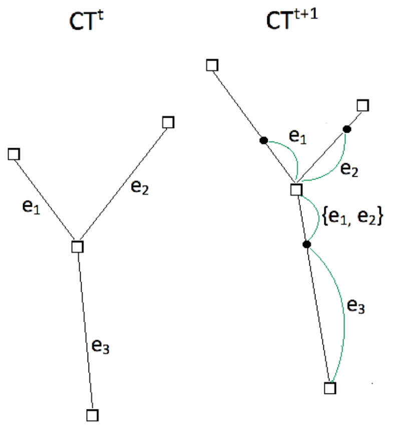


FIGURE 3.8: Possible contour correspondences between two contour trees.

Figure 3.8 shows how the two contour trees have a similar structure, although the scalar values of the nodes are modified. The edge correspondences expected in this example are the intuitive ones. We can see how the $\{e_1, e_2\}$ label on the root arc could mislead a naive approach to assign all $\{e_1, e_2, e_3\}$ to it. In Figure 3.9 we can see how the pruning can determine the sudden appearance of two upper leaves from e_2 in the second time step. We would like to conclude that all the three edges (the two upper leaves and their joining arc) correspond to the initial e_2.

The above examples suggest that obtaining the edge correspondences for an arc is difficult if we only use the arc's own contour correspondences information. Therefore, we realized that also considering the contour correspondences of other relevant edges will be required in order to obtain a robust labeling algorithm.

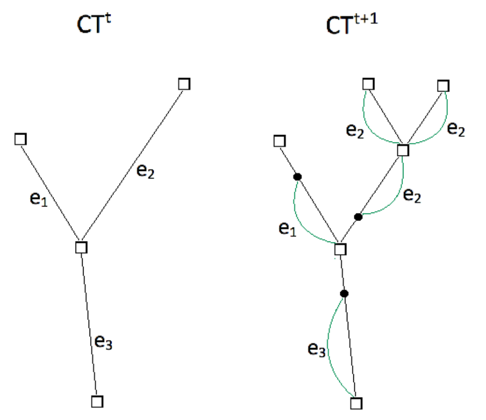


FIGURE 3.9: Possible contour correspondences between two contour trees.

We decided to use the nesting relationship of the contour tree features in order to compute the correspondences between arcs. For each arc e^{t+1} of the second contour tree which has a contour labeled with an edge e^t of the first contour tree, we are going to assign the e^t label to e^{t+1} only if the label sets in the leaves of their upper trees coincide. Therefore, we combine the overlap knowledge given by the contour correspondence with the topology information given by the contour tree structure.

The algorithm we propose for obtaining the arc correspondences between two contour trees, CT^t and CT^{t+1}, can be summarized as follows:

1. For each arc e^t of CT^t, compute $leaves^t(e^t)$, representing the set of upper leaf arcs in the upper subtree of e^t. If e^t is an upper leaf arc, $leaves^t(e^t) = \{e^t\}$.

2. Sort the nodes of CT^{t+1} by their scalar value.

3. Traverse the nodes in decreasing order by value. For the current node n, traverse its down arcs and compute their correspondents from CT^t, as described in the next steps.

4. For a fixed arc e^{t+1}, compute the set $leaves^{t+1}(e^{t+1})$ by appending the similar sets of the adjacent up arcs.

5. For e^{t+1}, consider the set E_1 containing the edges of CT^t with contours that evolve into e^{t+1}. We consider the upper leaf arcs and non-upper leaf arcs of E_1 separately. For upper leaf arcs e^t, we say that e^{t+1} corresponds to e^t if $leaves^{t+1}(e^{t+1}) \subseteq \{e^t\}$. After processing all the upper leaf arcs, we augment the set $leaves^{t+1}(e^{t+1})$ with the obtained correspondences from CT^t, and then proceed by considering the non-upper leaf arcs of the set. For arc e^t, we now say that e^{t+1} corresponds to e_1 if $leaves^{t+1}(e^{t+1}) = leaves^t(e^t)$.

The algorithm is able to compute the desired correspondences in the two examples given earlier in this section. It can be noticed that we are not using the size of the contour correspondence range between two arcs, but only the fact that at least one contour corresponds. We chose to do this, as the length of a scalar range on an arc is not generally correlated with the size of the associated mesh region. Therefore, it cannot be reliably used as a good measurement for the significance of spatial overlap.

3.3 Summary

This chapter presented the feature tracking algorithm developed as part of this project. We described how the algorithm for computing correspondences between contours was implemented, and how it was improved to support the prior application of contour tree simplification techniques. Then, we focused on how the contour correspondences were translated to correspondences between arcs.

In Chapter 4, we will present a tool that uses this algorithm in order to track features visually in scalar fields. In Chapter 5, we will evaluate the effectiveness of the algorithm against test cases of increasing difficulty.

Chapter 4

Tool implementation

As part of the project, we aimed to develop a novel way in which researchers are able to understand time dependent scalar fields, motivated by examples from aeronautics fluid flow simulations. We decided to wrap our feature tracking algorithm in a software tool, that would integrate the visualization of the features of such fields with their contour tree representation.

4.1 Functionality overview

The graphical interface of the tool, illustrated in Figure 4.1, provides the following functionality:

- A visualization of the contour tree of a field, supporting the process of understanding its topology.
- The ability to interactively select subtrees of the contour tree, and visualize their representative local isosurfaces. This way, a researcher is able to observe the field's interesting features, by using the contour tree as an index into the data. Multiple contour tree regions can be selected at once: they are assigned tags of different colors, visible both in the contour tree and in the field visualization component.
- Feature tracking capabilities. Using the keyboard arrows, we are able to move forwards or backwards in time throughout the loaded field, while the tags are updated to reflect the correspondence between features of the consecutive time steps.
- Functionality improving the data analysis experience, such as displaying the value of a contour tree node while hovering over it, or the ability to change the aspect ratio of the contour tree layout.

4.2 Data manipulation pipeline

The main steps we perform before being able to obtain the above visualization can be summarized as follows:

1. Converting the raw output of PyFR into an appropriate input for our tool.

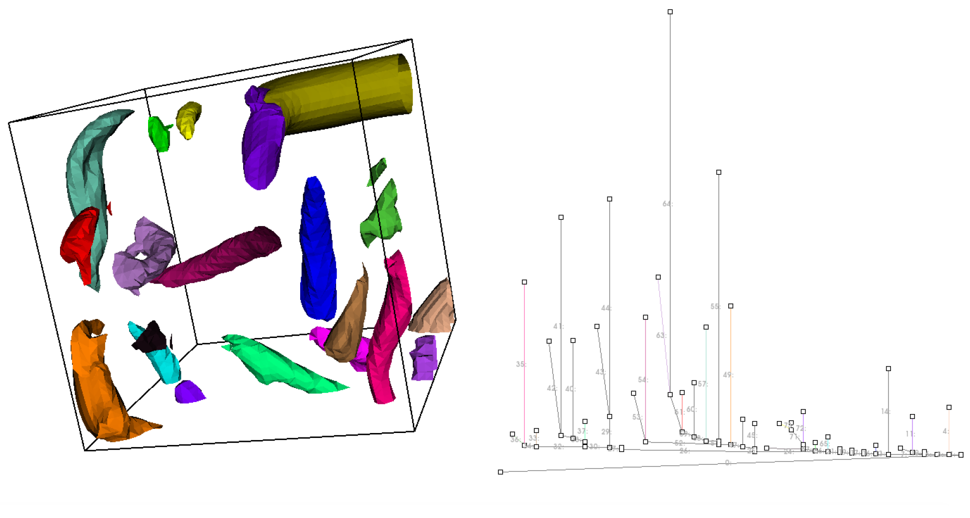


FIGURE 4.1: Graphical interface of the tool

2. Building the contour trees for all the temporal snapshots.

3. Simplifying the contour trees.

4. Using the structure of the simplified contour to build the local grids that will be used to render local isosurfaces.

5. Computing the feature tracking information as described in the previous chapter.

6. Optionally, running a strategy for automatically identifying features of interest in the start field.

7. Launching the user interface described in the previous section.

The rest of the section aims to give further details about each of the above steps.

4.2.1 Preprocessing the data

The raw output of a PyFR simulation is represented by a sequence of *.pyfrm* and *.pyfrs* files, corresponding to a sequence of snapshots in time. In order to be able to manipulate them efficiently, we firstly export them in *.vtu* files, a standard format for unstructured meshes. This is achieved using the Python export script provided by PyFR.

Our tool was designed to be generally usable for any sequence of meshes that have a scalar field associated, regardless the semantics of the scalar. However, throughout the project we used the Q-criteria for testing and assessing the efficiency of the product. The Q-criteria is not included in the output of PyFR, but it can be easily calculated from the gradients of the velocity field. We performed this using a Paraview *Calculator* filter, and added it as an additional scalar in the *.vtu* files.

A further preprocessing step consisted of applying a *Clean to Grid* filter, available in Paraview as well, that improves the quality of the grid by solving potential connectivity issues in the output of PyFR. For example, this filter unifies the grid vertices that share the same position in space.

These data preprocessing steps have to be done before starting the tool, which assumes the input files are in the *.vtu* format, clean and with no further field values calculation required. We decided to keep these preprocessing steps out of the main program, due to a number of reasons:

- Performing them inside the tool would restrict its generality. We would be bounded to the PyFR file format, so the tool would lose its value in the context of other CFD solvers; in contrast, *.vtu* is a much more general format. We would also be bounded to use the Q-criteria for analysis.
- Calculating fields such as the Q-criteria is done using methods well-known by the community, so including this step in the tool would not produce any extra value.
- Depending on the size of the simulations, the preprocessing steps may take very long to compute. This is clearly undesirable for a data analysis tool.

Note that we used Paraview's Python scripting capabilities in order to automate the data preprocessing step, since doing them using the Paraview graphical interface would have been clearly unfeasible for simulations consisting of hundreds of snapshots.

4.2.2 Building the contour trees for feature extraction

We proceed by loading the unstructured grids in memory, each having a number of scalar fields associated. In order to save memory, we filter out the fields that are not important for the analysis. The relevant fields, as well as the one used for building the contour trees, have to be passed as command line arguments. Note that we may be interested in storing more than one field in order to combine the analysis of multiple fields, as we demonstrate in Chapter 6.

We continue by building the contour trees using the *libtourtre* library. The interaction with the library is done by just passing the following information:

- A function that takes a vertex as argument, and returns a list containing the mesh neighbours of the vertex. In order to implement this efficiently, we have to build the connectivity graph of the mesh beforehand, and store it as adjacency lists. Otherwise, the VTK format for unstructured grids is not indexing edges by nodes, so retrieving the neighbours of a vertex would require to traverse all the graph edges.
- A list containing the mesh vertices in ascending order by value. Therefore, it is our responsibility to resolve any value equality in the field.

At this stage, we accepted any ascending ordering offered by the sort function we used, since any ordering would produce a valid contour tree.

The contour trees built for large-scale data are huge at this point. Therefore, we continue by applying certain simplification techniques to reduce their size. Since we usually used the Q-criteria scalar, we massively pruned the bottom part of the tree that contained negative values, as they represented noise. Also, we generally pruned by volume, with various thresholds, in order to filter insignificant field features.

As mentioned before, the tool is able to render local isosurfaces, corresponding to subtrees of the contour tree. This is achieved using VTK *Contour Filters*, which are able to draw fixed isovalue contours for a grid passed as a parameter. We could obtain a global isosurface of the field at a fixed value by contouring the entire grid that we read from a file. However, in order to generate local isosurfaces, which correspond to individual features, the idea is to use a Contour Filter on *subregions* of the initial grid. For each contour tree arc, we generate a grid subregion, which contains all the cells adjacent to at least one point on that arc. Remember that each mesh vertex is associated to exactly one contour tree arc, which contains a contour that passes through the vertex. We therefore use this *vertex-arc mapping* obtained from *libtourtre*, and updated during the pruning process, to build the local grid for each arc. In order to obtain the subregion corresponding to a contour tree subtree, we are using a VTK *Append Filter*, which appends the grids corresponding to the arcs belonging to the subtree.

To perform these steps, we adapted the code developed by Daniel Simig in [38]. His document contains more details about the interactions with *libtourtre* and the tree simplification process, as well as an attempt to mediate the memory usage overhead caused by the storage of the arcs local grids.

4.2.3 Detecting features of interest

Step 6 of the data manipulation pipeline refers to the process of preselecting the field features that have certain properties. As a basic example, in many scenarios included in the Evaluation chapter, we wanted to preselect the elementary vortex features corresponding to the upper leaves of the tree. On the other hand, in Chapter 6, we will see an example of how the tool can be extended with more advanced filters, that look at other properties of features, such as their shape or position within the field.

The initial version of the tool only includes upper leaves filtering, spheres filtering (described in the end of the current section), and streamwise low-velocity streak vortices filtering (addressed in Chapter 6). However, adding more filters in the future, according to the specific needs of researchers, can be done by just providing new implementations of the *FeatureFilter* abstract class. Note that this step is optional: the tool can be launched without any preselected feature, and the user can proceed by finding the features of interest manually.

Filtering spheres

Although a spherical shape is not typical for vortex structures existing in fields from aeronautical simulations, we decided to implement this filter for demonstration purposes. It is going to be used later, in the Evaluation chapter.

In [43], Wadell et al. introduce the *sphericity* metric, a scalar defining how close the shape of an object is to a mathematically perfect sphere. The sphericity, Ψ, of an object, is defined as follows:

$$\Psi = \frac{\pi^{\frac{1}{3}}(6V_p)^{\frac{2}{3}}}{A_p}$$

where V_p and A_p represent the volume and the area, respectively, of the inspected object.

In our implementation, we used a VTK *vtkMassProperties* calculator to compute the required properties (i.e. volume and surface area) of the features we extracted from the contour tree. In order to classify objects as spheres and non-spheres, we established a threshold $T_{sphericity}$ for the sphericity metric. All features with the sphericity greater than the threshold were then considered to be close enough to the shape of a perfect sphere. In order to select the sphere-like features in Section 5.1, we were able to set a high value, 0.9, for $T_{sphericity}$.

4.3 The graphical interface

The graphical interface was realized using the built-in capabilities of VTK, and it contains two major components: the field where a number of isosurfaces are displayed, and the contour tree of the field. Note that the contour tree is laid out such that the nodes are vertically following a linear scale, using the approach described in [38]. This layout strategy guarantees that, in contour trees without splits, there will be no crossing between arcs.

As mentioned before, tagging can be done manually or by the initial filtering. Each tag will correspond to a different color, but multiple features can have the same tag: this happens when selecting non-upper leaf nodes, or when splits happen during the tracking process. The colors assigned to the tags were imported from a dictionary of distinguishable colors, available at [40]. The dictionary consists of a total of 191 colors. If the number of tags is larger than this number, the tool is going to assign random colors to as few tags as possible.

Only the tagged features will be displayed as isosurfaces in the field. The isovalue chosen for the contour of an arc is computed by a simple linear interpolation formula: $0.9 * LOW + 0.1 * AVG$, where LOW represents the value of the bottom node of the arc and AVG denotes the average of the values of voxel associated to the arc. Therefore, the isovalue is close to the bottom node, where the contours are usually more pronounced. On the other hand, this formula may lead to choosing an isovalue out of the value range of an arc, when the tree structure above the arc has been pruned. If this happens, we contour at the isovalue $0.9 * LOW + 0.1 * HI$ (where HI is the value of the upper node of the arc) instead, which is guaranteed to be within the range. The reason for not using this as standard is that the movement of pruned features can significantly change the highest value of the arc they are pruned into. Therefore, the appearance of the isosurface displayed for a tracked object could change too much relatively to the actual position and values change suffered by the object.

The tool allows the tracking of the tagged features along the temporal axis. When moving from a time step to the next one, we are going to use the correspondence information in order to propagate the current tags: if tag t corresponds to the set of edges E, when we switch to the next time step we will assign t to all edges that correspond to edges in E. In this way, the tool is able to track features of interest as they move, split, or join with other features.

4.4 Summary

This chapter presented the visualization tool for time-varying fields that was implemented as part of this project. The focus was, alternatively, on the graphical interface and its functionality, as well as on the pipeline of events happening before the graphical interface is launched. In Chapter 5, we will evaluate the tool, looking at both its correctness and its functionality.

Chapter 5

Evaluation

In this chapter we aim to demonstrate the correctness of the feature tracking tool developed, by running it on examples of increasing difficulty. Later, in Sections 5.3 and 5.4, we further analyse the feature tracking algorithm, while in Section 5.5 we briefly compare it with similar techniques existing in the literature. Lastly, Section 5.6 performs a succinct evaluation of the tool.

5.1 Tracking features in artificial fields

In this section we apply the developed tool on small scale data, in order to prove the correctness of the feature tracking algorithm against test cases with known correct behaviour.

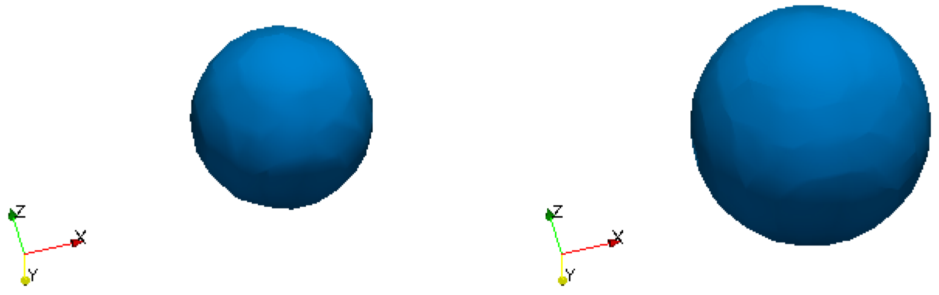
5.1.1 Generating small scale meshes with features

In order to evaluate the expected behaviour of the software on data of smaller scale, we proceeded by generating 3D meshes with artificially introduced geometrical shapes that would imitate the vortex structures usually observed in fluid dynamics flows.

The trivial method used for obtaining smaller but still highly relevant 3D meshes was to extract subregions of two existing simulations: the Taylor Green Vortex and the plane Couette flow, which are described in Appendix A. The subregions were obtained by clipping these meshes using certain bounds to produce rectangular boxes having the desired dimensions.

The vertices in the obtained meshes were then assigned small random negative scalar values: this was to reproduce the case of fluid flows, where regions with no vortices are characterised by negative Q values, which do not have any physical meaning. Therefore, we have now obtained 3D meshes full of noise. We are now going to introduce different shapes defined by positive scalar values, that will correspond to features in the scalar field:

- Spheres. They are characterised by an (x, y, z) origin, a radius r and a scalar range (min, max). We are going to traverse all mesh vertices within distance r of origin, and assign them a scalar which is inverse proportional to the distance, normalized in the range (min, max). This ensures that the vertex closest to (x, y, z) will have the maximum value (still lower than max) compared to the rest of the sphere, and it will correspond to an upper leaf in the contour tree.



(A) Isovalue 3.5 (B) Isovalue 2.2

FIGURE 5.1: Isosurfaces of a field containing a sphere feature, taken at two different isovalues. Entire sphere value range: [0.0, 4.0]

A sphere produced using this method can be seen in Figures 5.1a and 5.1b. The figures were produced using Paraview contours at two different isovalues. However, very similar isosurfaces would have been obtained using our software, as we are going to see below. One thing to notice is the larger size of the sphere when it is plotted at the lower isovalue, due to the fact that the radius of the spherical contour has increased.

- Bananas. We mathematically modeled them to be sections of toruses. The standard shape of a torus is included in Figure 5.2. A torus around the origin and the z-axis of the 3D Euclidean space is defined by the following equation:

$$z^2 \leq r^2 - (R - \sqrt{x^2 + y^2})^2$$

Here r is the radius of the torus body, whereas R is the distance from the centre of the torus body to the centre of the shape. Both r and R are parameters of the banana, along with the central point (the origin) of the shape. Note that the equation also includes the points inside the shape, not only the border. Similarly to the spheres case, we traversed the mesh vertices, and, for a vertex (x, y, z), we computed the (z_1, z_2) range of z coordinates belonging to the torus for the fixed x and y, by solving the above equation. Then, if z was in the range, we would assign a positive scalar to the vertex, inverse proportionally to the value $z - z_1$ in order to obtain the expected contour behaviour. Since the equation assumes the origin is $(0, 0, 0)$ we also had to translate each (x, y, z) by the fixed origin of the banana before doing the test. In order to obtain the banana shape, we partially discarded the points of the banana that had the y coordinate greater that a certain value. A banana obtained this way can be seen in Figures 5.3a and 5.3b. These were again produced using Paraview, at two different isovalues.

Note that in order to obtain rotated bananas, one additional step has to be done: after translating (x, y, z) to be relative to $(0, 0, 0)$, we will also

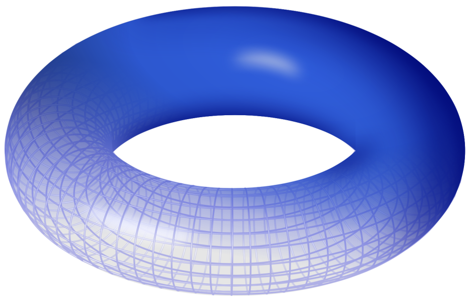
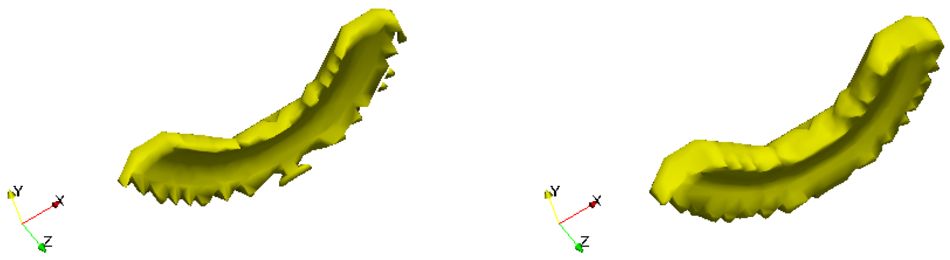


FIGURE 5.2: Shape of a 3D torus



(A) Isovalue 2.5 (B) Isovalue 1.5

FIGURE 5.3: Isosurfaces of a field containing a banana feature, taken at two different isovalues. Entire banana value range: [0.0, 4.0]

multiply the point with a rotation matrix.

5.1.2 Tracking moving spheres

Figure 5.4 shows a scalar field containing two spheres, and its corresponding contour tree, which is laid out such that the vertical positions of the nodes follow a linear axis, according to their values. The picture was taken using the tool developed as part of this project, and manually annotated with the isovalues marking significant events for the purpose of this explanation. The isovalue used for contouring is 0.4, with the value range of the spheres being again [0.0, 4.0]. Note that for isovalues greater than 4.0 no isosurface would have been visible. This is also the case for isovalues lower than the minimum noise value. In contrast, for an isovalue in the range of the contour tree root arc, we can distinguish both spheres, but also a significant amount of noise as well, as seen in Figure 5.5 produced using Paraview. An interesting aspect is why the spheres are still distinguishable, even when taking isosurfaces at values out of the range used during their construction. This happens because of the existance of certain cells in the mesh that connect sphere voxels with noise voxels. Since linear interpolation is used to compute the values inside the cells, there will still be sphere-like contours to be rendered at those isovalues.

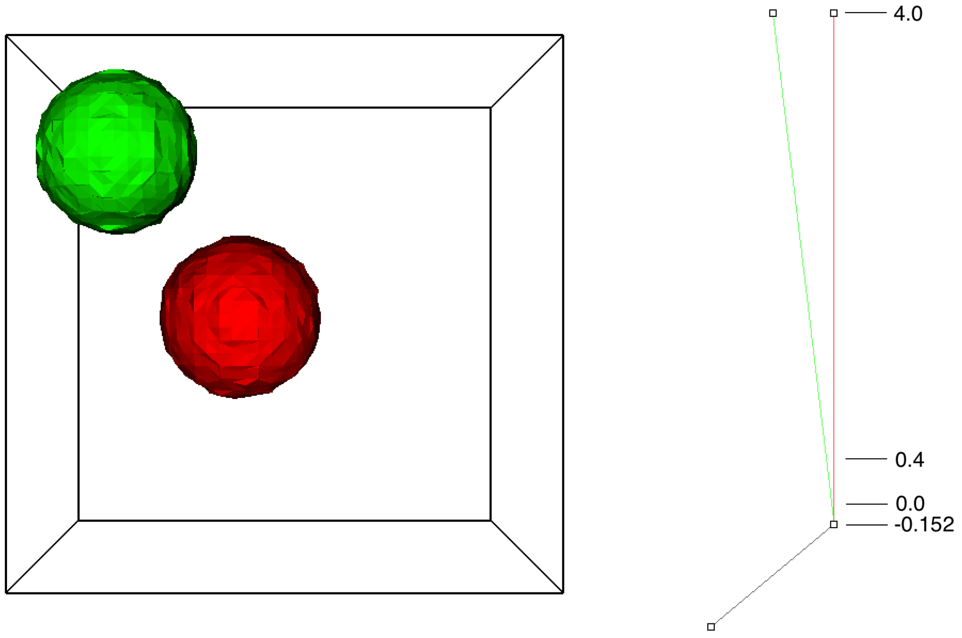


FIGURE 5.4: An isosurface and the contour tree of a scalar field containing two spheres. Isosurface taken at 0.4.

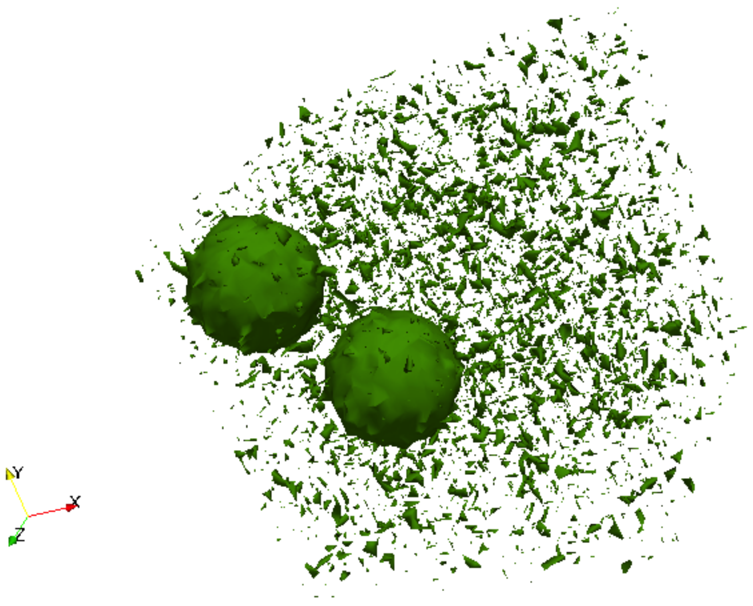


FIGURE 5.5: Contour of the field at isovalue -0.1

The two upper leaves of the contour trees are on the same horizontal line, and represent the creation of the two spheres at the isovalue 4.0, corresponding to the centers of the spheres. Going down, each intersection of a horizontal line with the tree will yield two points, representing particular contours of the spheres. As the horizontal line isovalue is decreased, the sphere-like contours are growing. We can see that they eventually join at isovalue -0.152. This happens slightly below 0 since the two spheres will join using mesh vertices associated to noise scalars, as they otherwise do not share any vertex. All the voxels with scalar values lower than 0.0 belong to the root edge. Although this encapsulates a complex subtree structure built during the initial contour tree construction, it was reduced to a single edge during the pruning step. This is to avoid cluttering the contour tree structure with complex subtree structures corresponding to noise.

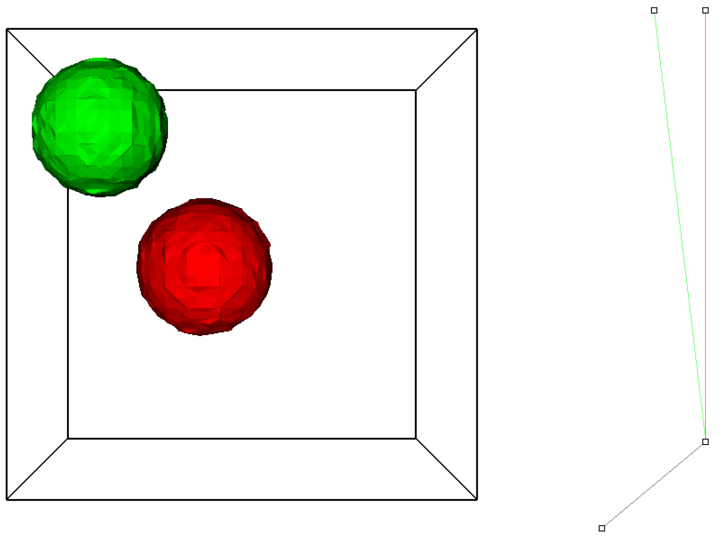
Figure 5.6 highlights the time evolution of the scalar field over four snapshots. The two spheres are only moving in the positive x direction, and the contour tree remains the same. The scenario is testing the ability to track two moving objects which do not overlap or split. Here the difficulty was to propagate the green and red tags assigned to the two initial structures. In the second step we see that the order of the two tagged edges is changing. This happens since the contour tree horizontal layout is arbitrary, and does not account for any correspondence information. We are able to maintain the appropriate tags, as highlighted by the coloring of the spheres and of the contour tree edges.

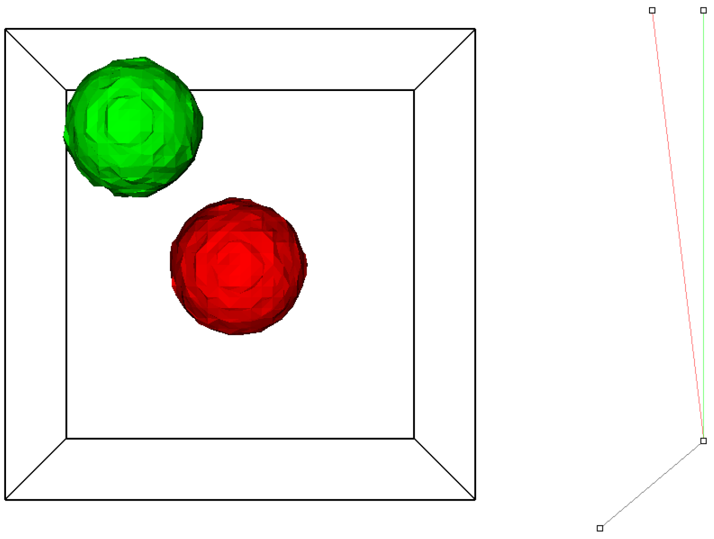
5.1.3 Spheres that join and split

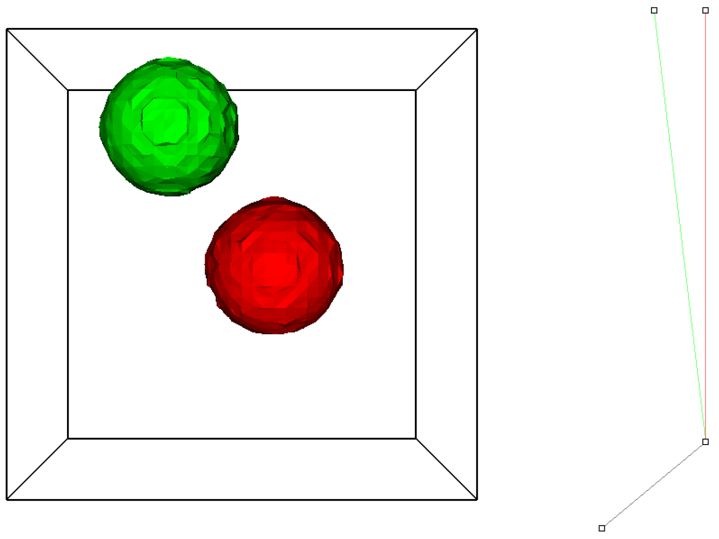
The setup (Figure 5.7) is similar to the previous one, but a third, smaller sphere has been added, with the range of values [0.0, 2.0]. This smaller range is highlighted in the contour tree, by the blue arc starting later than the other two features.

Apart from movement, the algorithm should identify the joins and splits of structures. In Figures 5.7b, 5.7c and 5.7d we can see how the red and blue spheres are getting closer and closer, merging at step 3 in a purple object (the result of mixing blue and red). The initially distinct features are now combined into only one, reflected in the contour tree, by the purple arc. At the next step the two spheres bounce, so they split again, but now both are purple since the initially distinct identity of the objects was lost. Note that the join arc of the two objects is purple now, since it also corresponds to the unique arc representing the two features at the previous step.

The simulation can also be done backwards. At step 4, we can assign color tags to the features defined by the upper leaves of the tree (as we have done for all the examples presented so far). Then we move to the left in the temporal direction. This produces the simulation shown in Figure 5.8. The noticeable thing here is that after the split happening between steps 2 and 3, there will be no join arc that only connects cyan objects, so there will be no cyan join arc.


(A) Step 1


(B) Step 2


(C) Step 3

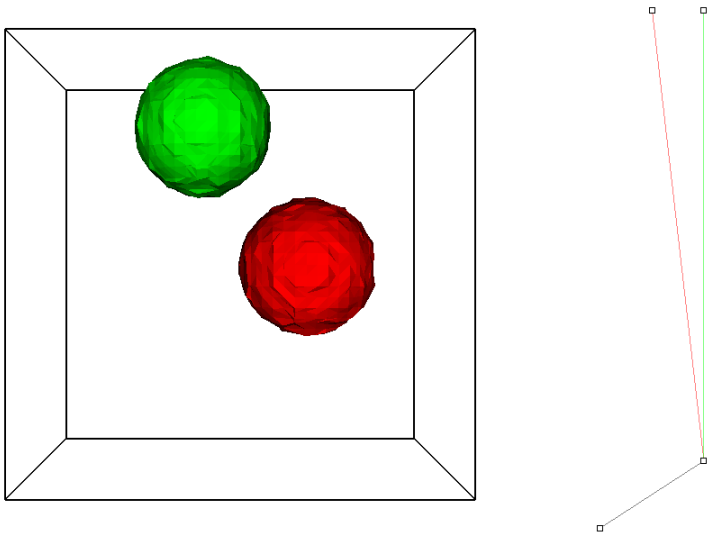
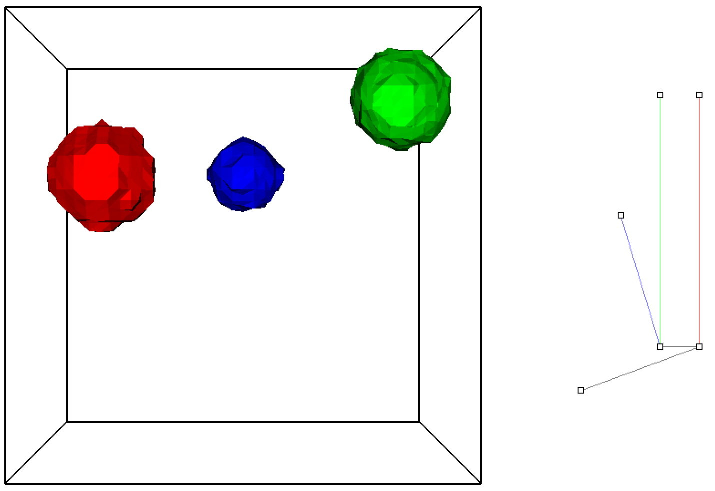
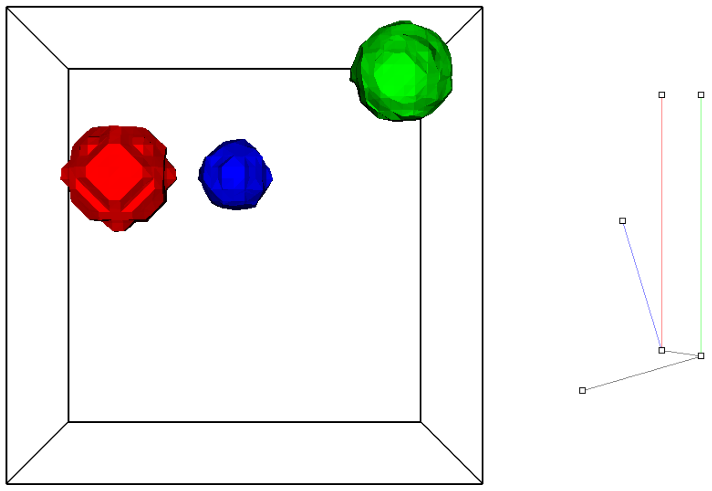
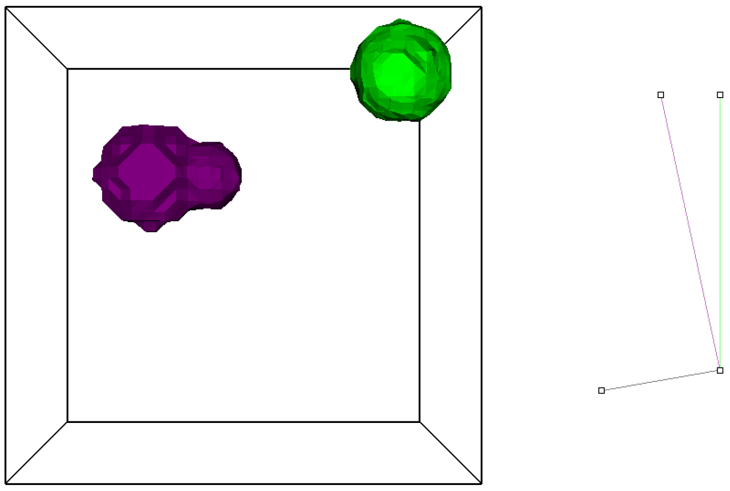

(D) Step 4

FIGURE 5.6: The successful tracking of two spheres moving in the positive x direction.


(A) Step 1


(B) Step 2


(C) Step 3

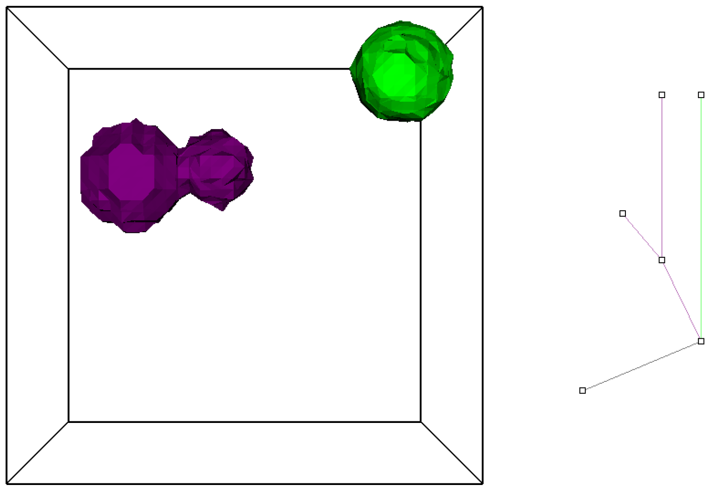
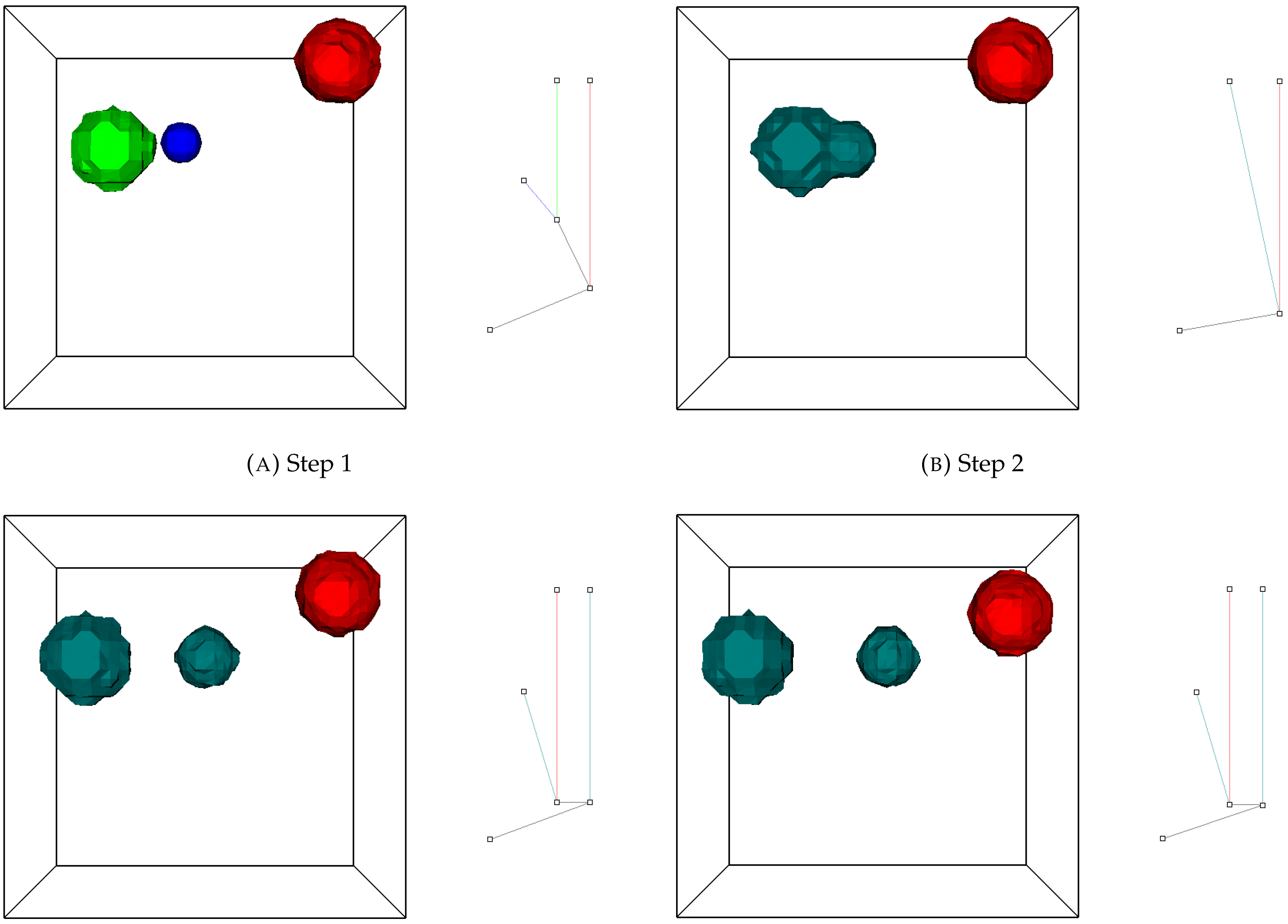

(D) Step 4

FIGURE 5.7: The successful tracking of three spheres that move and interact.



(A) Step 1

(B) Step 2

(C) Step 3

(D) Step 4

FIGURE 5.8: The successful backwards tracking of three spheres that move and interact.

5.1.4 Identifying and tracking a banana

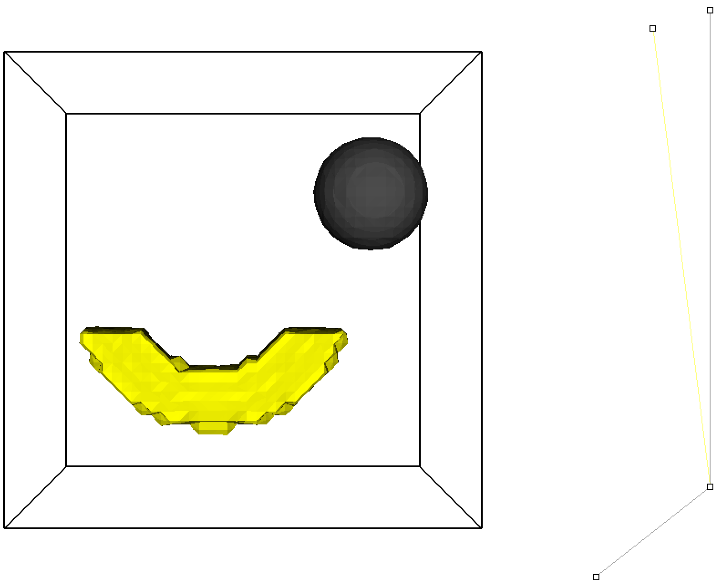
In the current scenario (Figures 5.9) we are looking at one sphere and one banana. The isovalue used for contouring is 1.2. The contour tree structure is the same as in Figures 5.6, since contour trees are not aware of the shapes of the features. This example demonstrates two aspects:

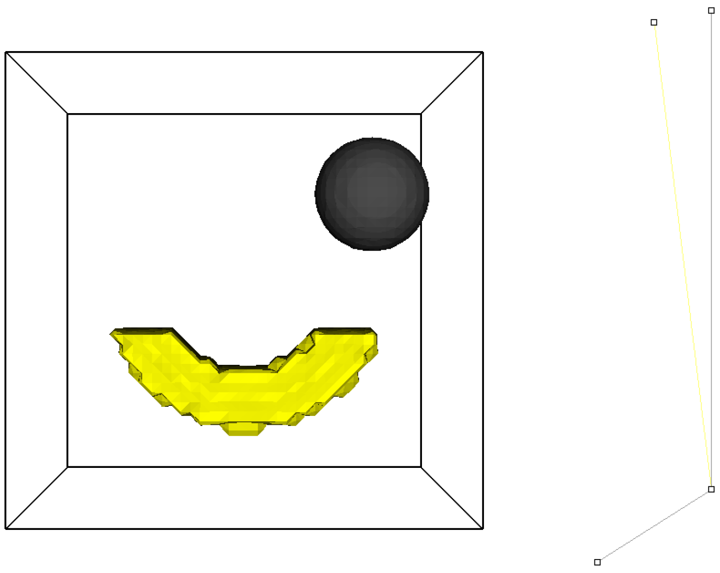
- The tracking algorithm is not limited to tracking spheres.
- We are able to track edges selectively. Here we distinguish between the banana and the sphere by using the method described in 4.2.3, and we assign it the yellow color, while everything else is left in dark gray. We then track the banana over the next three snapshots. We are able to do this without running again the non-sphere identification algorithm.

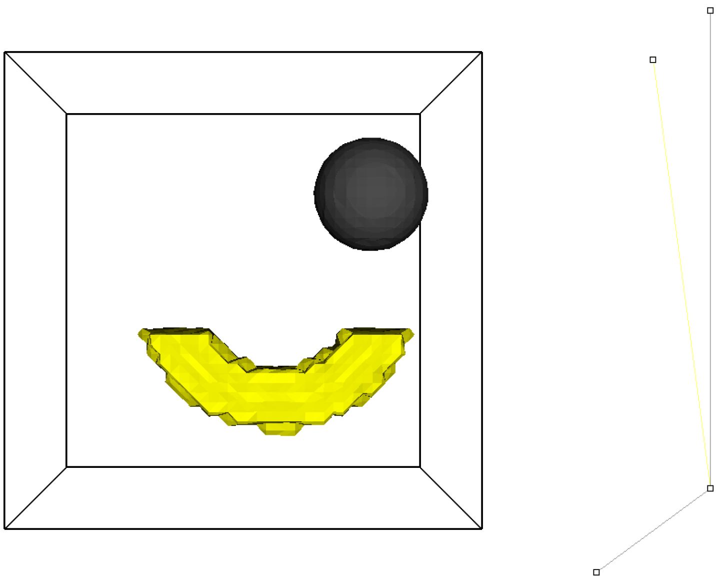
5.1.5 Tracking a banana that rotates

The initial setup in Figure 5.10 is identical to the one in the previous example and we are again able to distinguish the banana from the sphere. However, the feature is now rotating rather than moving. We can see that even if the banana is rotated by 90 degrees along the OZ axis, the algorithm is able to find the correspondence for a certain tolerant threshold. In this particular example, the choice of the threshold was related to the ratio between the radius of the banana and its "length". If we decrease this ratio (so the banana becomes thinner and longer), for the same threshold we are going to consider the overlap insignificant and will lose the tag between the snapshots, as seen in Figure 5.11.

Note again that successfully detecting a correspondence is dependent on the value set for the significance threshold. This means that, for any correspondence found using threshold t_1, there exists another threshold t_2 with $t_2 > t_1$ such that the same correspondence is no longer identified. In the case of Figure 5.10 we used the threshold 0.2, a relatively small threshold in practice, in order to be able to detect the correspondence. However, if we set the threshold to be 0.4, the correspondence is no longer detected, as shown in Figure 5.12.


(A) Step 1


(B) Step 2


(C) Step 3

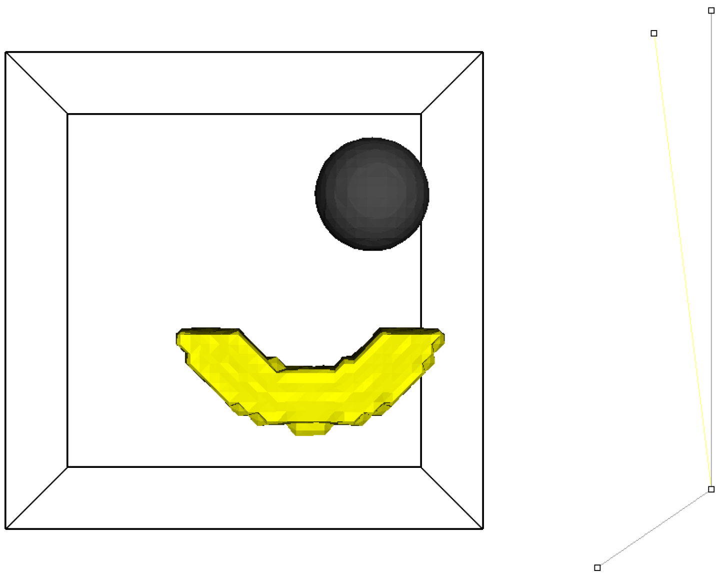

(D) Step 4

FIGURE 5.9: Tracking a banana that moves in the positive x direction.

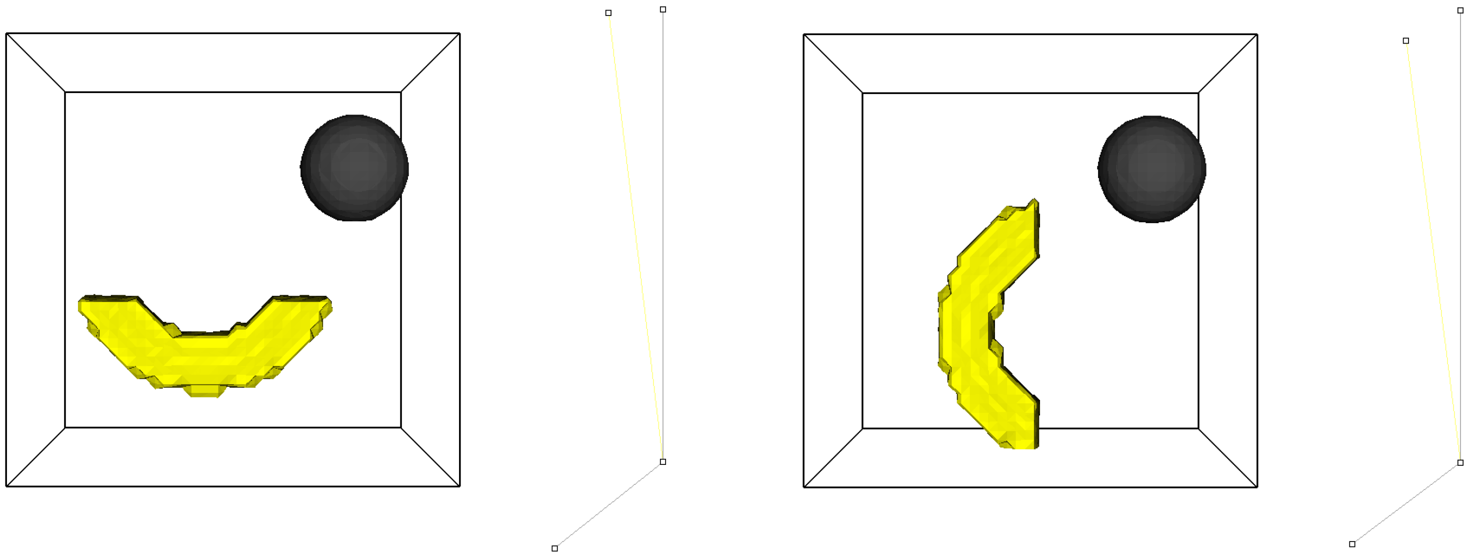

(A) Step 1 (B) Step 2

FIGURE 5.10: Successful tracking a banana that rotates 90° along the OZ axis.

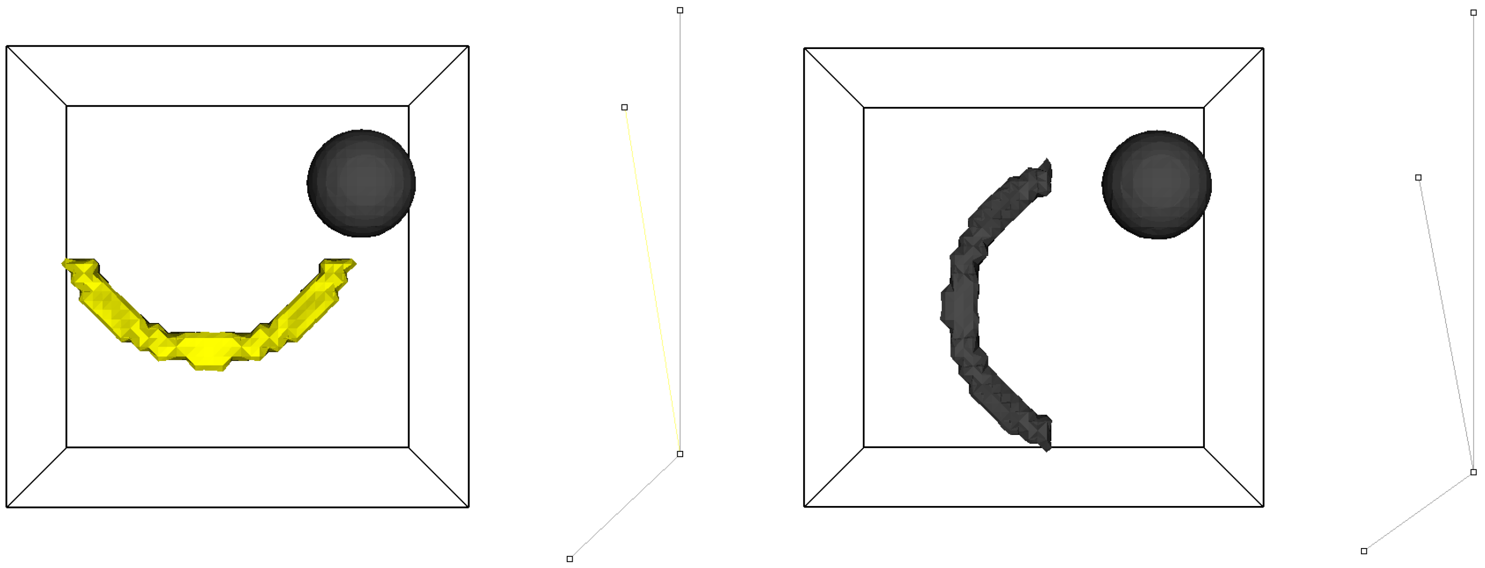
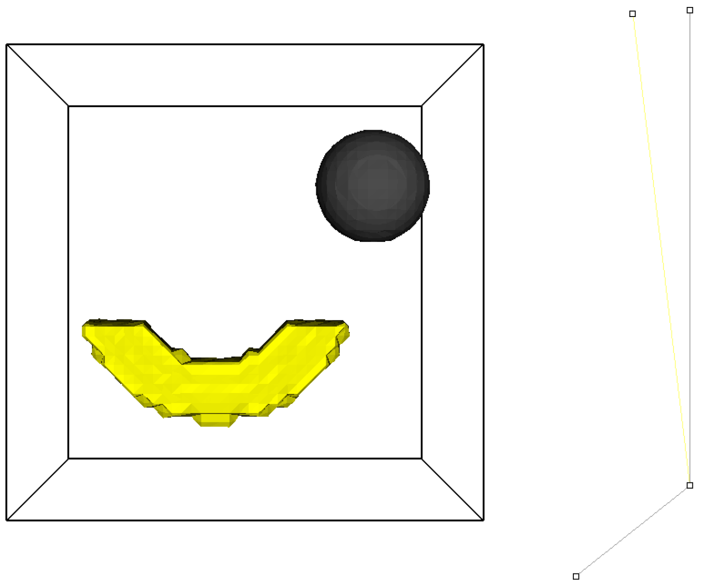

(A) Step 1 (B) Step 2

FIGURE 5.11: Failing to track the banana when the significance threshold is too high compared to the degree of overlap.


(A) Step 1

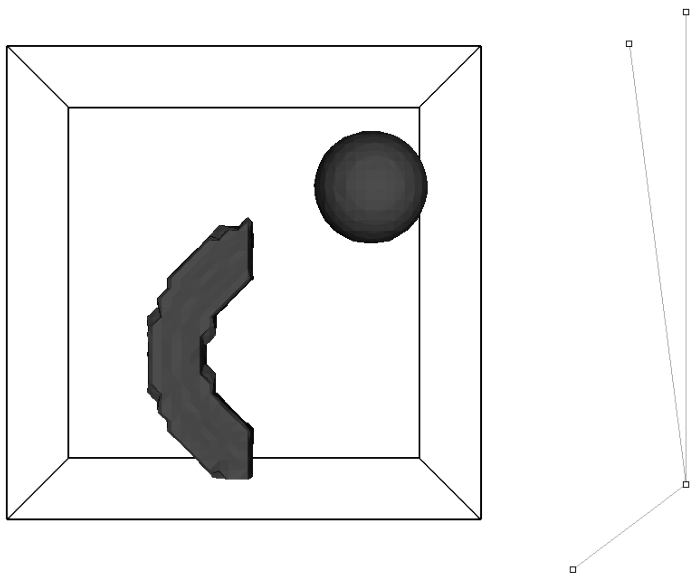

(B) Step 2

FIGURE 5.12: Failing to track a banana when the threshold is too high compared to the degree of overlap.

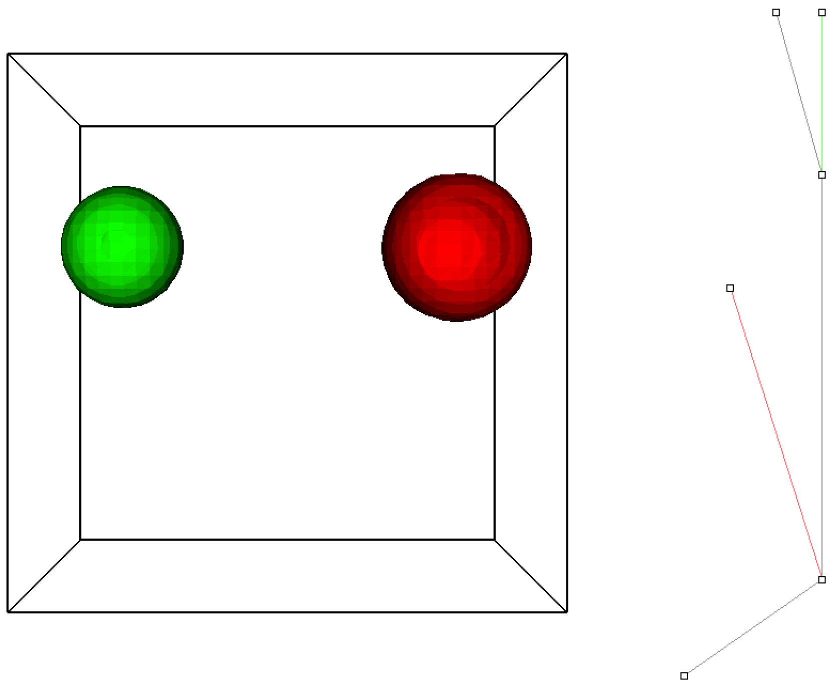


FIGURE 5.13: Spheres plotted at different isovalues

5.1.6 Plotting contours with different isovalues

All the contours plotted so far were taken at unique, global isovalues. However, in the general case this may not be possible: there may be no horizontal line that intersects all the tagged edges of the contour tree. Therefore, we improved the tool with the ability of isocontouring local contexts with different values, such that features at disjoint ranges of values are visible at the same time. This adopts the flexible isosurfaces idea described in [8]. In addition, we are not going to contour every single feature anymore, but only the ones that are tagged. Hence, the gray isocontours will not be visible anymore.

Figure 5.13 contains three spheres:

- One tagged green, with the value range [2.1, 4.0]. The contour shown for this feature is taken at the isovalue 2.7.
- One tagged red, with the value range [0.1, 2.0]. The contour shown for this feature is taken at the isovalue 0.7.
- One untagged. This is not shown since its local context is distinct from the local context of the green sphere, although they have identical isovalue ranges.

5.1.7 Larger example with bananas and spheres

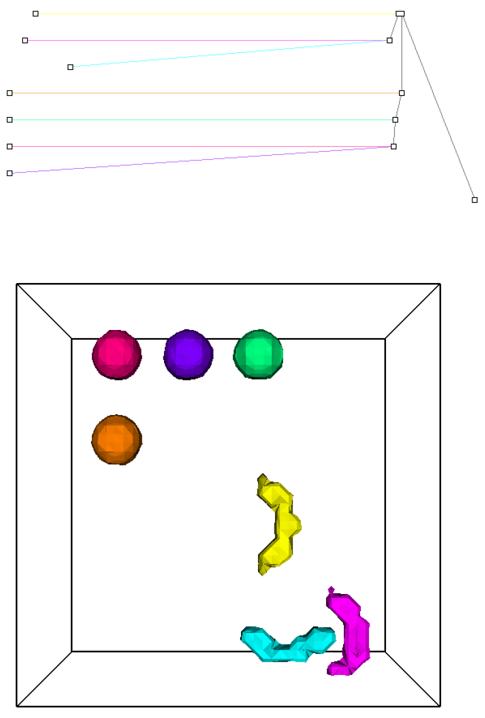
The steps starting with Figure 5.14a correspond to an initial scalar field containing three bananas and four spheres. There are a number of aspects to be highlighted in this scenario:

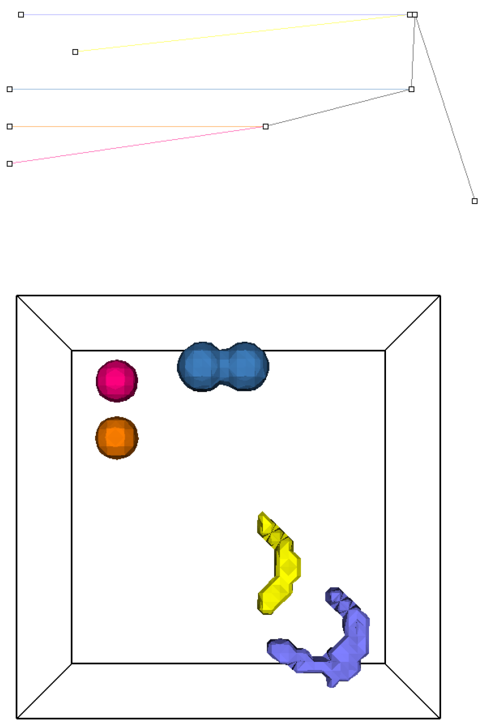
- Detecting joins between features is not shape dependent: in step 2 we can see how the blue and the pink bananas are joining in a bigger structure. In step 3 this structure will join with the yellow banana, resulting in a horseshoe-like feature tagged gray from now on.

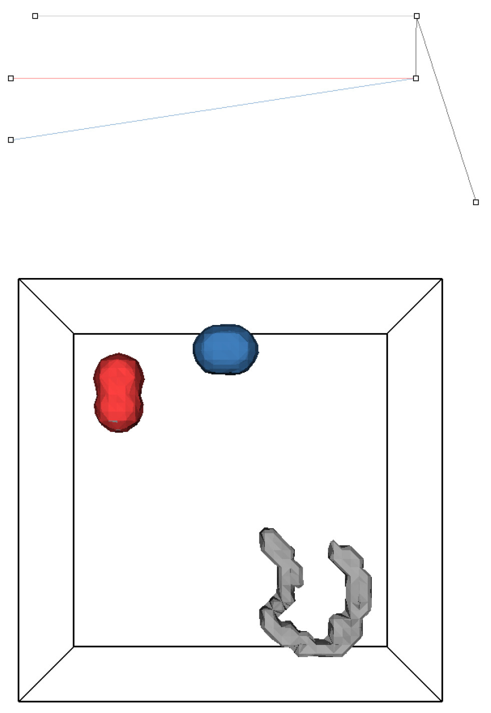
- In step 4, the horseshoe feature is split again in the three initial bananas that are moving away from each other. However, now we also see a certain amount of noise, caused by the fact that the join arc of the bananas is tagged as well. Therefore, a contour will be rendered for it.

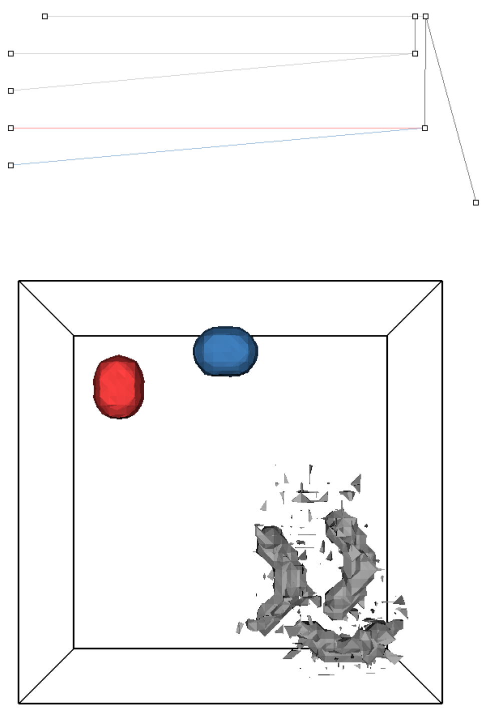
- Two pairs of spheres (the pink and the orange one, and the purple and the green one) are approaching each other and intersecting. In steps 2 and 3 they are merging and considered unique features, fact reflected in the topology of the contour tree. In step 6 these features are splitting, after the spheres passed one through the other. Note that here, since the spheres have the same radius, one passing through the other would be equivalent to bouncing back after joining. Similarly, after the two splits, one red and one blue tagged structures are now approaching and joining in step 8.

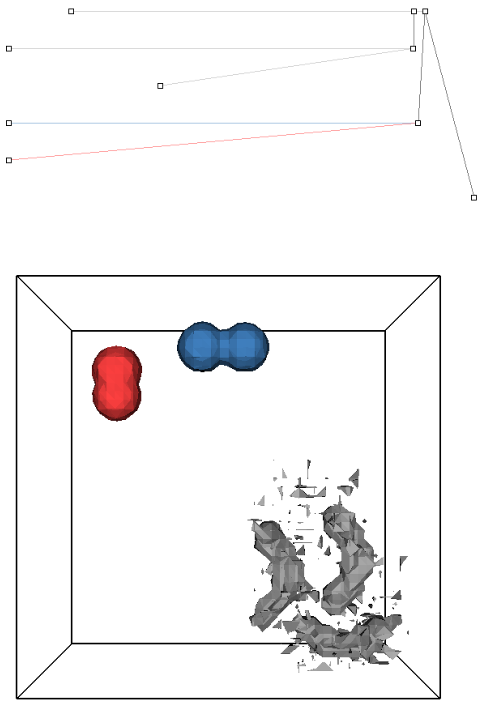
- The red and blue spheres are getting closer between steps 7 and 8. It also appears that both of them become smaller, and indeed the radius of the two contours is smaller. However, this is not because the radius of the spheres was indeed decreased, but due to the fact that the contour value used was increased for both of them. For one sphere the local contour value is given by a horizontal line intersecting its edge, above the join point between the intersection with the other feature. The join point of the red and the blue spheres is high in step 7 because the two spheres now share non-noise vertices on the mesh, and their contours are therefore unified early.

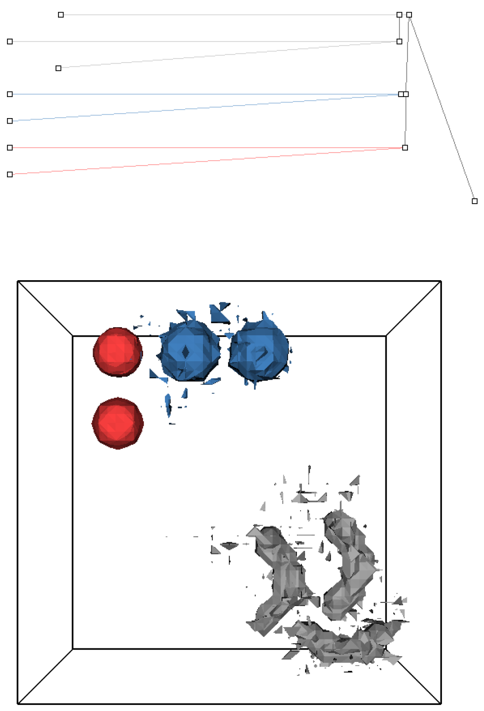

(A) Step 1

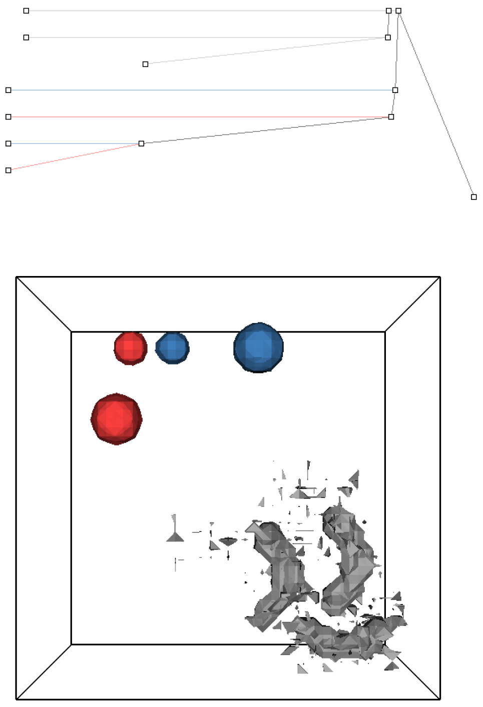

(B) Step 2

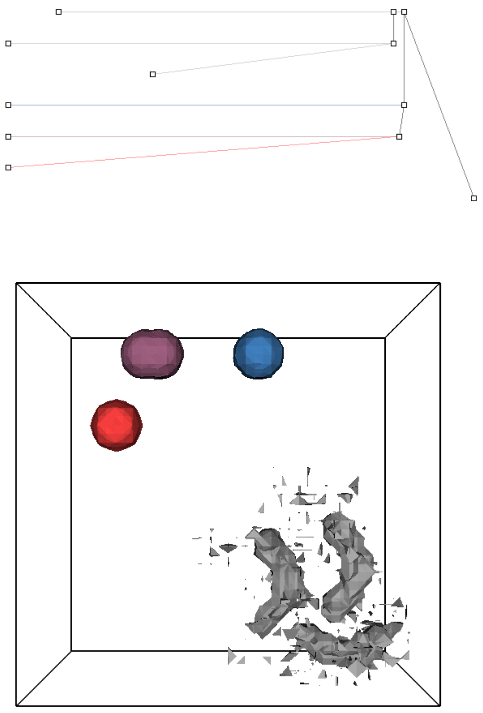

(C) Step 3


(D) Step 4


(E) Step 5


(F) Step 6


(G) Step 7


(H) Step 8

5.2 Tracking features in fields from aeronautical simulations

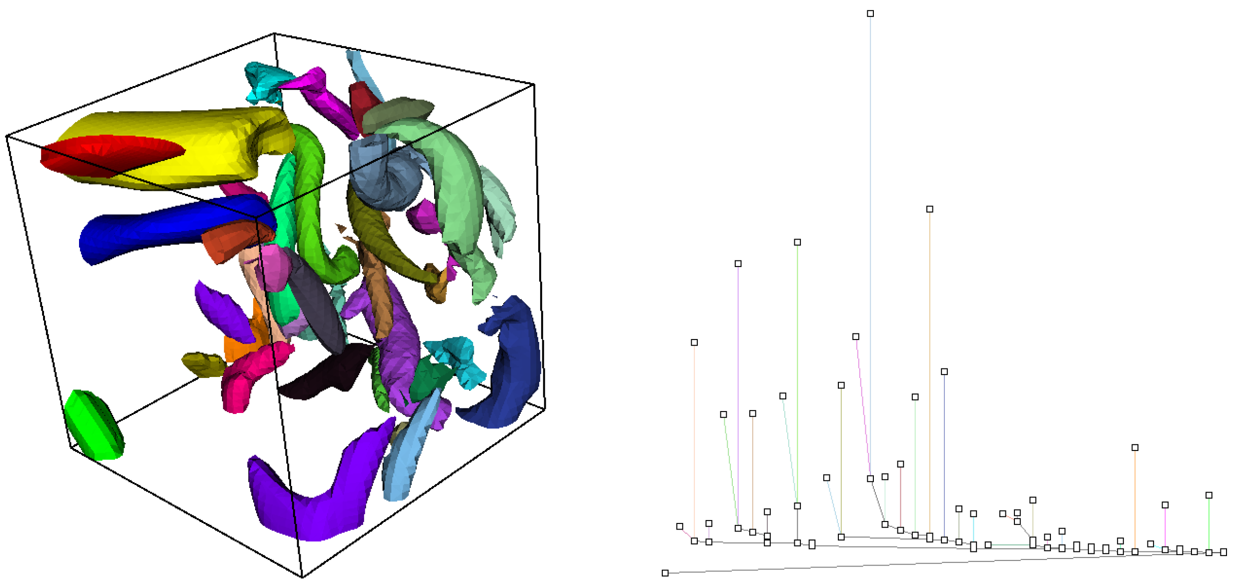
In this section, we want to prove the applicability of our algorithm on research and industry sized data, using a number of fluid simulations performed using PyFR. The large size of the fields is reflected in the size of the associated contour trees. These have been simplified using volume pruning and the pruning of negative values, such that the remaining contour tree arcs correspond to sensible vortex structures.

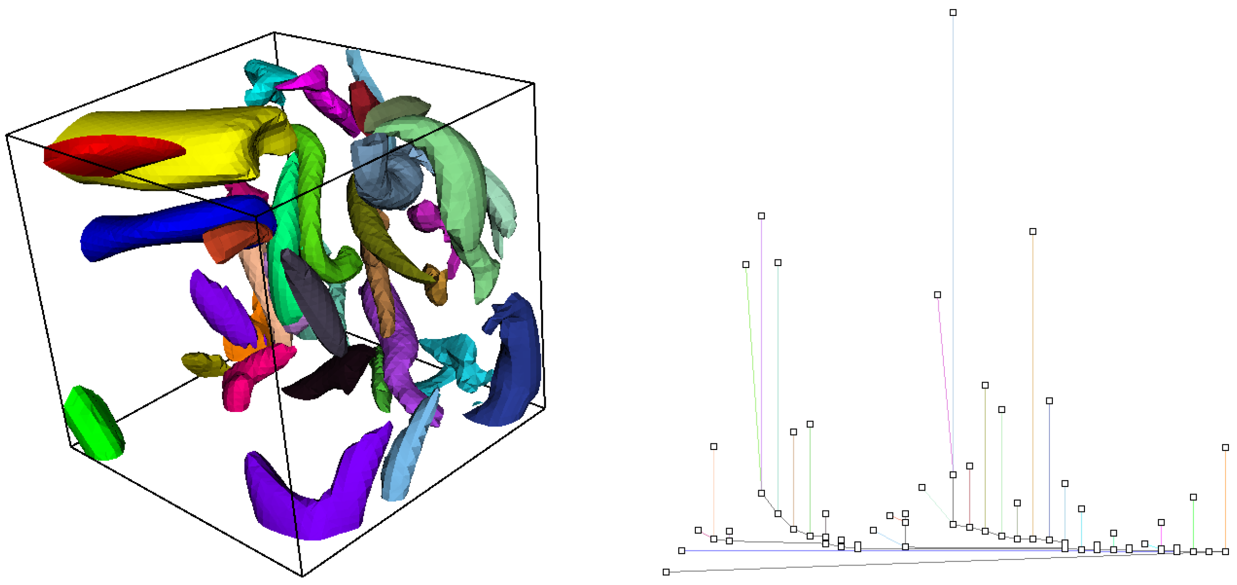
5.2.1 The fully developed stage of the Taylor Green Vortex

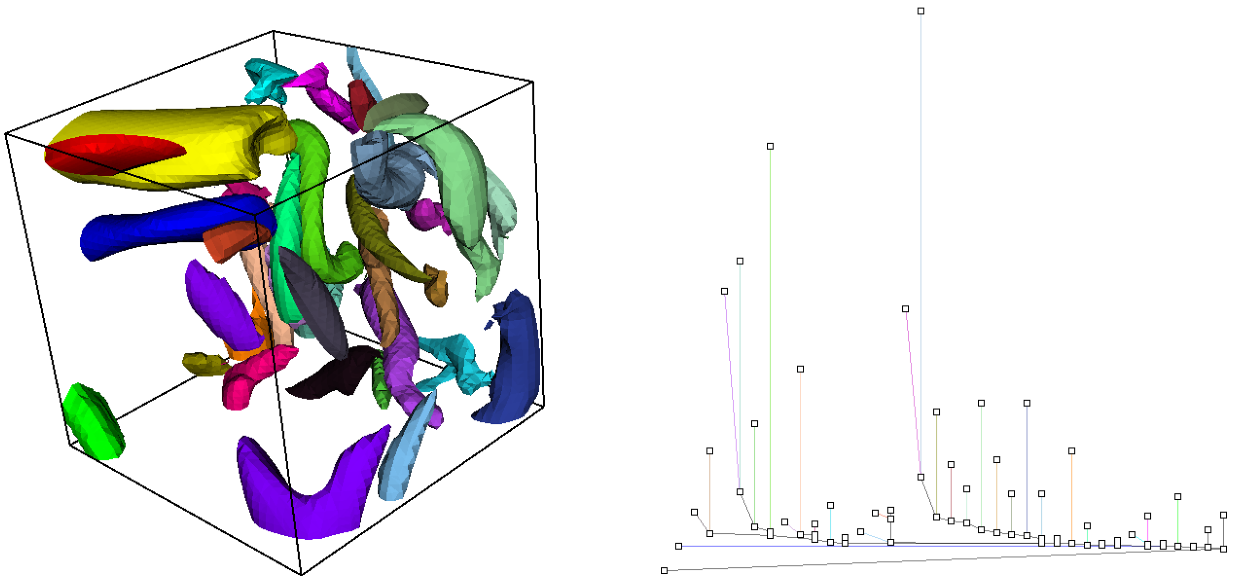
We are going to demonstrate the capability of the tool to track features in the fully developed stages of the Taylor Green Vortex simulation, described in Appendix A. This stage of the fluid flow is associated to large values of the Q-criteria, and to the presence of the largest number of vortex structures during the flow evolution.

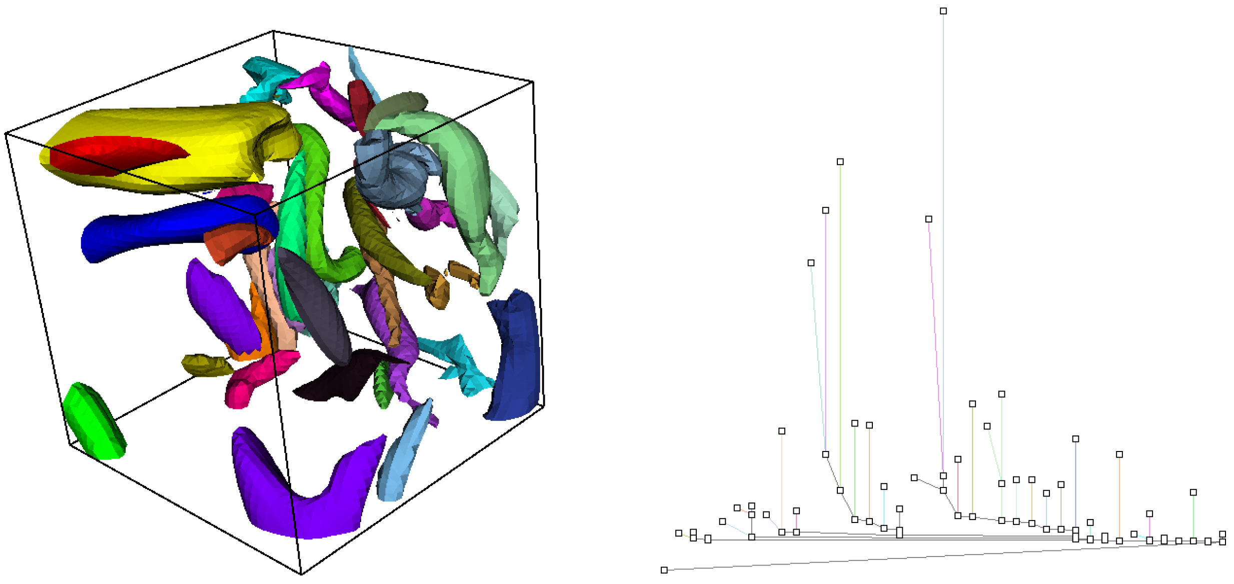
In order to be able to visually observe the vortex structures included in the field, we decided to extract a fixed subregion from consecutive meshes. The subregion represents approximately 0.8% of the entire field. The sequence of figures starting at 5.15a shows the evolution in time of the vortex features existing in the selected region. The initial choice of features was done again by assigning different tags to the upper leaves.

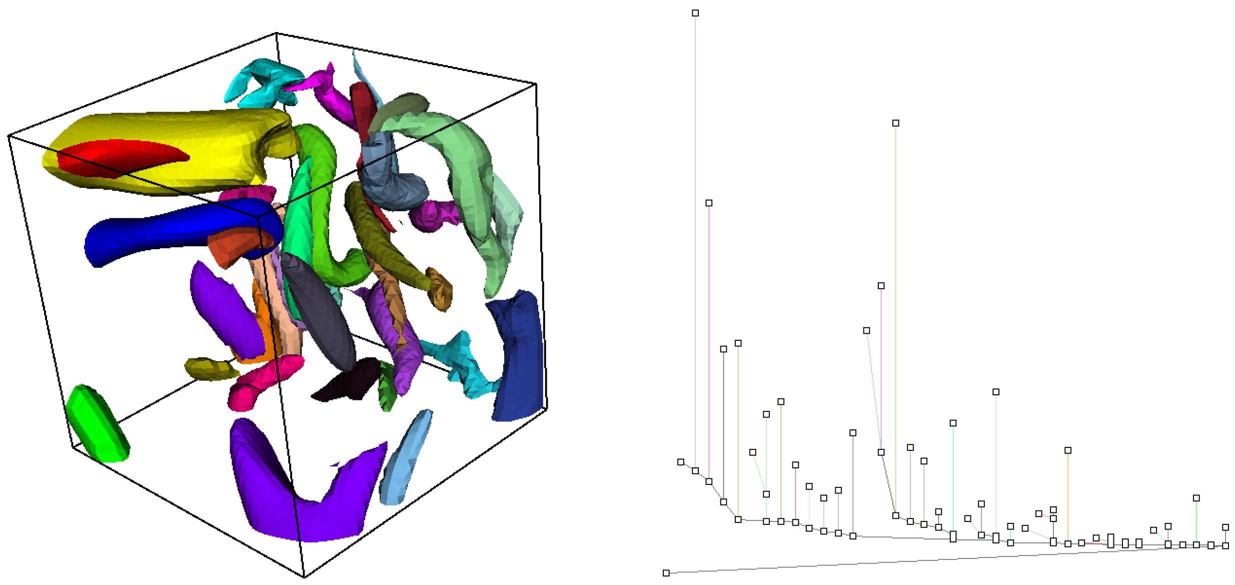
Using our tool, we are also able to track features selected manually. We are now going to focus on a particular vortex structure, which we observed to suffer a split during the visualized time steps. This is shown in Figures starting at 5.16a, where we included the steps relevant for the split event. We can see how in step 4 the selected (red) arc becomes two upper leaves (that were not visible before because of the volume pruning) joined by an arc. In step 7, the two upper leaves are no longer connected by an immediate join arc, and the feature is splitting.

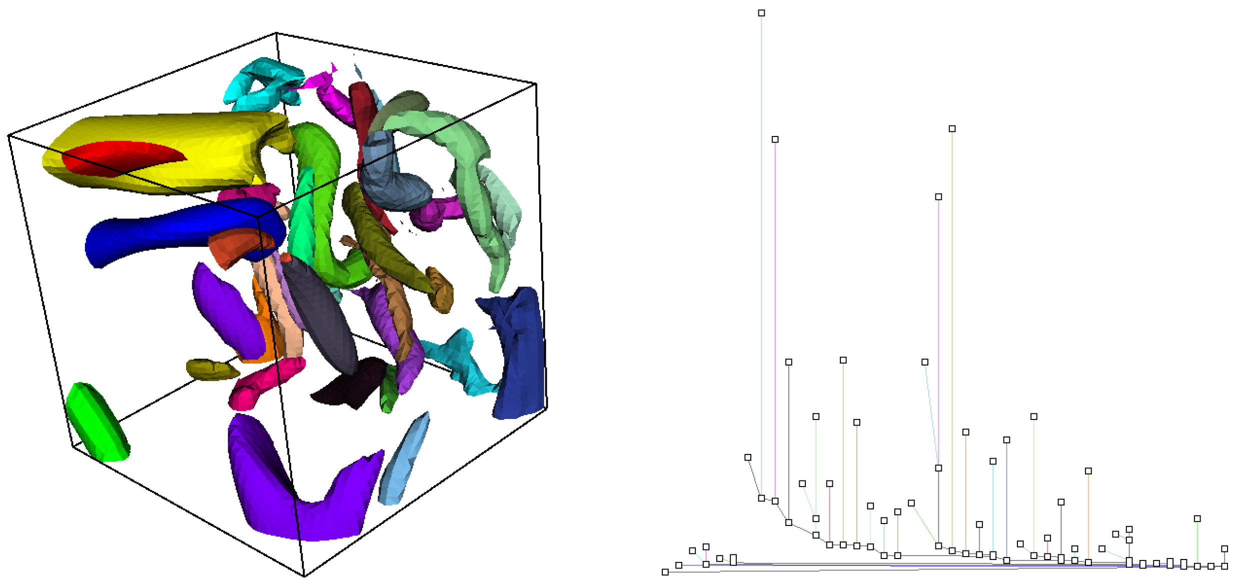

(A) Step 1

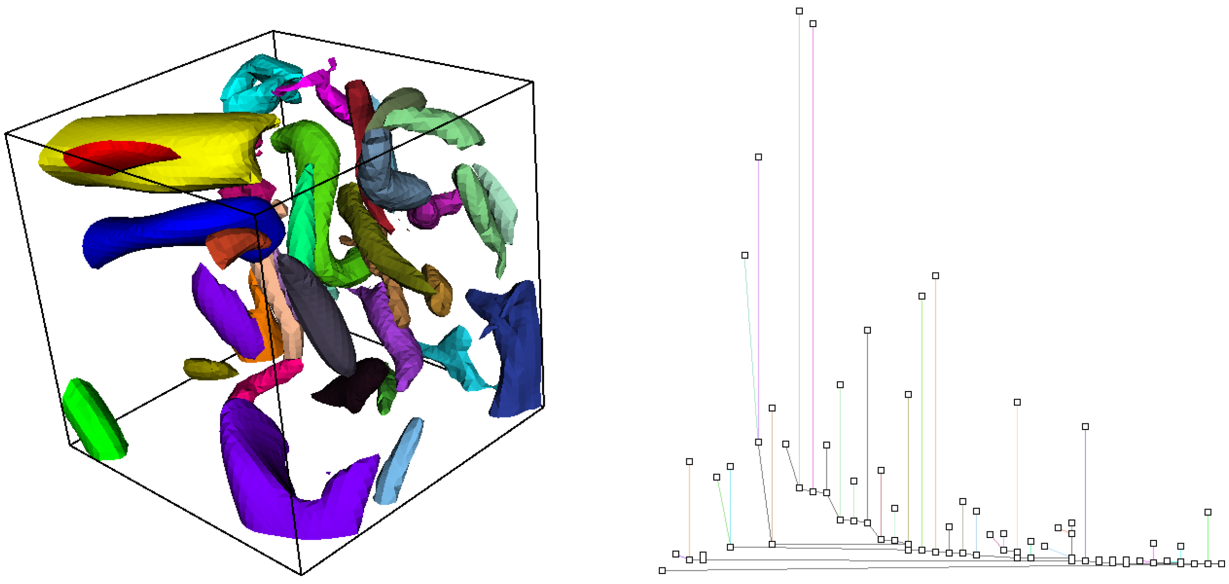

(B) Step 2

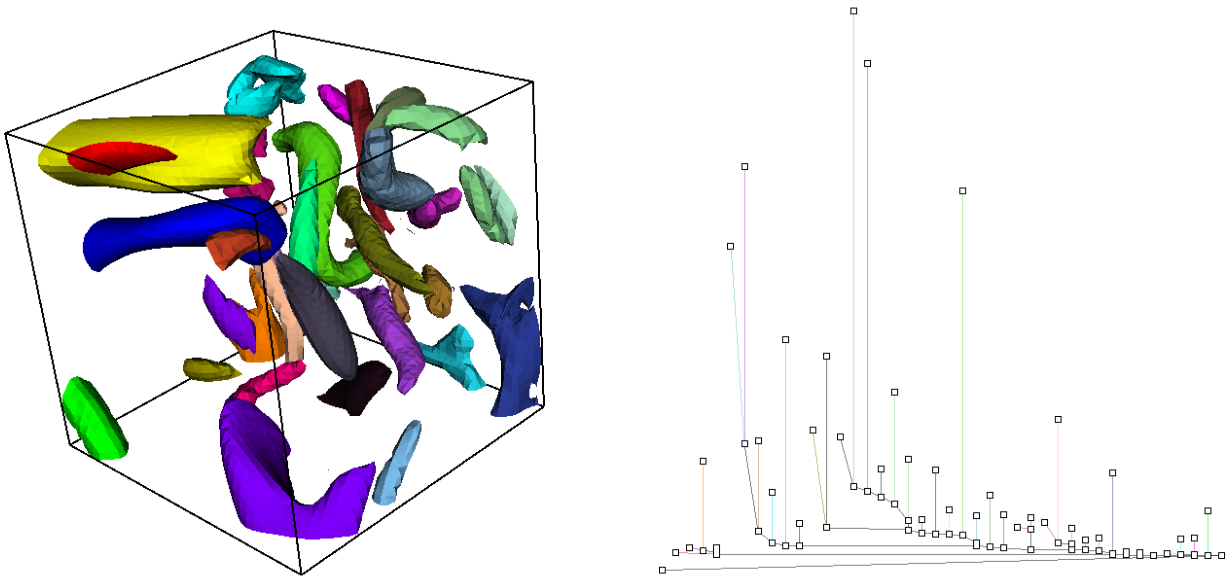

(C) Step 3

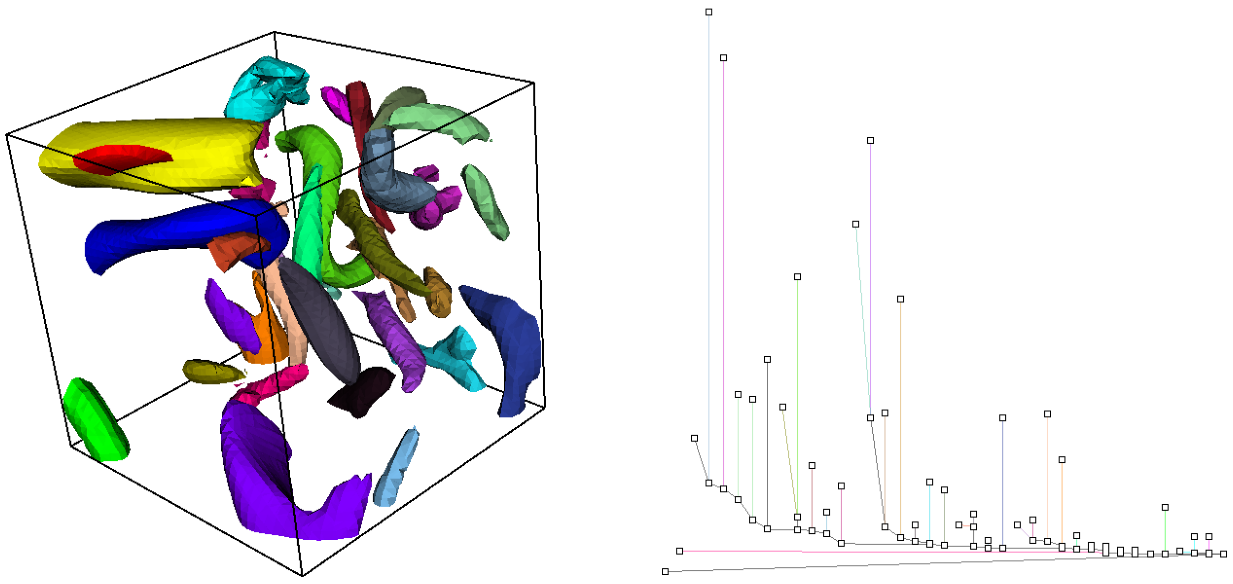

(D) Step 4

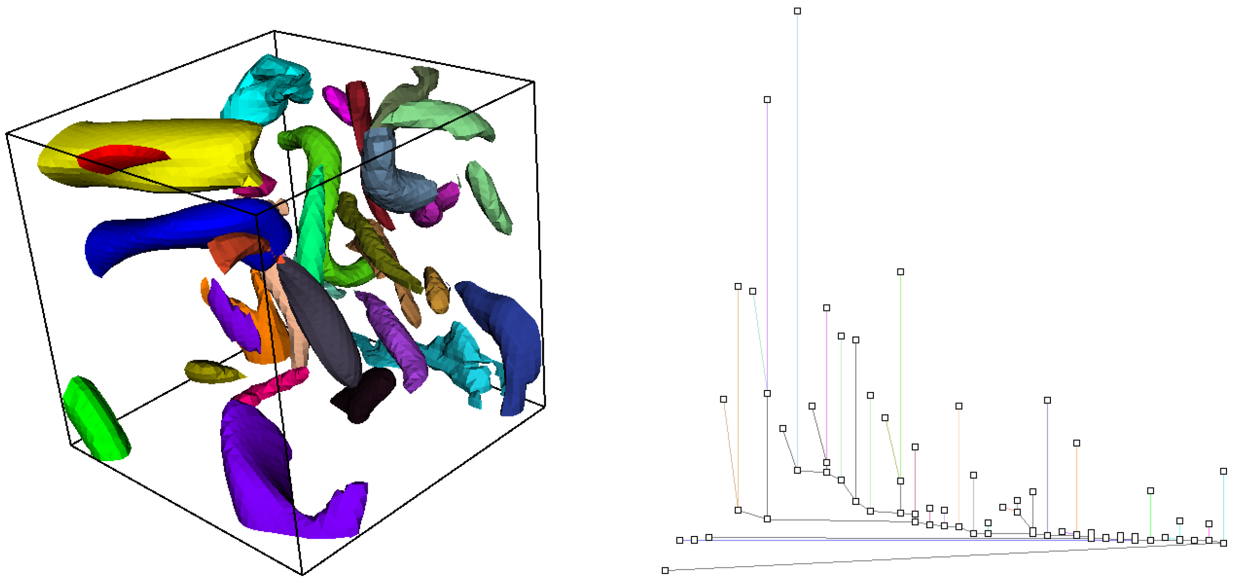

(E) Step 5

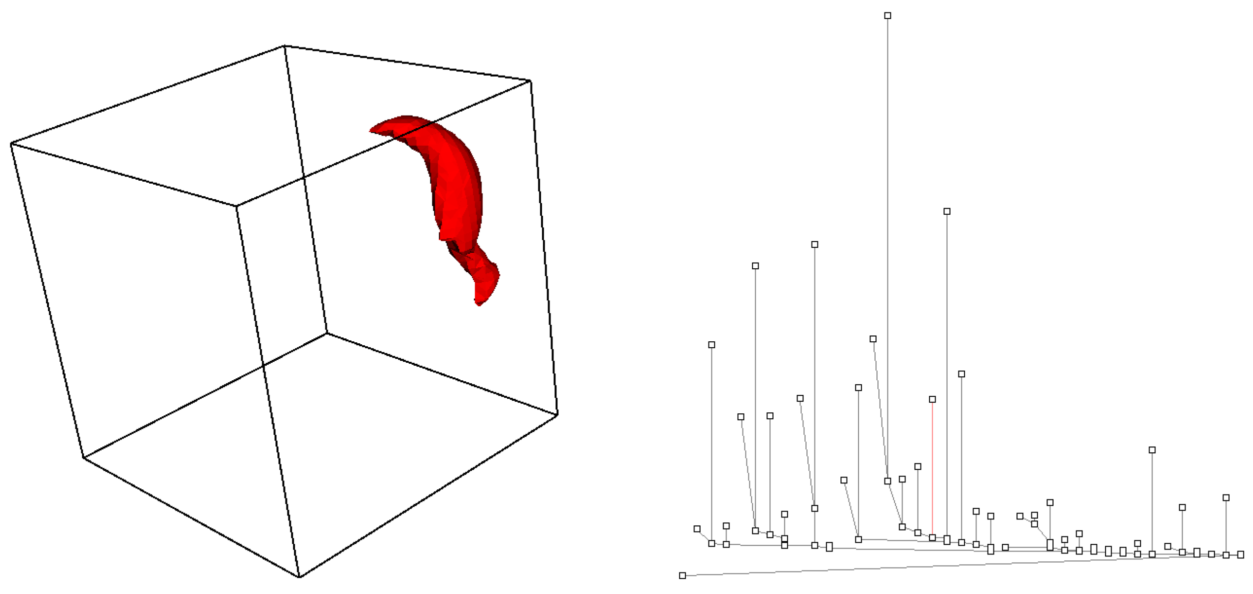

(F) Step 6

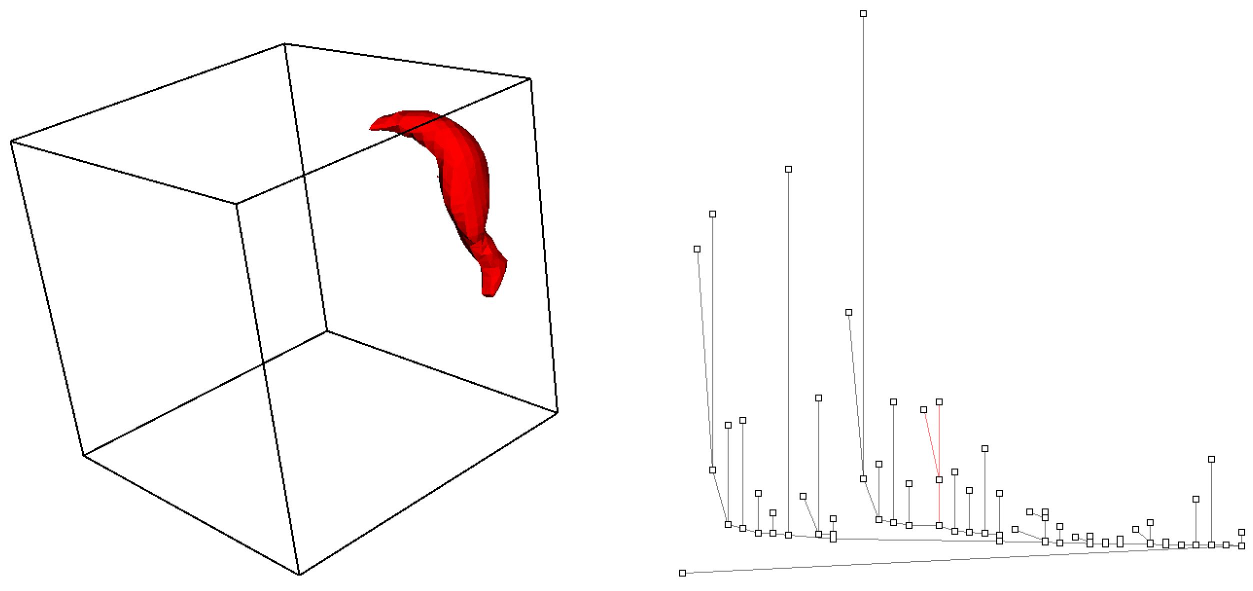

(G) Step 7

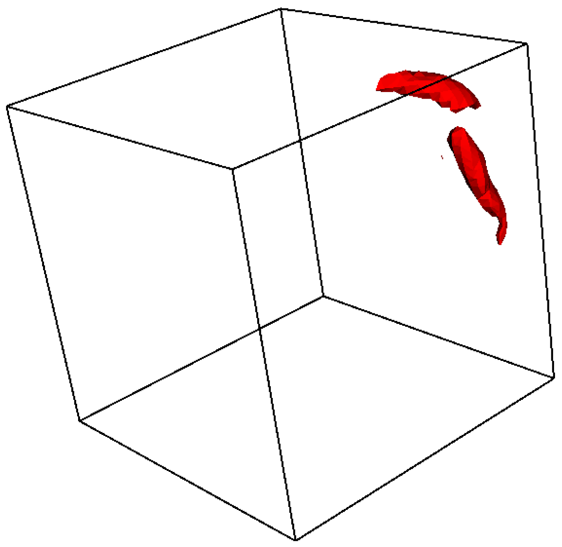

(H) Step 8

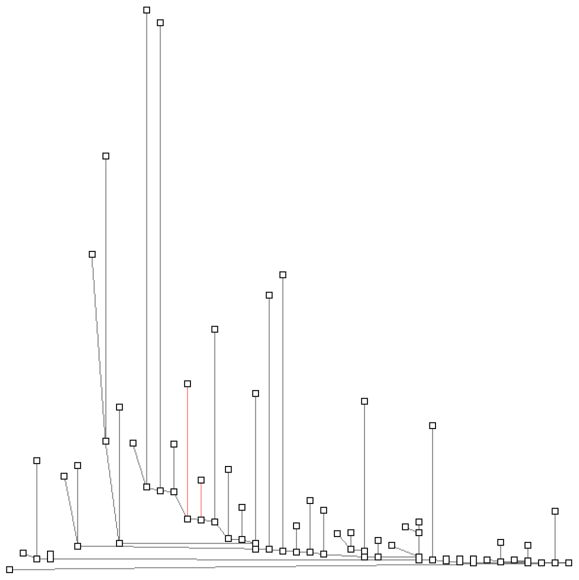

(I) Step 9


(J) Step 10


(A) Step 1


(B) Step 4





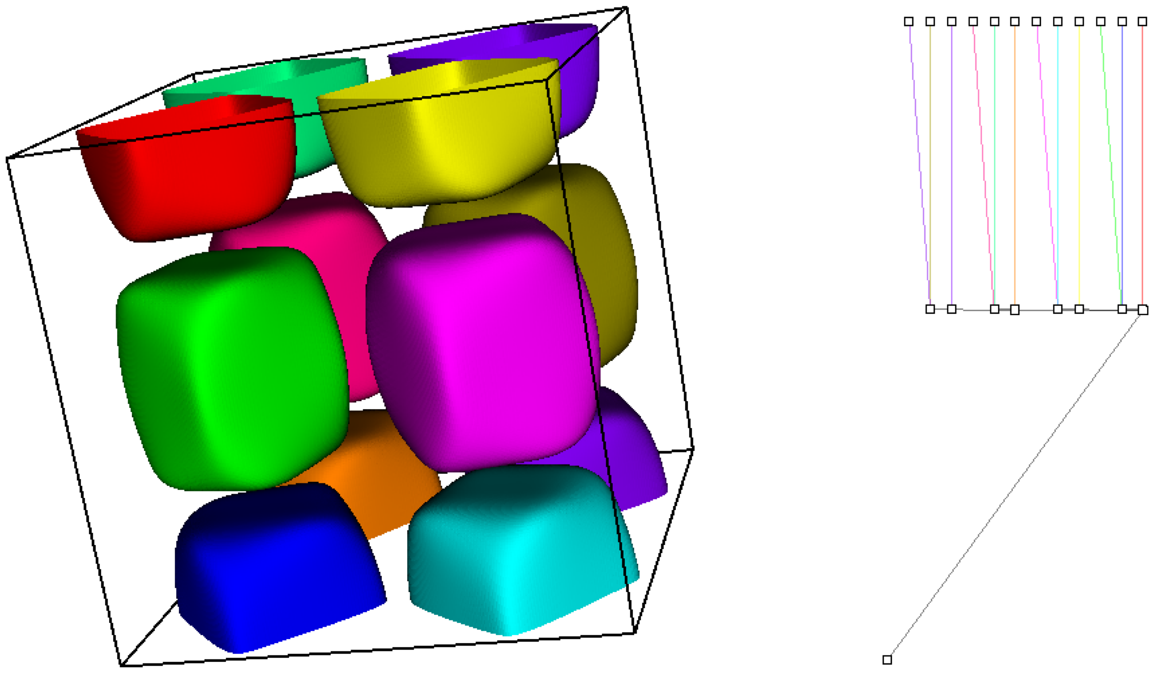
(c) Step 7

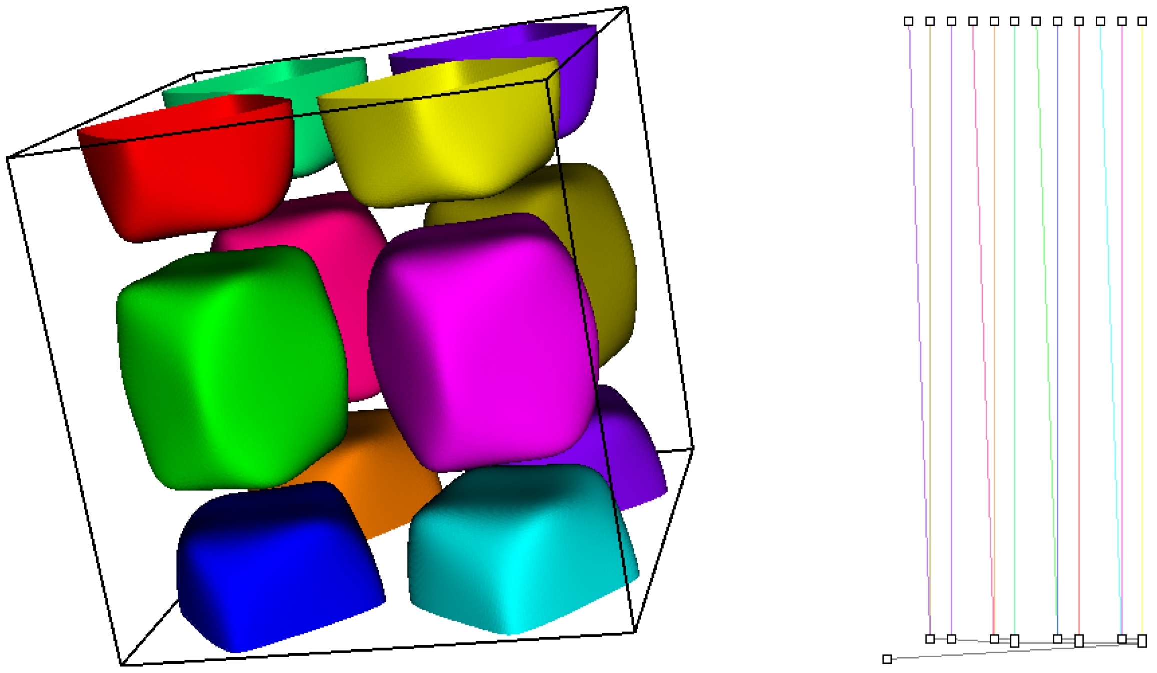
5.2.2 The developing stage of the Taylor Green Vortex

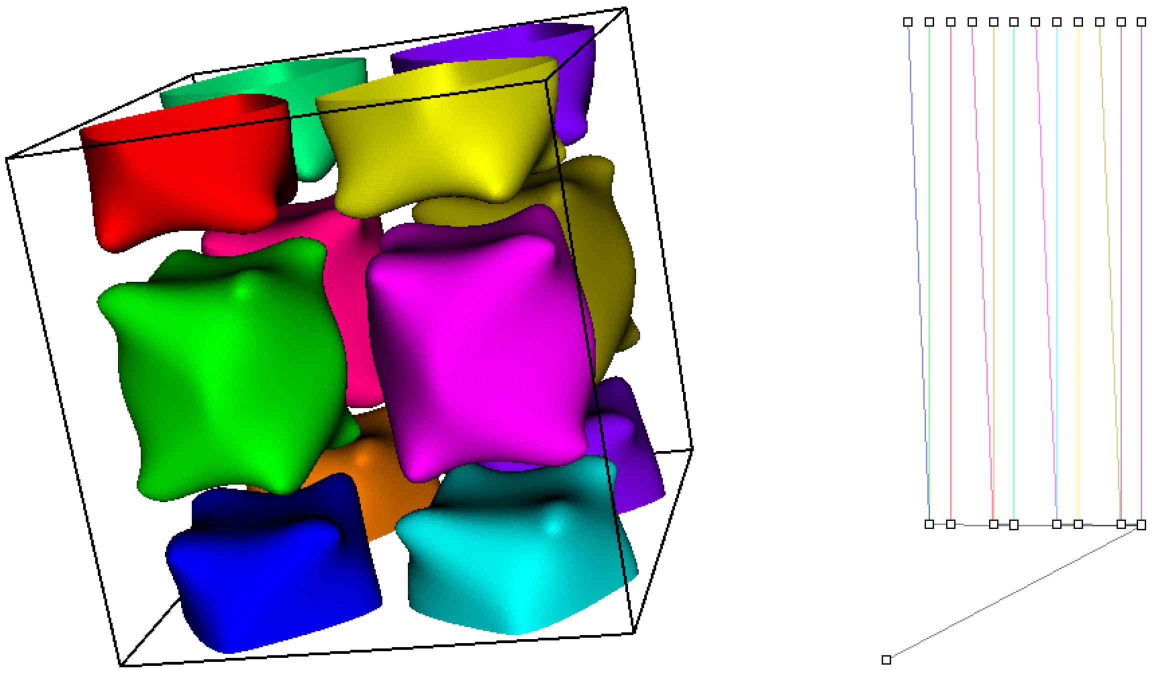
We are now going to visualize the evolution of the Taylor Green Vortex during its incipient stage. This stage of the simulation has very small Q-criteria values, in the range [-0.03, 0.008]. In contrast, the Q-criteria values in the fully developed fields from the previous subsection fall in the range [-2.5, 2.5].

The field contains much noise as a consequence of the small range of Q-criteria, but the interesting features are large, and at small positive values. This enabled us to prune the resulting contour tree more aggressively, as much of its upper portion was associated to noise. This way, although we now consider the entire mesh for analysis (in contrast to the previous scenario, where a mesh subregion was extracted), the structure of the contour tree is simpler. We can see how the flow evolves, and how its features are successfully tracked in Figure 5.17.

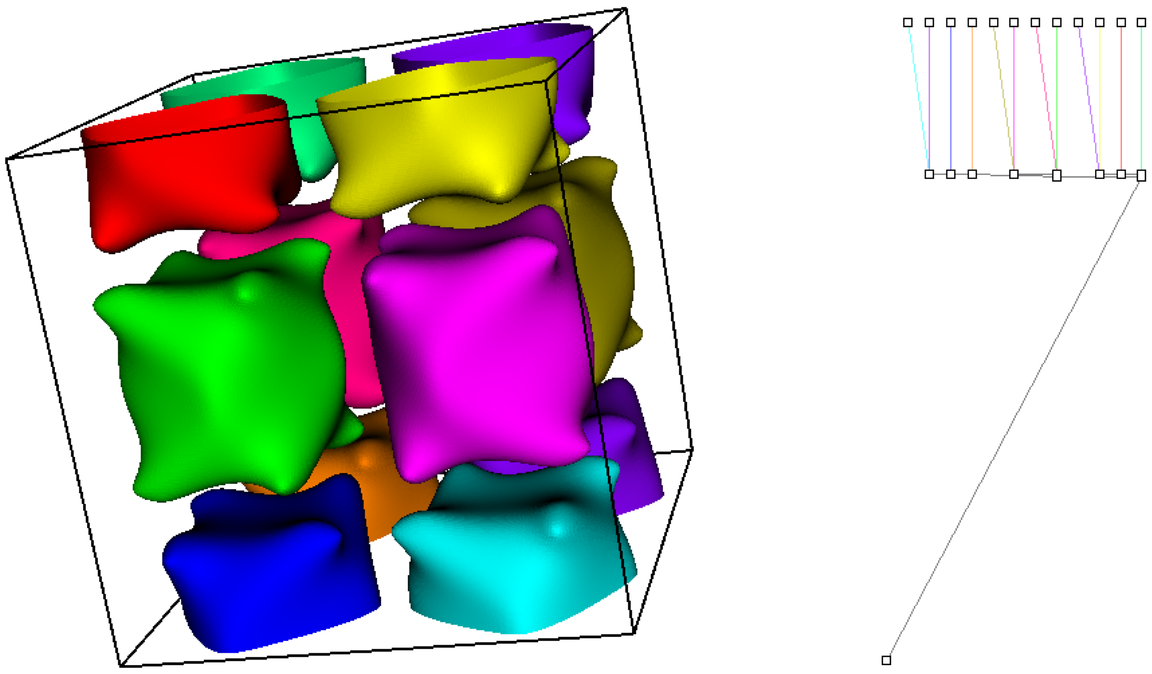
Again, choosing the initial features was done by assigning different tags to the upper leaves. We notice that in step 5 (Figure 5.17e) the arcs representing the middle structures are becoming a join. This is due to the fact that the currently joined structures had been pruned in the first four steps, but are not anymore now since they grew and their size became significant. We are able to focus on these small structures by running the analysis backwards (Figure 5.18). We see how they lose significance, and are eventually pruned from the contour tree, causing their tags to be joint. Once the tags are joint, we omit the rest of the steps as the evolution of the feature can be seen in the first three steps of the forward simulation.


(A) Step 1

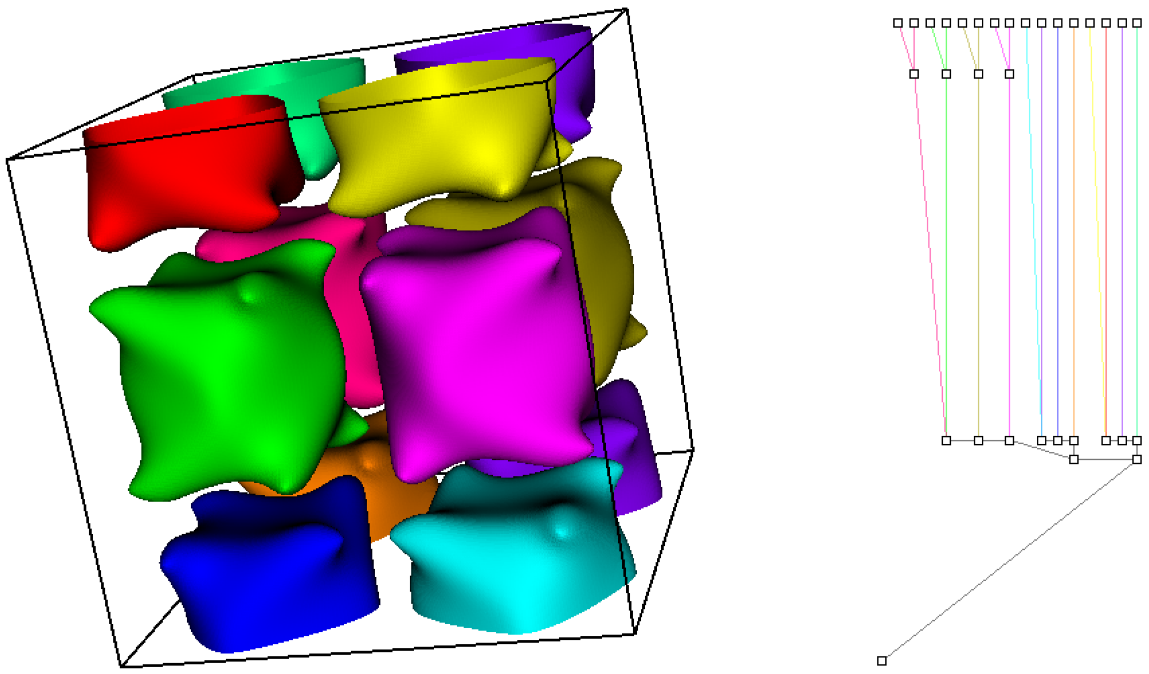

(B) Step 2

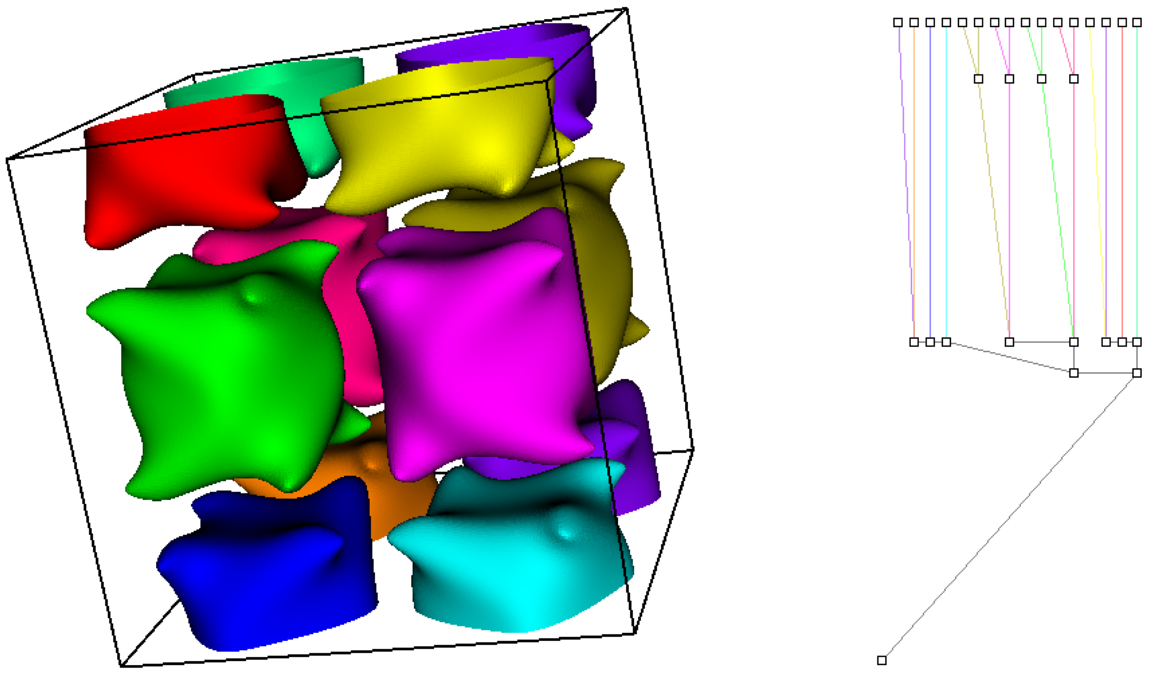


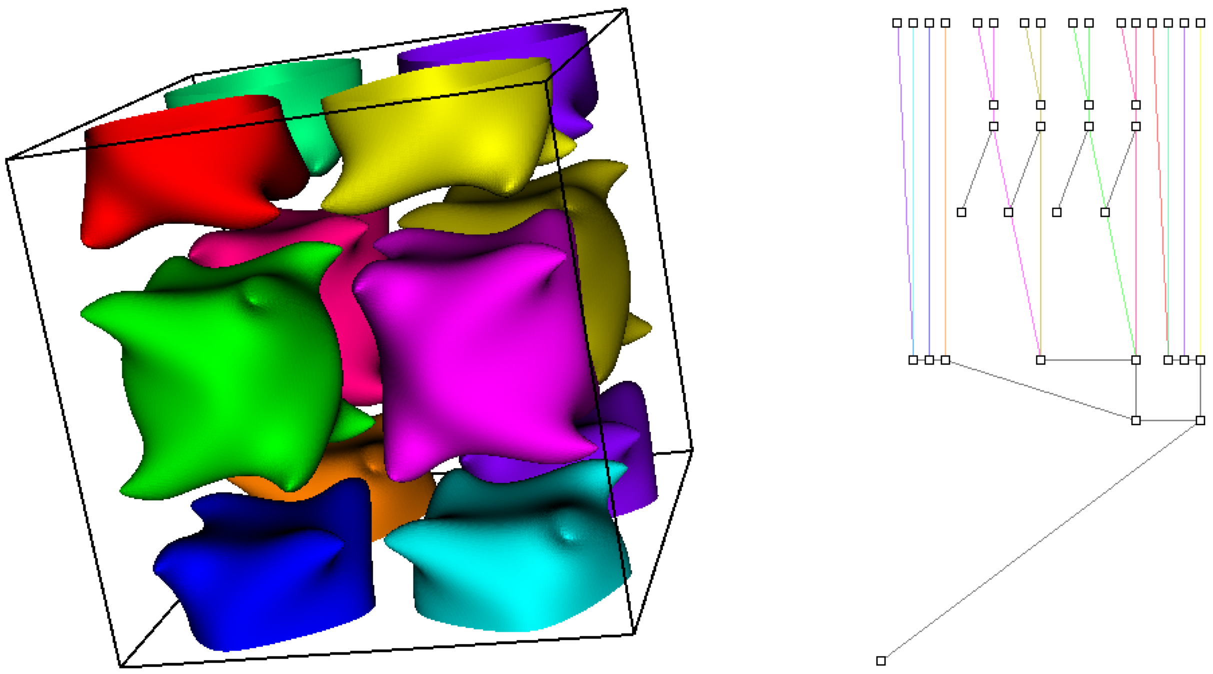
(C) Step 3

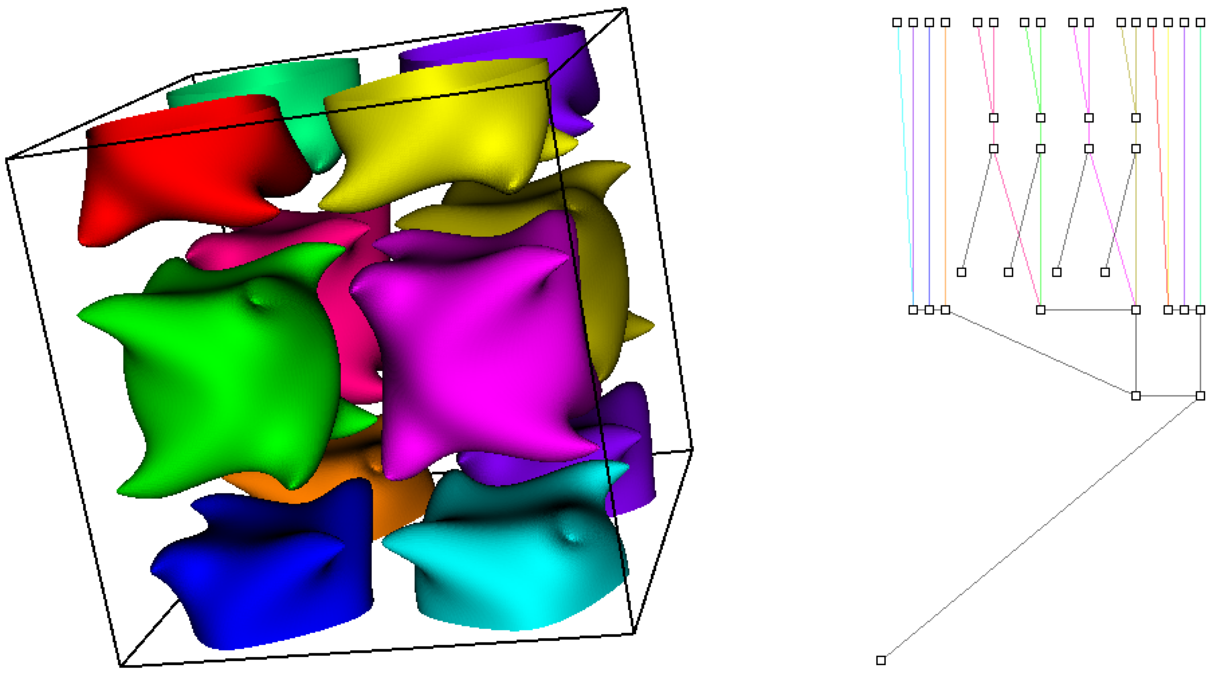


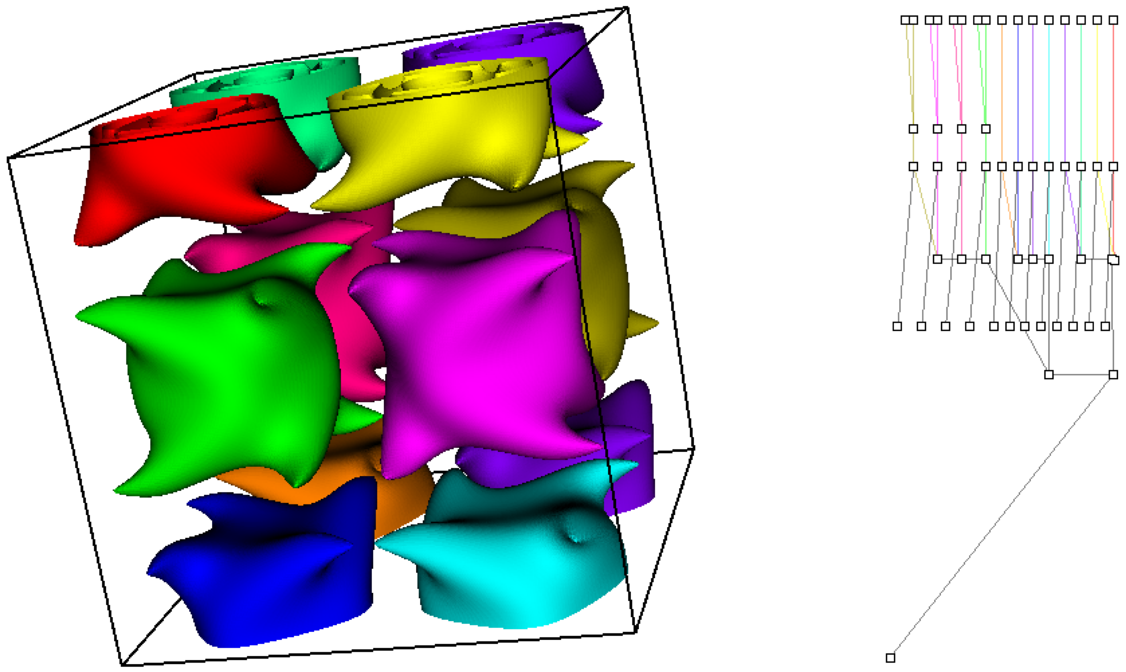
(D) Step 4


(E) Step 5

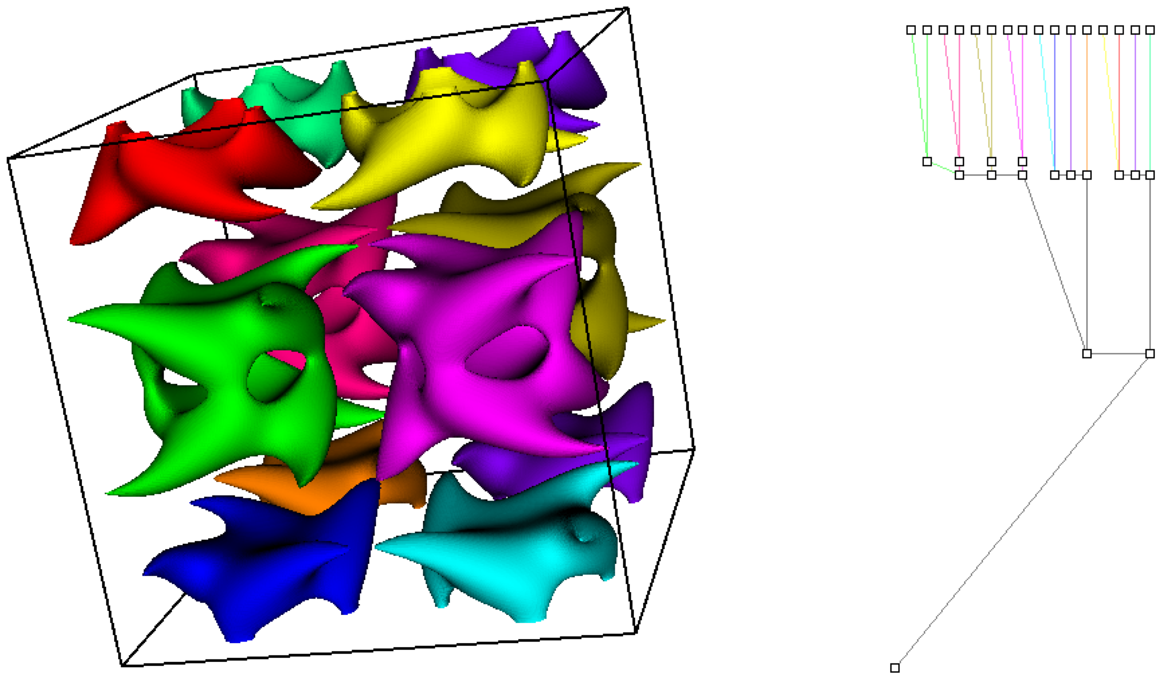

(F) Step 6


(G) Step 7


(H) Step 8

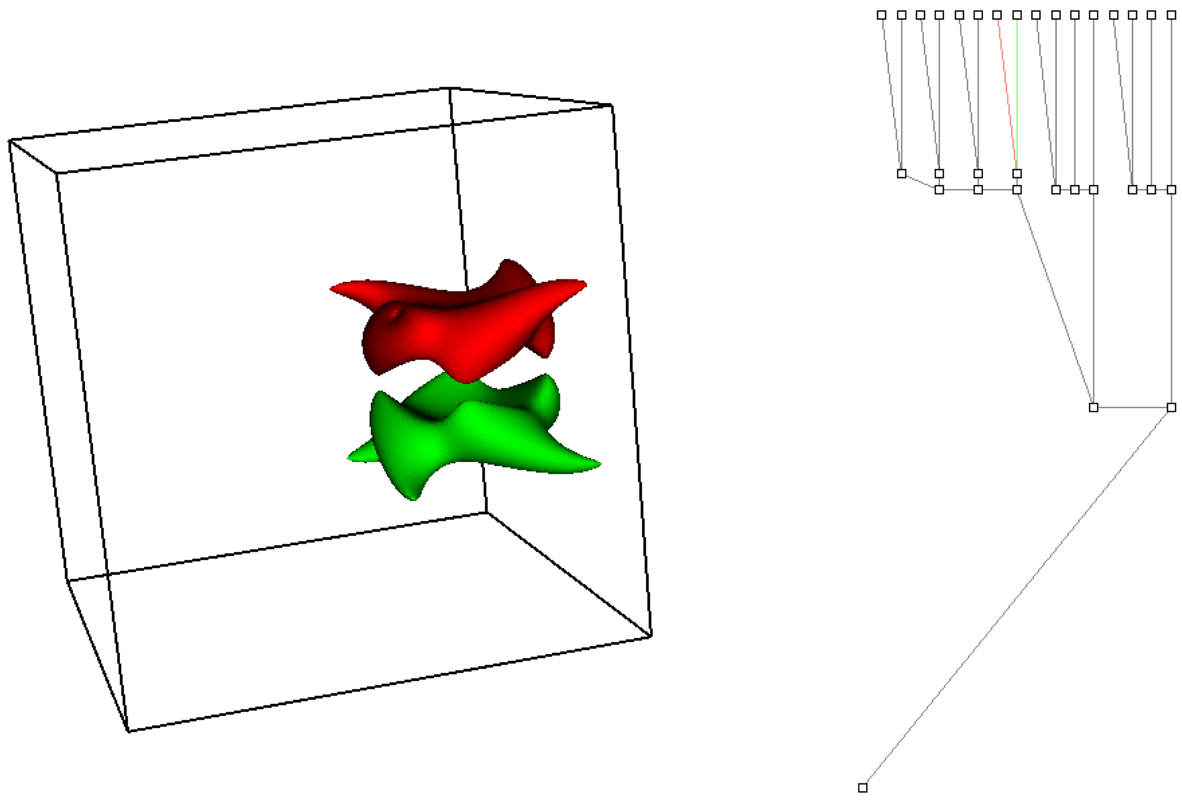


(I) Step 9

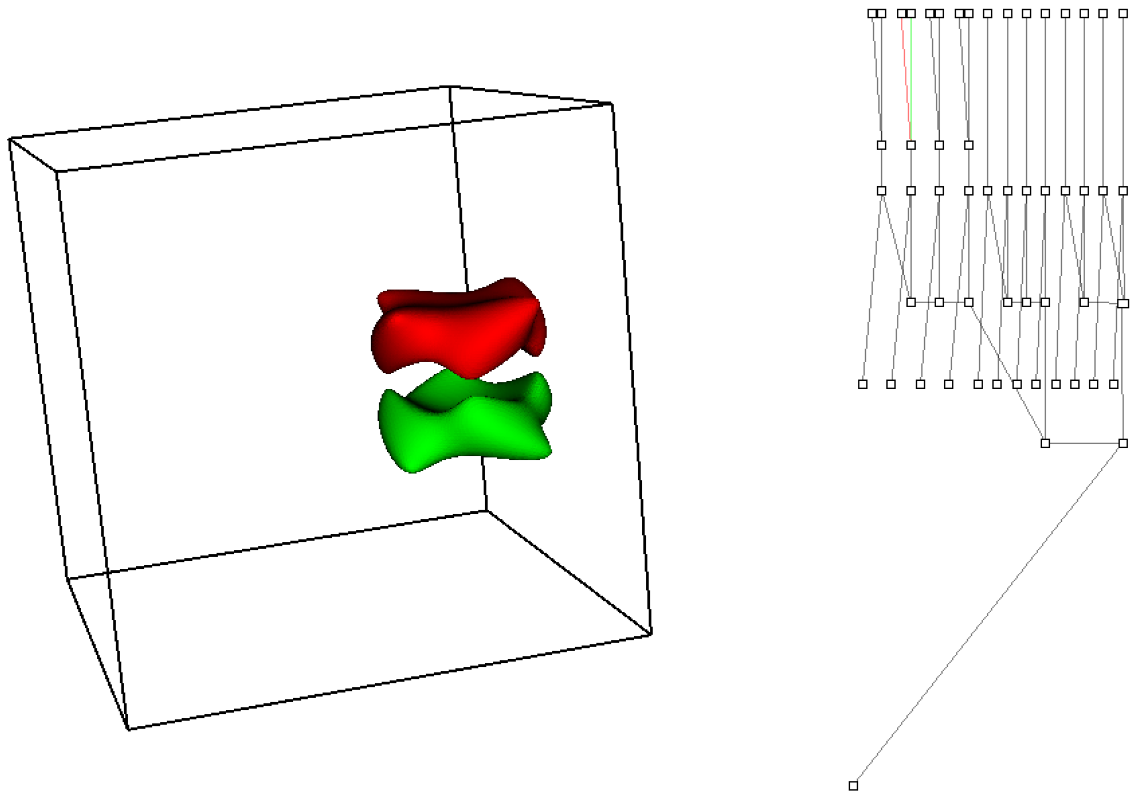


(J) Step 10

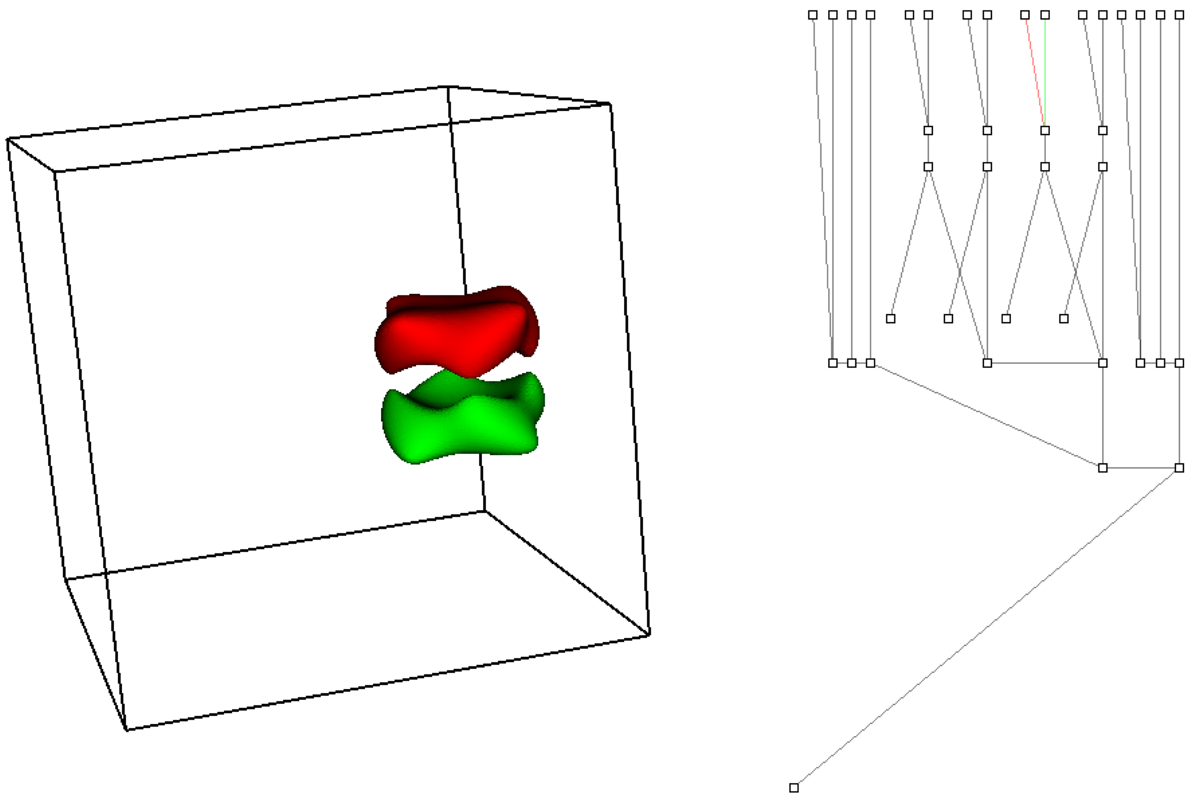
FIGURE 5.17: Tracking features in the incipient stage of the Taylor Green Vortex.

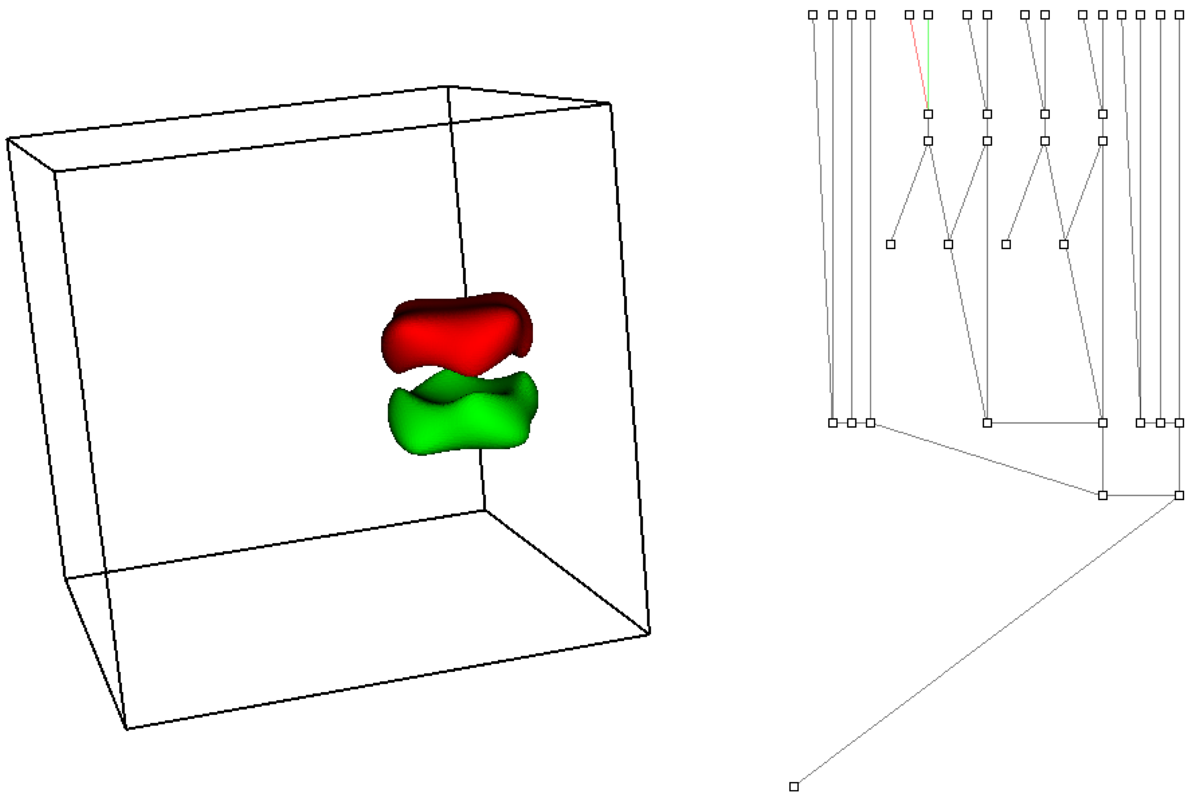


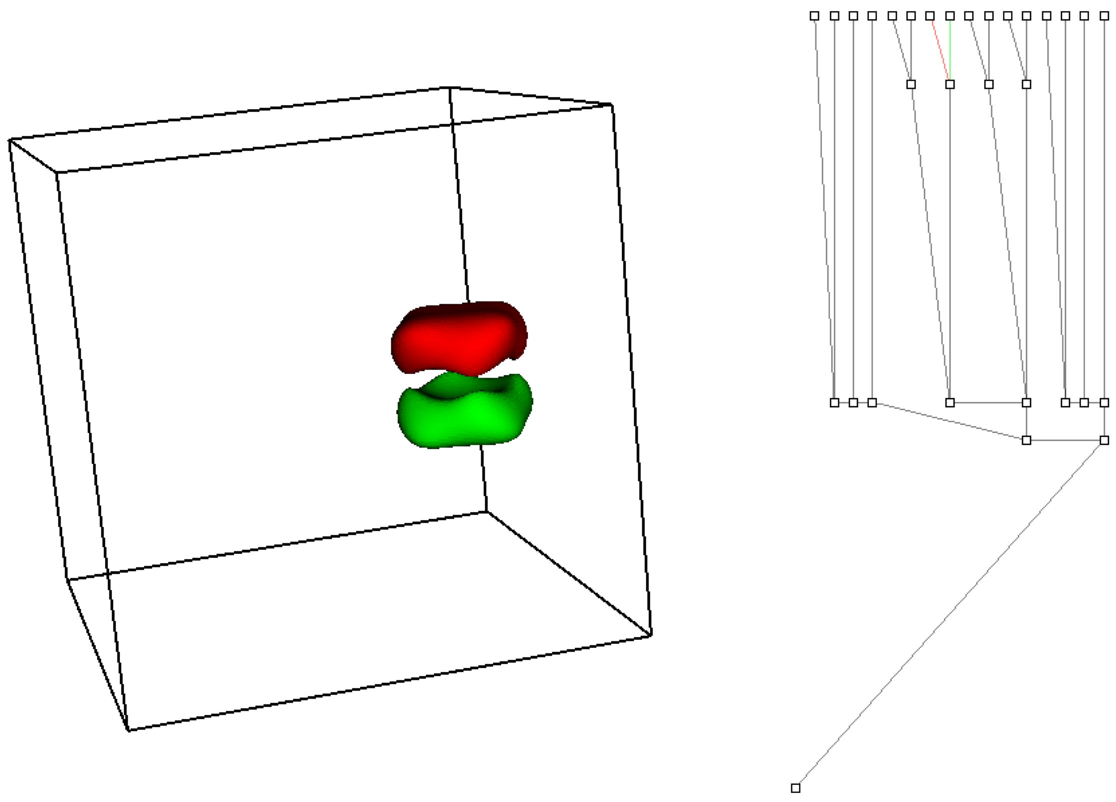
(A) Step 10

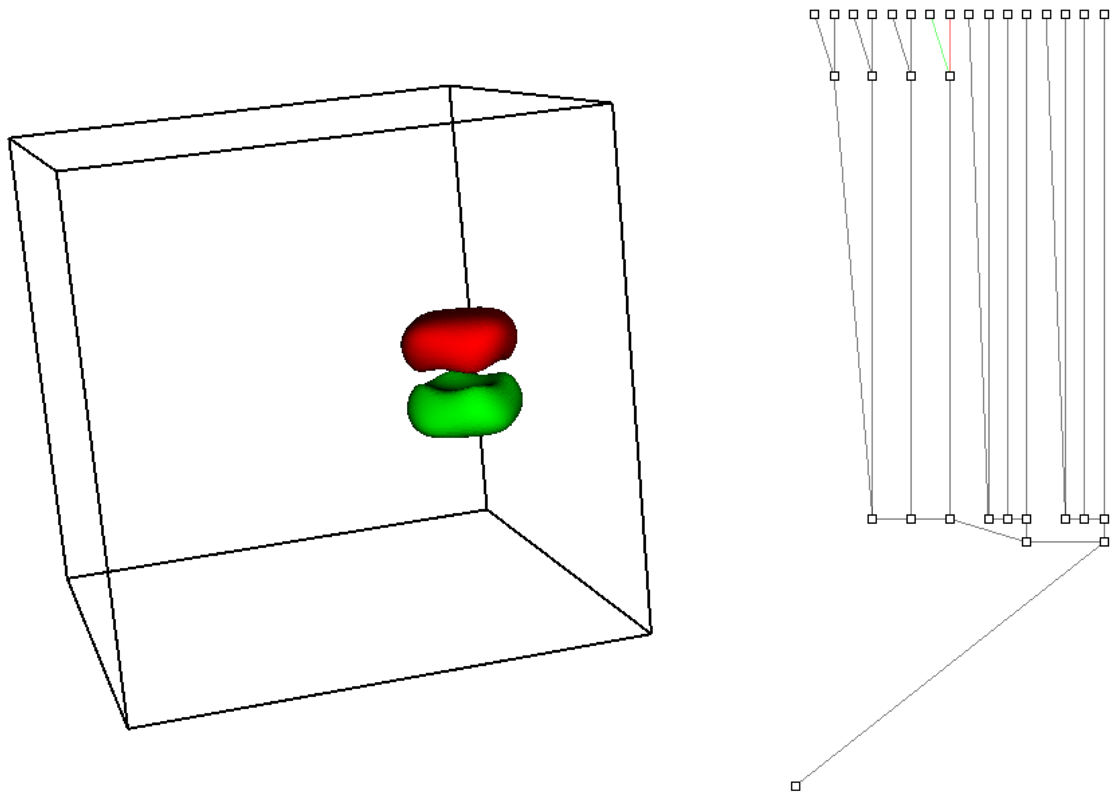


(B) Step 9


(C) Step 8


(D) Step 7


(E) Step 6


(F) Step 5

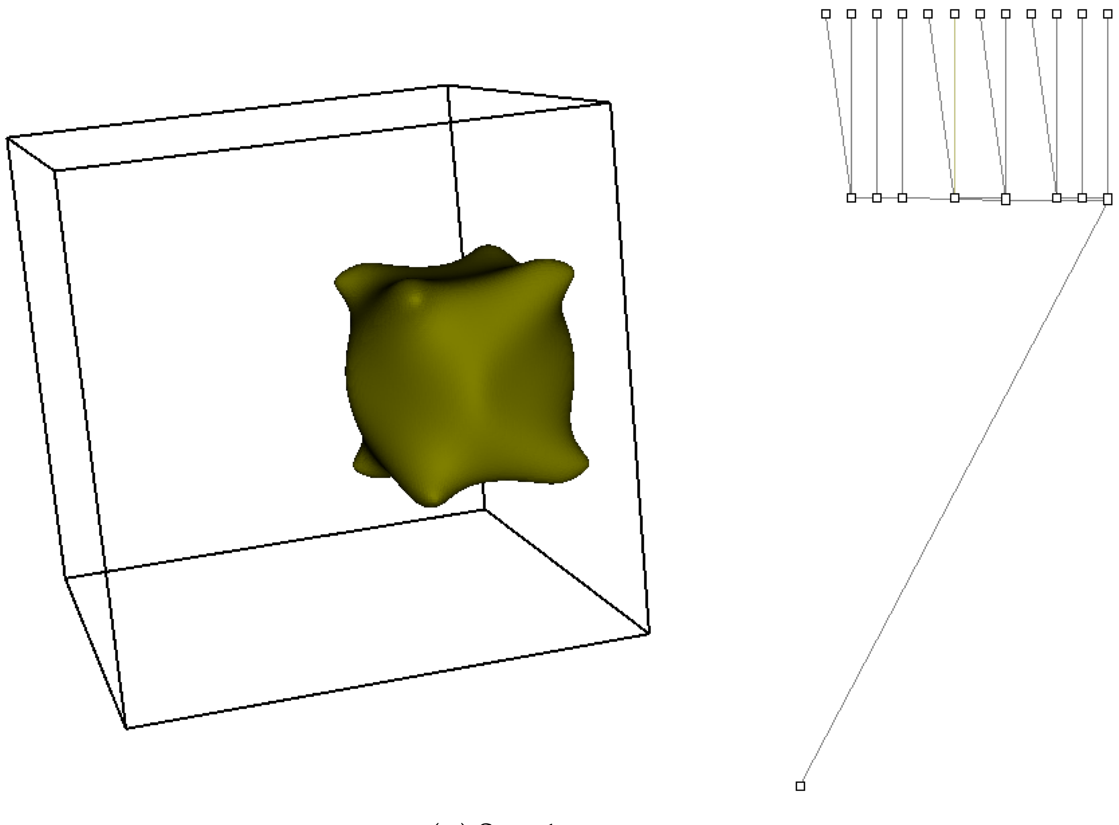

(G) Step 4

FIGURE 5.18: Backwards tracking of features in the incipient stage of the Taylor Green Vortex.

5.2.3 Selected features in the fully-developed stage of the turbulent plane Couette flow

We are going to visualize the plane Couette flow at its fully developed stage. Figure 5.19 contains an image produced using our tool, which displays the contour tree of the field, as well as all the vortex structures corresponding to upper arcs in the tree. We can see the increased number of features and arcs in the contour tree.

Since it is difficult to observe the temporal evolution of the entire field, we are going to focus on a few vortex structures that we selected manually. Also, we are going to omit the contour trees from the pictures. The time evolution of the visualized features is included in Figure 5.22, where the viewpoint is normal on the XY plane. This way, we can observe how vortex structures in the top half of the field have a motion tendency towards the negative x direction, while the bottom half features are moving towards the positive x direction. This opposite movement behavior is associated to the opposite directions of the two moving walls bounding the flow, as explained in Appendix A.

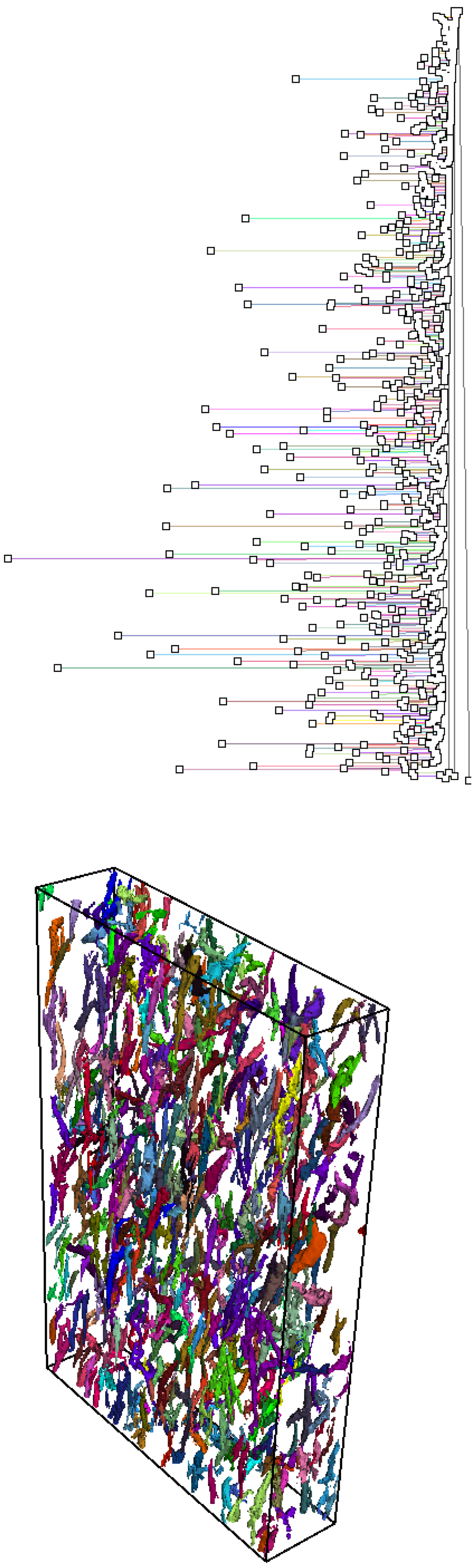
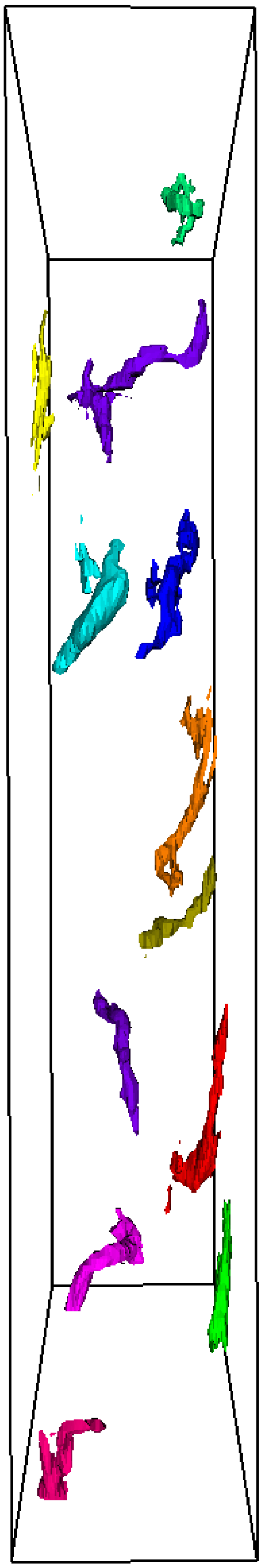
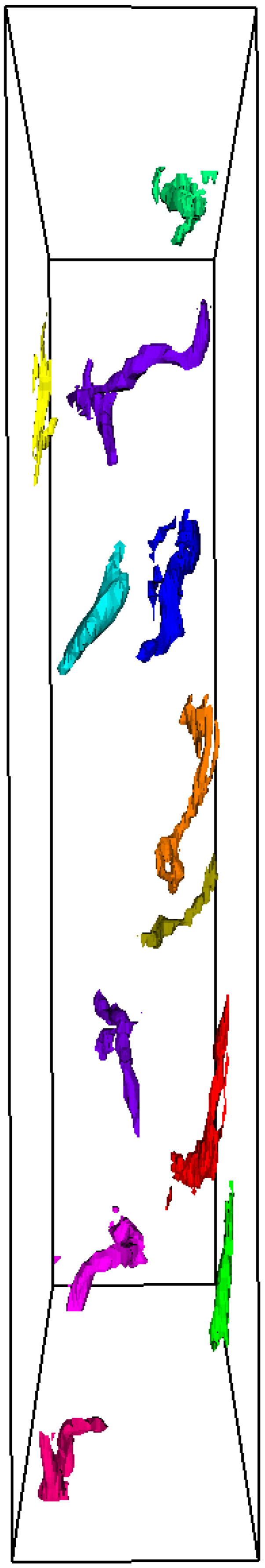
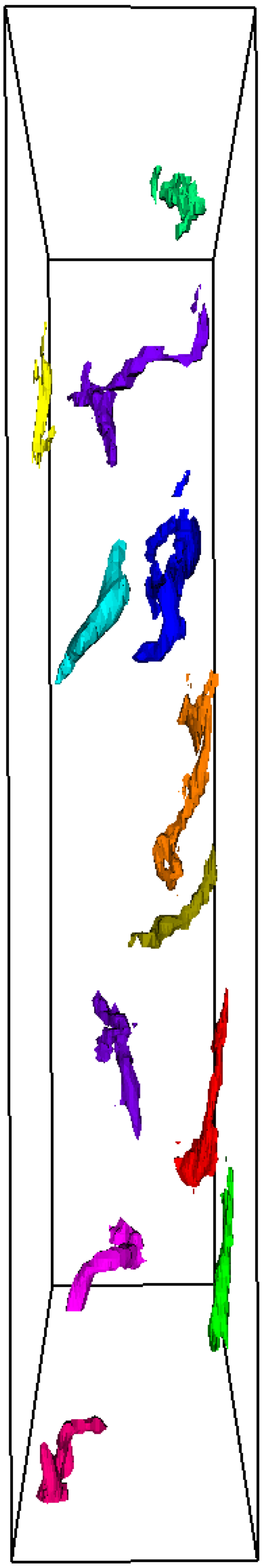
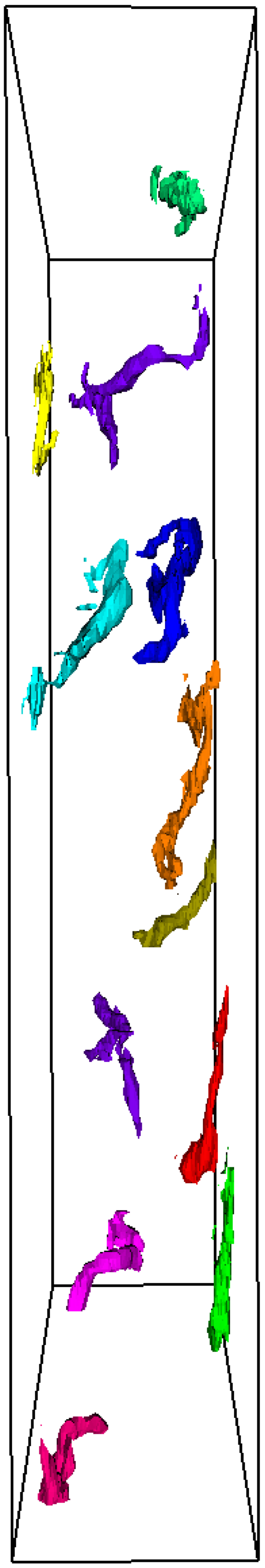


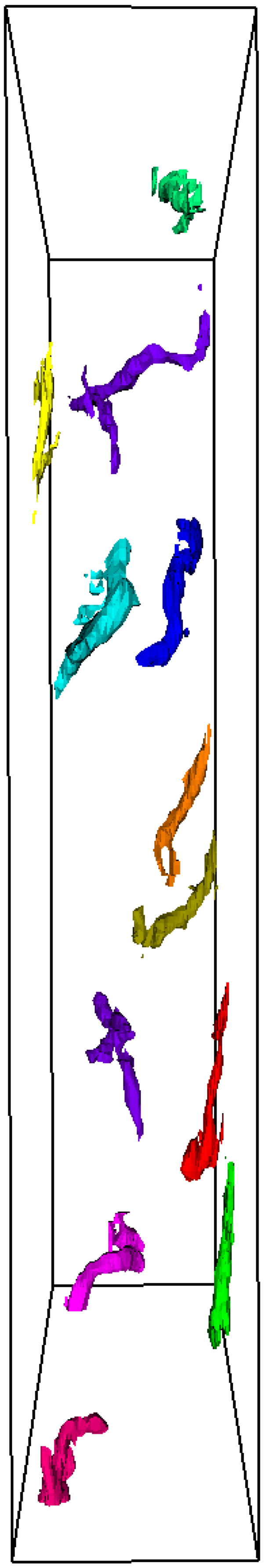
FIGURE 5.19: Vortex structures observed in the turbulent plane Couette flow.

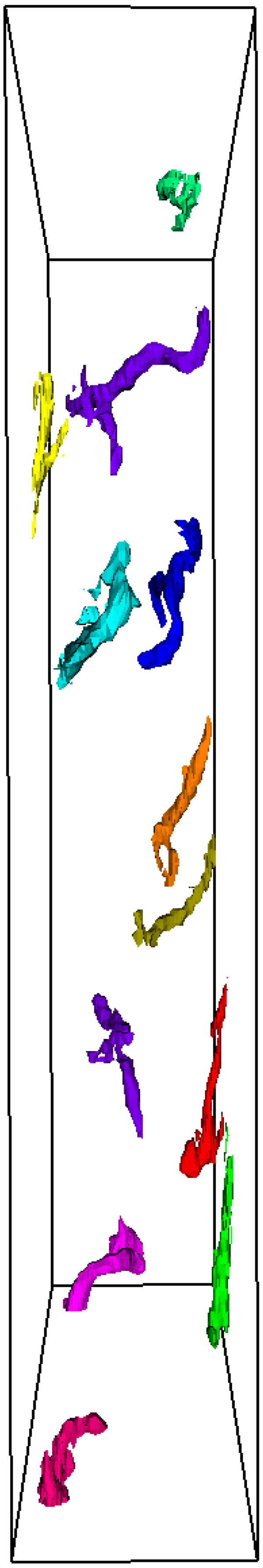

(A) Step 1

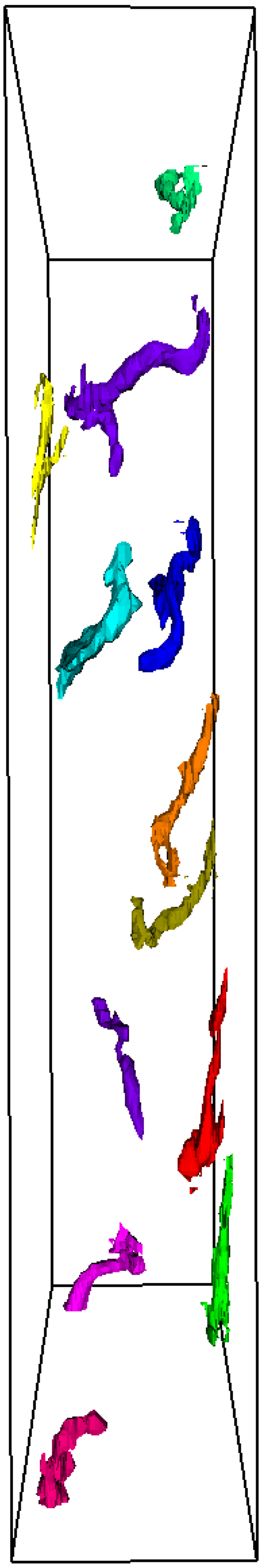

(B) Step 2

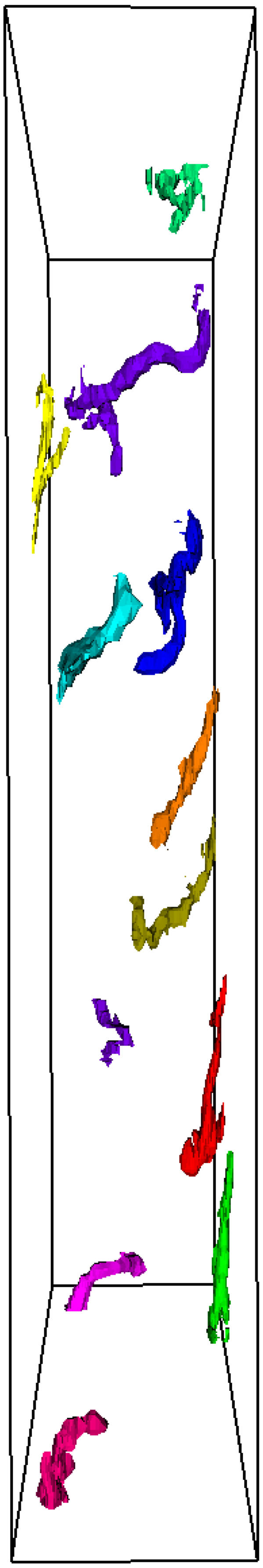

(C) Step 3


(A) Step 4


(B) Step 5


(C) Step 6


(A) Step 7


(B) Step 8

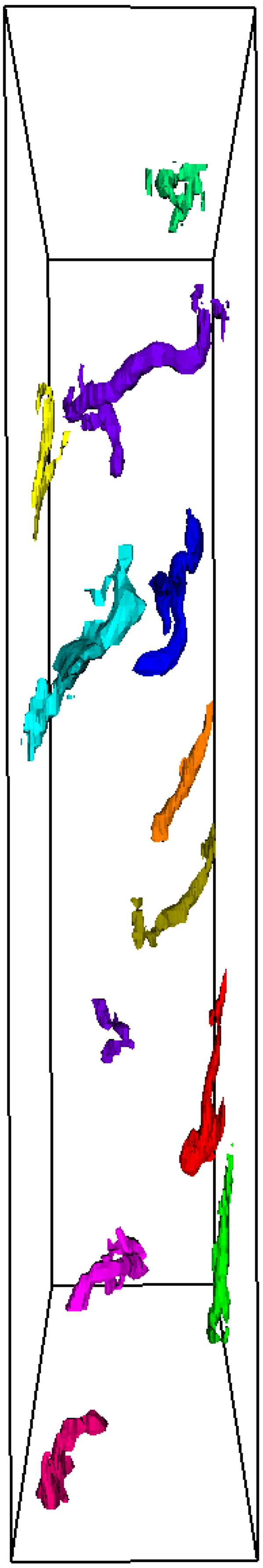

(C) Step 9

FIGURE 5.22: Tracking of selected features in the turbulent plane Couette flow.

5.2.4 Selected features in the fully-developed stage of the Taylor Green Vortex

We are going to consider again the time evolution of the TGV field at the fully-developed stage. In contrast to Subsection 5.2.1, we will now visualize the entire field, instead of a clipped version of it. Figure 5.23 contains the contour tree of a time snapshot, along with the field where vortex structures corresponding to upper leaf arcs are displayed. Figures 5.24 show the temporal evolution of randomly chosen features of the field. An apparently arbitrary movement behavior can be observed for all features. In addition, in step 5, we can see how the black, beige, brown and magenta vortex structures are integrated in larger x-shaped features. This happens because their corresponding contour tree regions are pruned, so their individual identity is lost. We observe how the x-shaped structures are successfully tracked over the next steps.

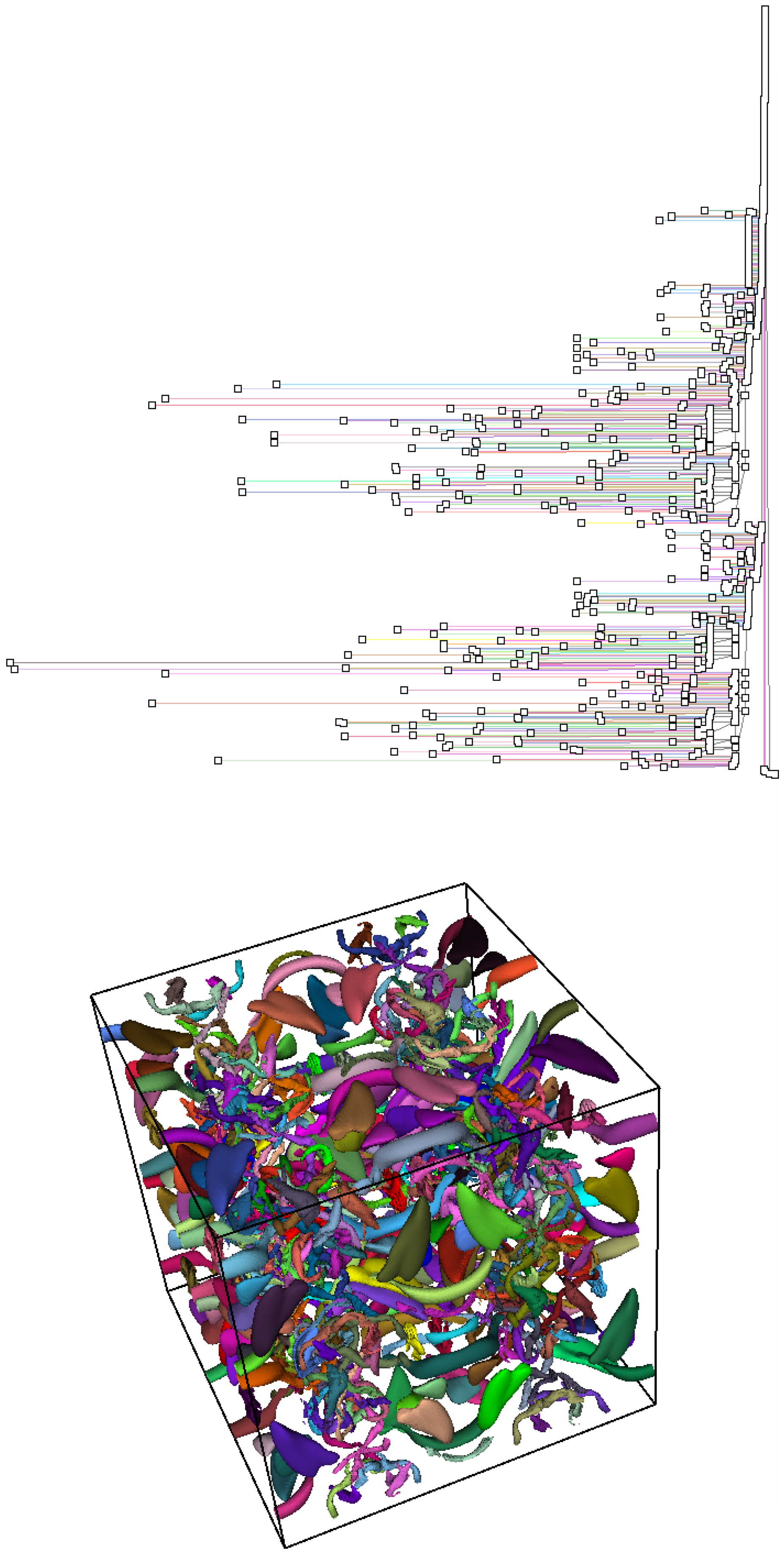
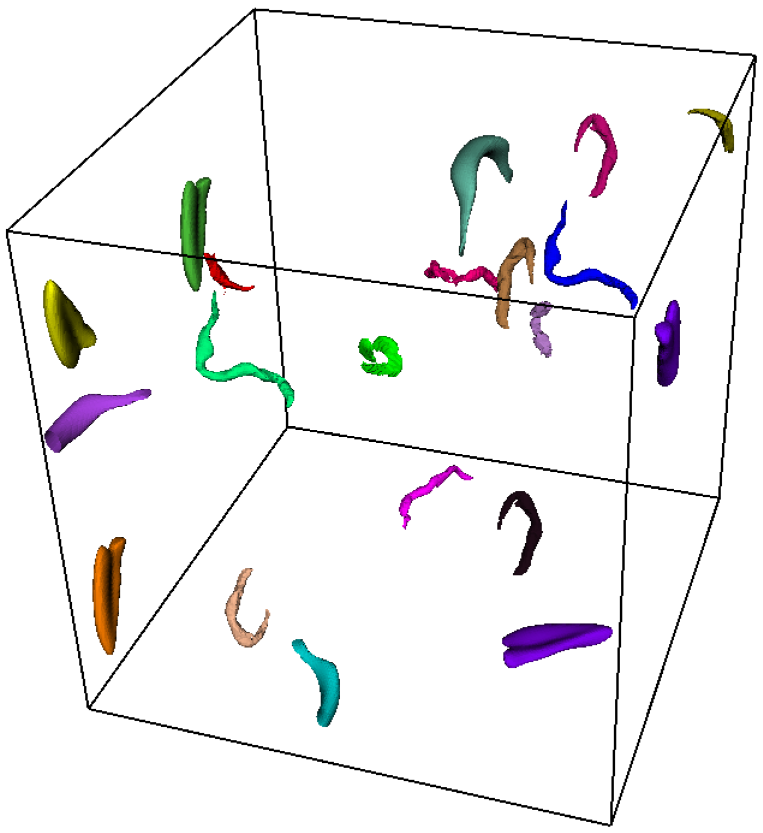
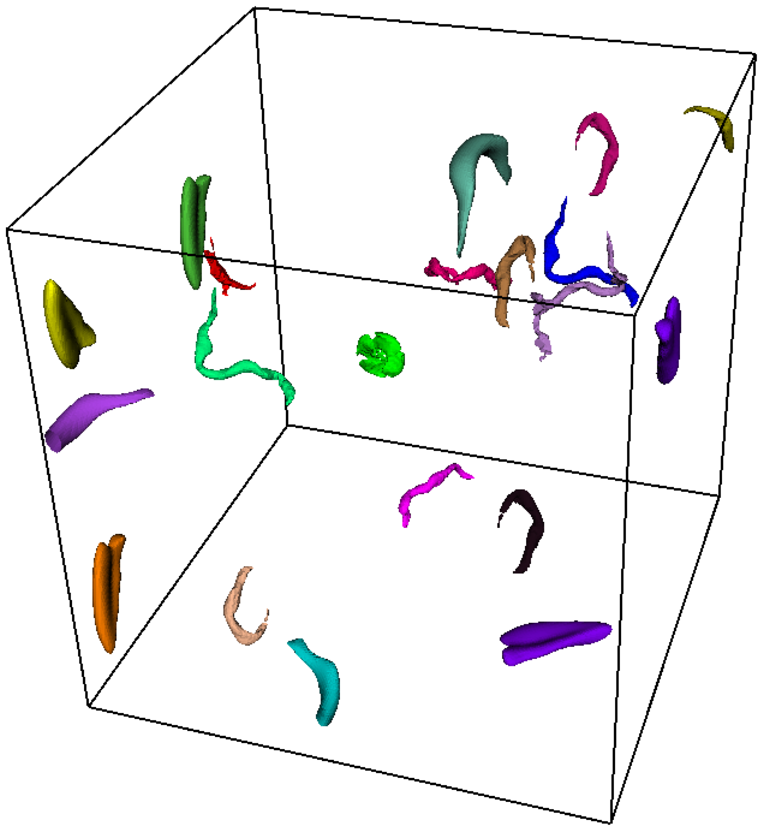
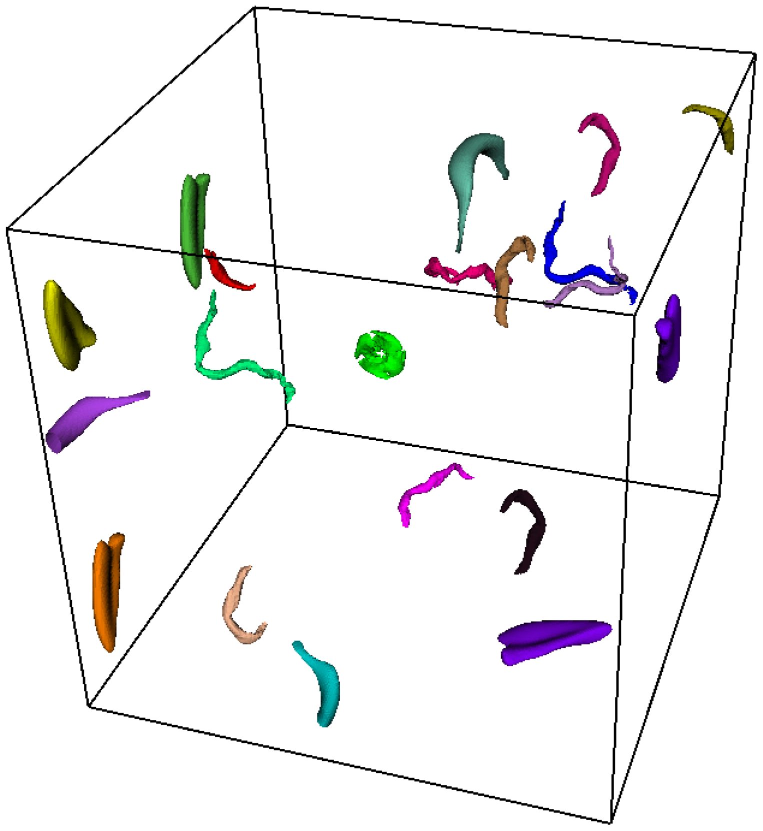
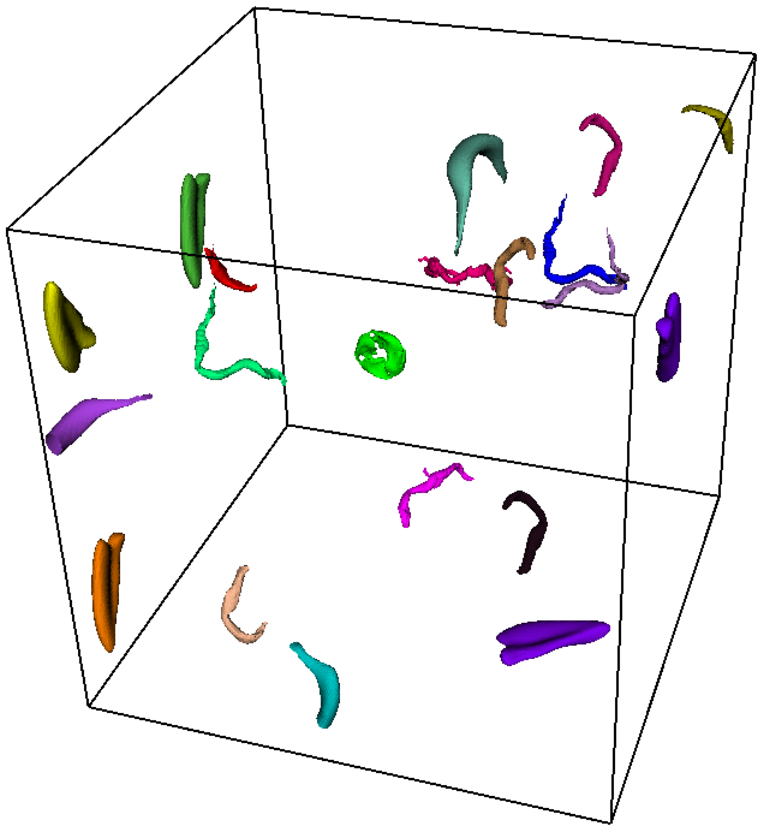


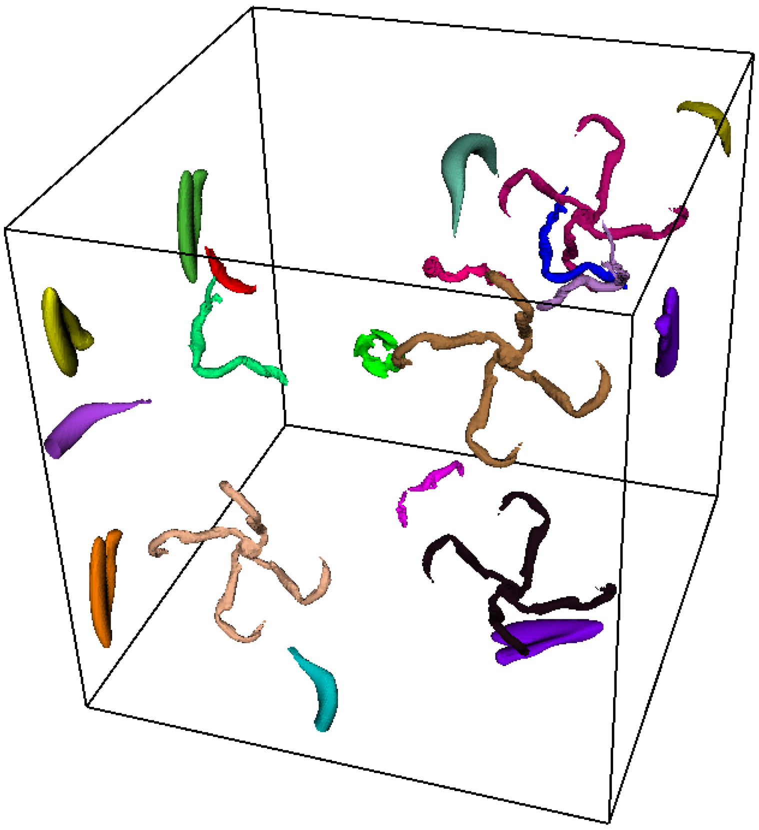
FIGURE 5.23: Vortex structures observed in the Taylor Green Vortex flow.

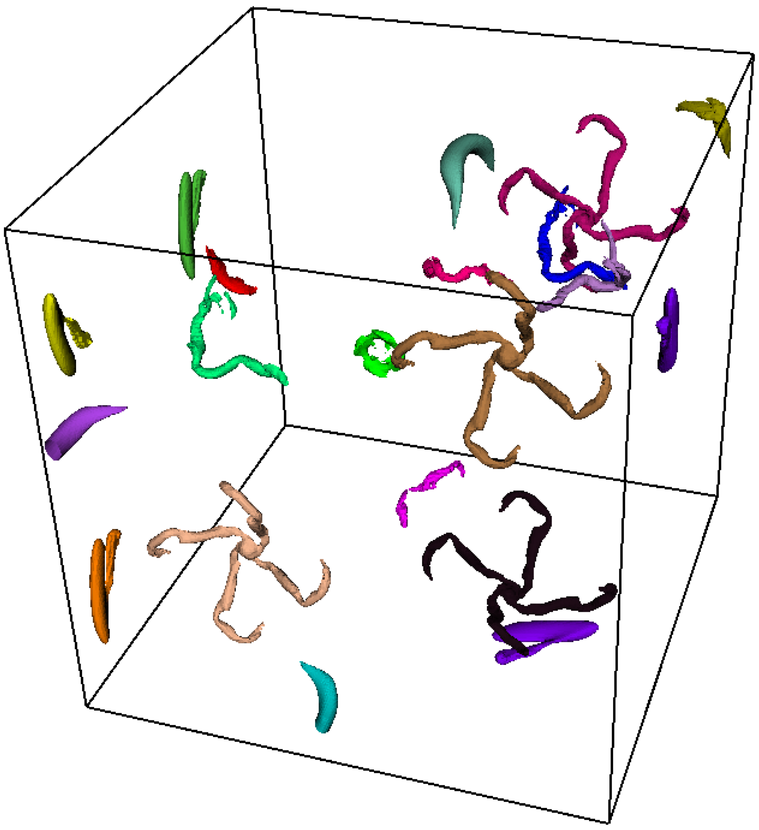

(A) Step 1

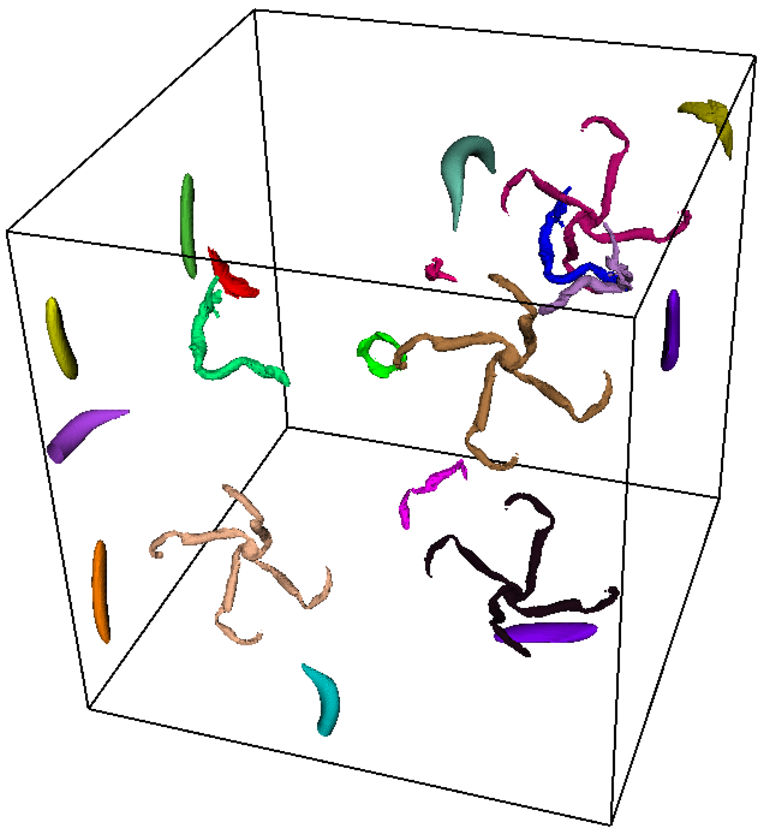

(B) Step 2


(C) Step 3

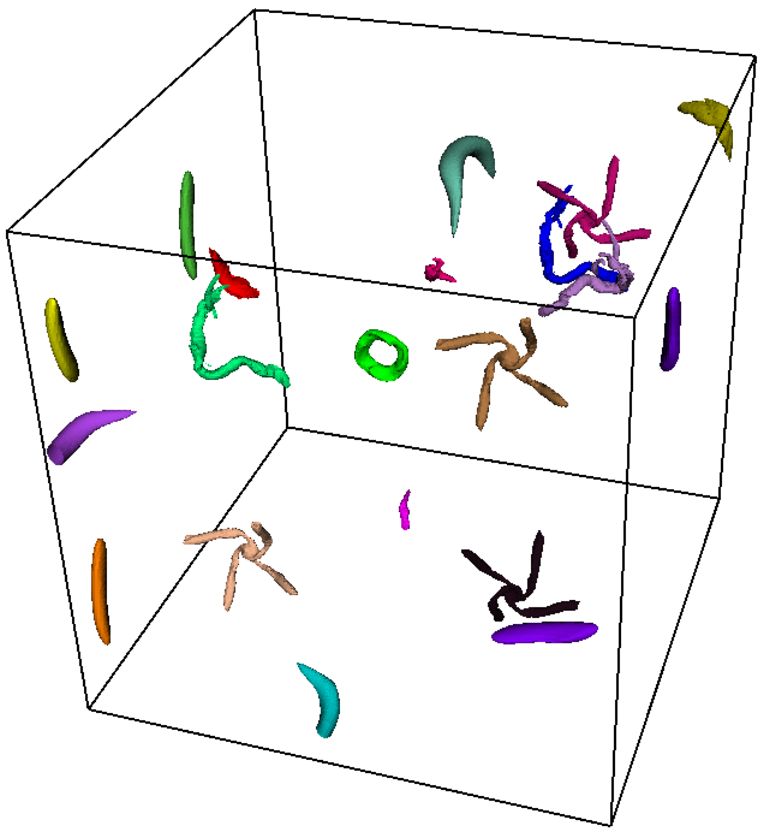

(D) Step 4


(E) Step 5


(F) Step 6



(G) Step 7



(H) Step 8

FIGURE 5.24: Tracking selected features at the Taylor Green Vortex fully-developed stage.

5.3 Feature tracking metrics

In this section, we are going to statistically present a number of metrics related to the feature tracking algorithm. We will also consider the impact of parameters such as the time distance between fields, or the value of the threshold used to define the significance of the spatial overlap between features. As testing data, a sequence of consecutive steps of the turbulent plane Couette flow was used.

We start by an analysis of the number of contour tree arc regions with different labellings that exist in the output of the contour correspondences calculation step. In theory, each voxel added to an object on an arc could change the set of labels, so in the worst case the total number of arc divisions could be similar to the number of vortices in the mesh. Figure 5.25 presents a histogram showing the frequency distribution of arcs with respect to the number of labeling regions they are divided into. Note that only regions with at least one label are counted for an arc. The figure was obtained for two consecutive fields, using a match threshold value of 0.7, where the second contour tree contains 617 arcs. We can see that most of the arcs are divided in very few regions, due to the coherency of the analyzed data (on average, each arc contains 2.184 regions). Also, interestingly, it can be observed that there is no arc which has no labeled contour. In contrast, when the contour correspondences were computed between two fields with 10 intermediate steps skipped, at the same match threshold, 3 unlabeled arcs appeared.

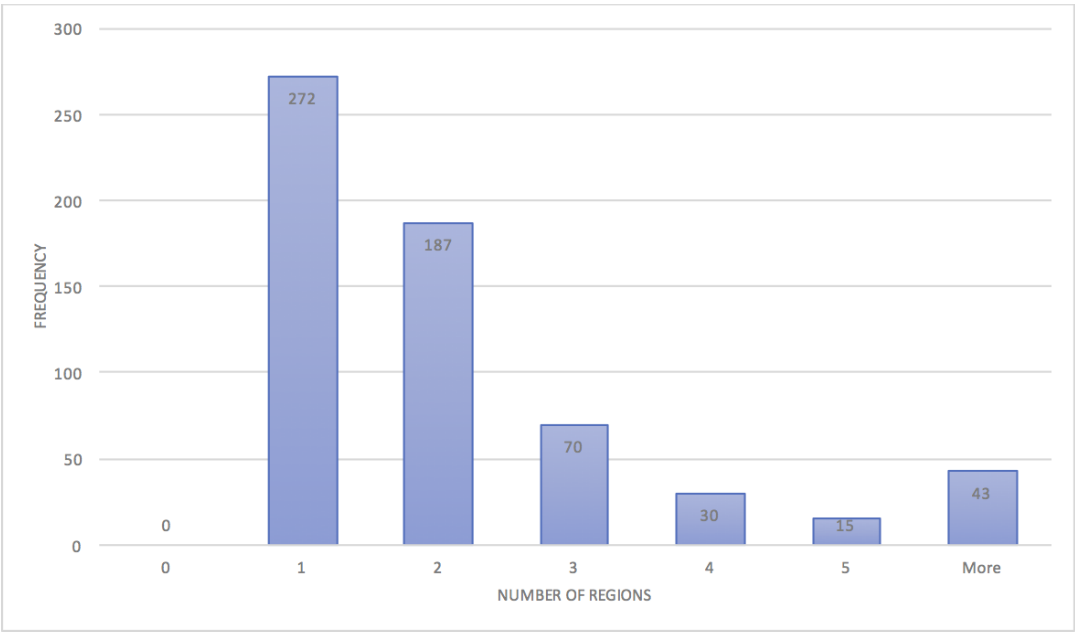


FIGURE 5.25: The frequency distribution of arcs with respect to the number of regions labeled differently.

We are now going to focus on metrics around the number of edge correspondences identified between time snapshots. Again, the Couette flow simulation was used. We started by looking at the percent of edges from the first contour tree which are continued in the next time step, in the sense that they have at least one correspondent edge in the second contour tree. For consecutive time steps of the simulation, the average continuation rate was 71.83% for all arcs, and 97.41% for upper leaf arcs, for the significance threshold value set to 0.8. Although the upper leaves are almost perfectly matched, we can see that the percentage of join arcs matched is significantly lower. This is a consequence of the sensitivity of the contour tree structure to the motion of features inside the field. If two features are currently having a direct join arc, but at the next step they are joining with other arcs instead, the track of their joining feature will be lost.

We have seen that the upper leaves are almost perfectly matched between consecutive time steps. We are going to analyze the same metric when the temporal distance between the snapshots considered for analysis is larger. In order to simulate a larger temporal gap between consecutive fields, we are going to skip a number of intermediate available steps. The influence of the number of skipped steps on the percent of continued arcs can be seen in Figure 5.26. It can be observed that the continuation rate is relatively high (0.853 for upper leaves), even when 19 steps are skipped. However, skipping more steps leads to a steep decrease in the continuation rate.

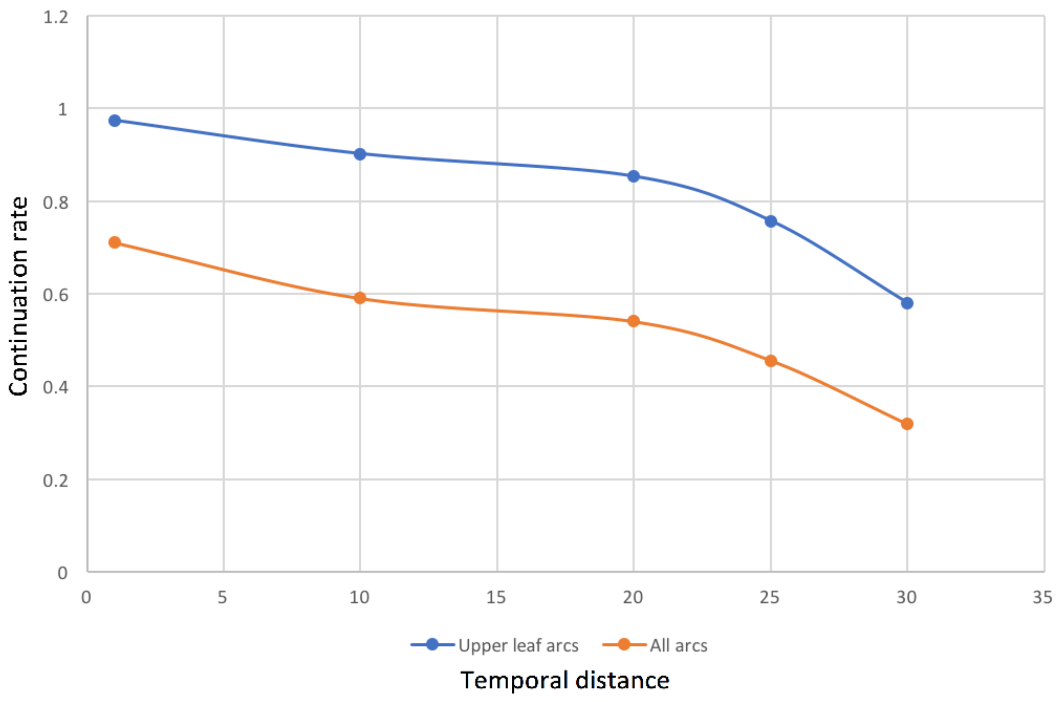


FIGURE 5.26: The continuation rate of the arcs against the temporal gap.

Naturally, the number of arc correspondences identified is dependent on the value of the match threshold used. We used consecutive time steps at distance of 20, belonging to the plane Couette flow, in order to assess the impact of the significance threshold value on the same continuation rate metric. The resulting plot is included in Figure 5.27. We can see how the continuation rate is quickly decreasing between the match thresholds 0.8 and 0.9.

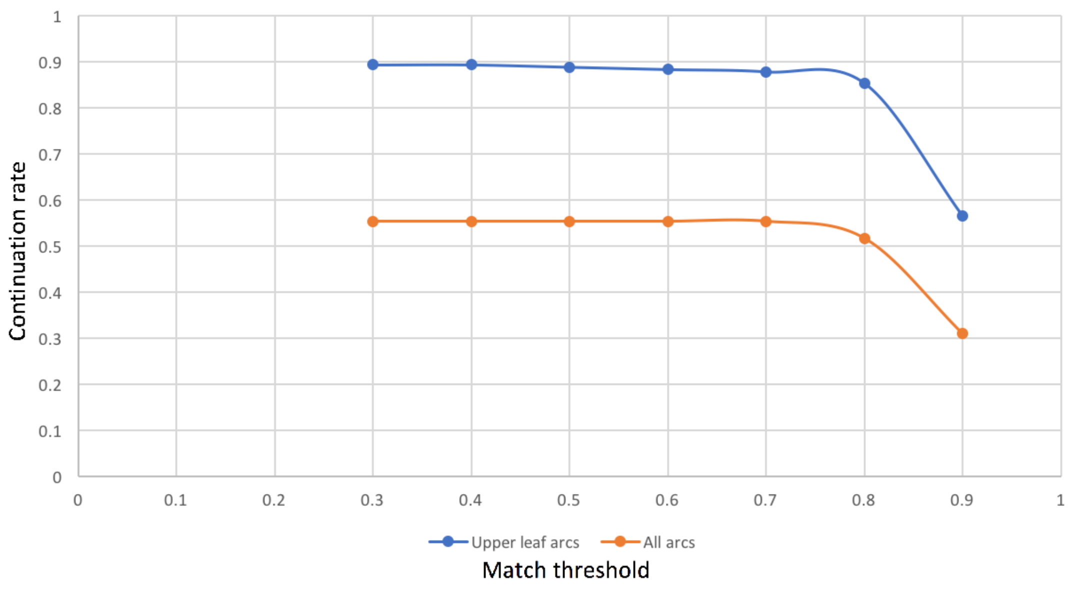


FIGURE 5.27: The continuation rate of the arcs against the match threshold.

5.4 Performance analysis

In this section, we are going to focus on the time and memory requirements of the implemented feature tracking algorithm. The used test data is a sequence of 25 consecutive steps of the turbulent plane Couette flow, whose mesh contains 1010101 voxels and 979200 cells. The obtained results represent averages of 24 runs of the feature tracking algorithm, one for every pair of consecutive snapshots in the considered sequence. The benchmarks included in this section were run on a MacBook Pro laptop running macOS Sierra. The laptop had a 2.5 GHz Intel Core i7 quad-core processor, and 16GB DDR3 RAM. The code was compiled with g++, with the O3 level of optimizations enabled.

We start by looking at the run time of the algorithm against meshes of different sizes. In order to ensure the coherency between the tests used, fields of different sizes were generated by taking subregions of the plane Couette flow mesh. With respect to the number of initial vertices in the mesh, the 5 different subfields generated represent 20%, 40%, 60%, 80% and 100%. The resulting plot can be seen in Figure 5.28. We can observe a non-linear tendency in the run time, which is expected as a consequence of the operations with quadratic complexity performed in the algorithm for computing contour correspondences.

Table 5.1 illustrates the time spent for each operation of the algorithm in the case of the entire Couette flow, leading to an averaged total of 20.99 seconds for one computation of arcs correspondences which is also visible in the previous plot. We can see that the step of building an upper objects tree that reflects the significant overlaps between the objects of two fields is the most expensive part of the algorithm, representing 66% of the total run time. It can also be observed that, in the case of the first three operations, the

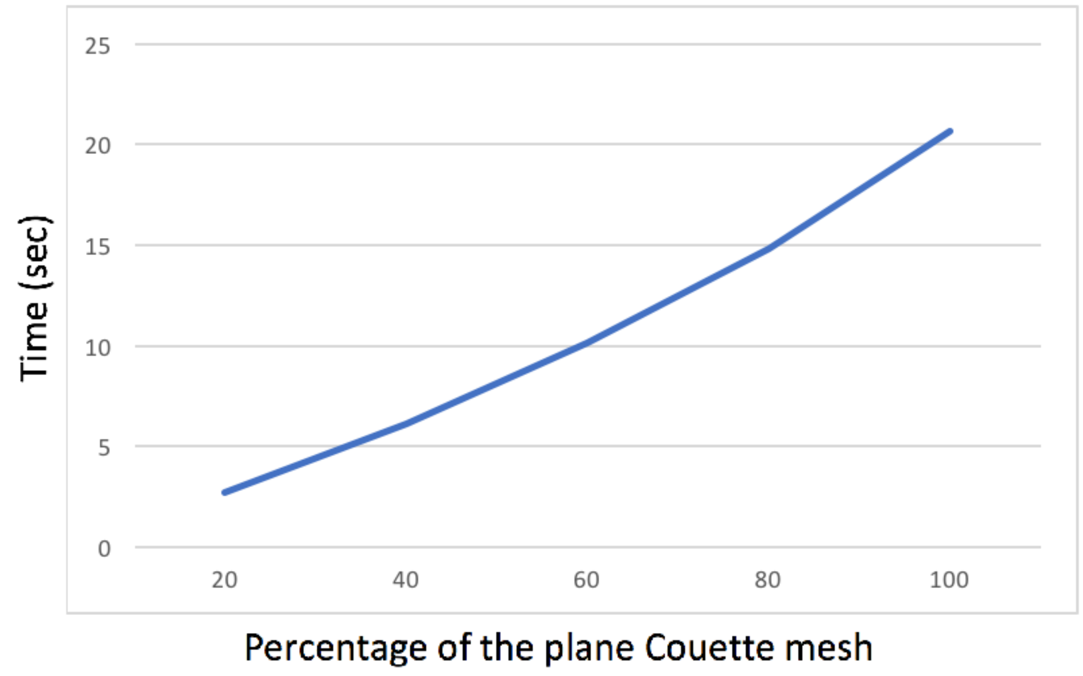


FIGURE 5.28: The run time of the algorithm against meshes of different sizes.

differences between the maximum and the minimum times are significant. This is explained by the much larger number of objects that are handled in the operations concerned with join trees, compared to the ones on split trees. The difference in the number of upper objects between the two symmetric cases is a consequence of the join tree dominating the contour tree for positive Q-criteria in the case of the plane Couette flow.

All the above results were obtained after applying two pruning criteria: pruning the leaves with volumes smaller than 150, and pruning the negative values. When the entire mesh of the plane Couette flow was considered, the obtained contour trees of the sequence had, on average, 406 nodes. However, in general, the amount of pruning we apply to the field has the potential to significantly influence the algorithm runtime. Figure 5.29 illustrates this aspect, by showing the total runtime of the algorithm against the number of nodes remaining in the contour tree after this is pruned. Therefore, the total number of vertices is constant (1010101), but the size of the contour tree and the number of pruned vertices are varied.

In order to analyse the memory usage of the algorithm, we used the Valgrind Massif [30] heap profiler. We measured the memory usage of a program run consisting of the following sequence of events: reading the contour trees from files, constructing the local grids for each arc, and computing the feature tracking correspondences. Therefore, the step of building and simplifying contour trees was skipped, in order to focus on the performance of the steps related to feature extraction and correspondences calculation. The program flow considered 5 consecutive steps belonging to the plane Couette

TABLE 5.1: Time spent to compute the feature correspondences between a pair of time steps of the plane Couette flow. Time is in seconds. Averaged across 24 steps.

Operation	Executions per time step	Average time	Minimum time	Maximum time	Total time
Upper objects tree construction	4	0.807	0.727	1.240	3.231
Upper objects tree construction with collisions	2	6.943	6.688	7.154	13.887
Labelling contour trees from upper object trees	2	0.161	0.015	0.323	0.323
Intersecting the join and split contour trees labels	1	3.426	3.379	3.473	3.426
Obtaining arc correspondences	1	0.132	0.117	0.153	0.132

flow, each *.vtu* file having the size of 141MB. The output of the profiler is included in Figure 5.30a. Figure 5.30b includes the graph corresponding to a second run of the algorithm, where the step of building the local grid for arcs was also skipped.

We can see how skipping the local grids construction reduced the peak memory usage from 2.798GB to 1.151GB. The two graphs show a steep memory usage increase while the contour trees, the field values, and the connectivity graph between vertices are read, and, in the first graph, while the local grids for arcs are computed. They represent the memory defining the baseline for the evaluation of the feature tracking algorithm. It can be seen that the peak memory usage indeed occurs during the calculation of the correspondences between the 4 pairs of consecutive fields, due to the fact that the baseline memory will only be deallocated at the end of the program. The memory usage during the execution of the feature tracking algorithms is fluctuating, as a result of memory being constantly deallocated after the operations complete. In both graphs, the peak memory usage of the feature tracking algorithm appears to be around 200MB, and the rest of the profiling information shows that it occurs during a construction of one join tree that reflects significant overlaps, the algorithm's operation that involves the largest number of data structures.

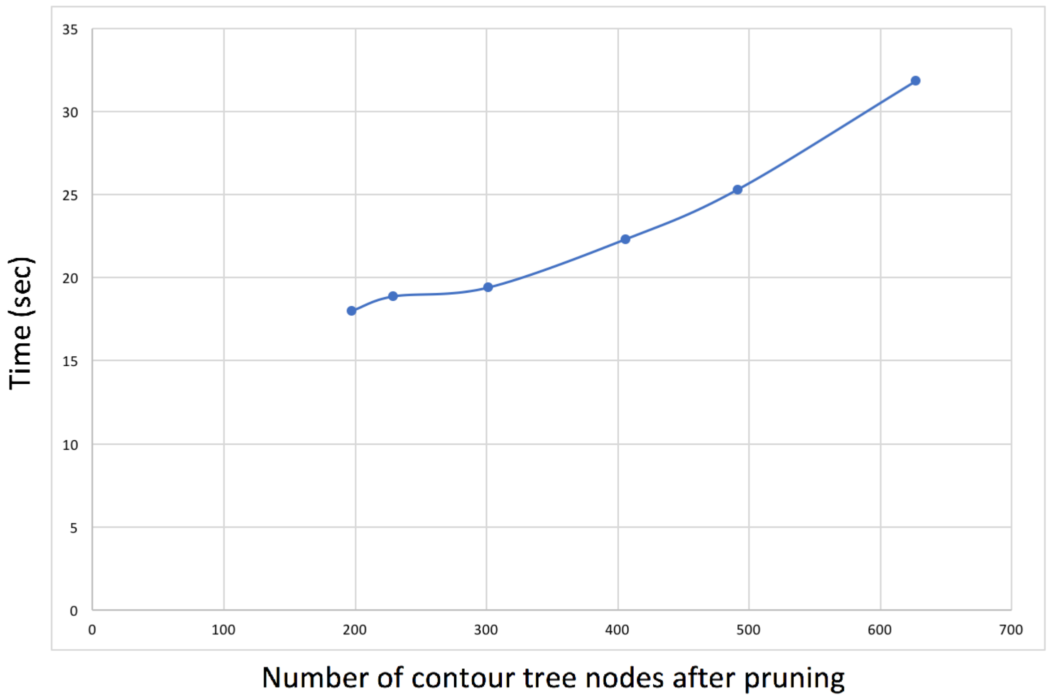
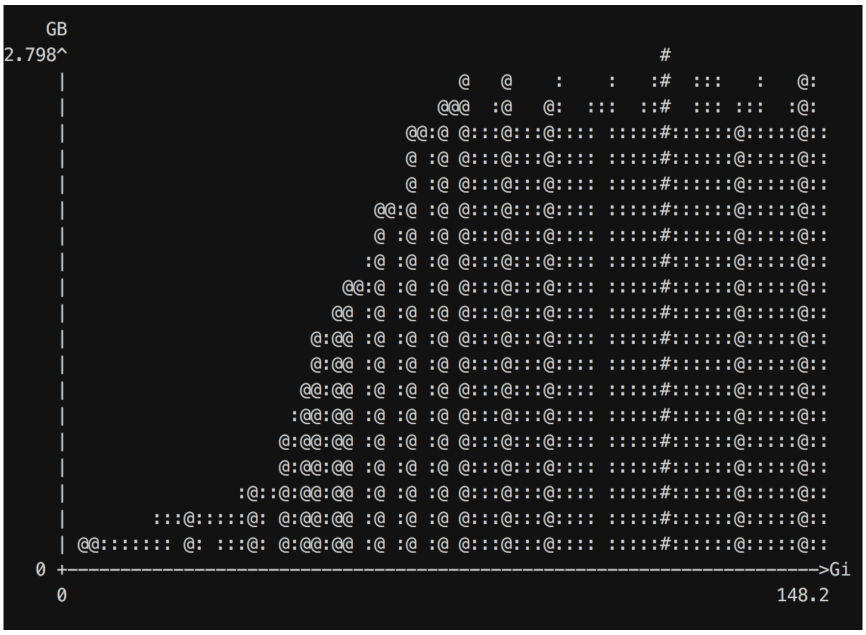
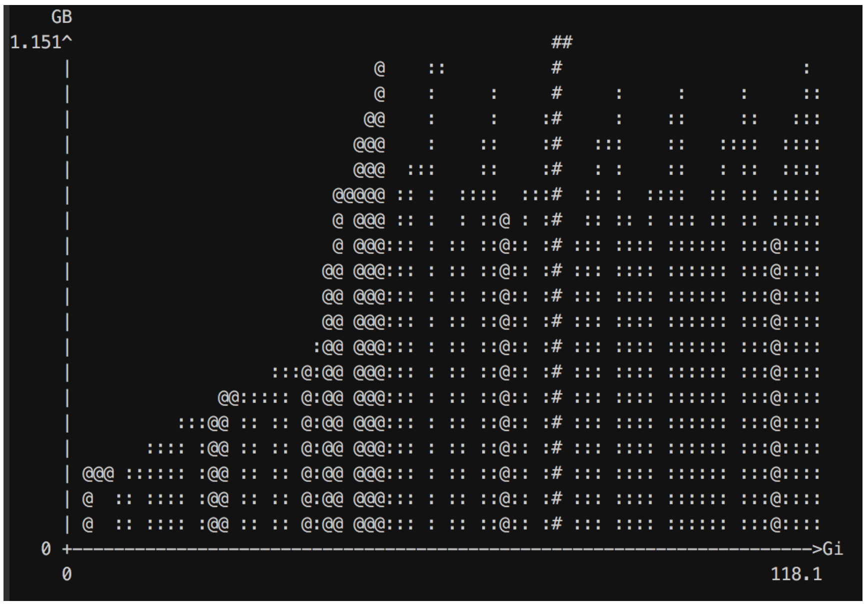


FIGURE 5.29: The run time of the algorithm against different contour tree sizes.



```
    GB
2.798^                                                        #
     |                                     @   @    :    :   :#  :::   :   @:
     |                                   @@@  :@   @:  :::  ::#  ::: :::  :@:
     |                                @@:@ @:::@:::@:::: :::::#::::::@:::::@::
     |                                @ :@ @:::@:::@:::: :::::#::::::@:::::@::
     |                                @ :@ @:::@:::@:::: :::::#::::::@:::::@::
     |                             @@:@ :@ @:::@:::@:::: :::::#::::::@:::::@::
     |                             @ :@ :@ @:::@:::@:::: :::::#::::::@:::::@::
     |                            :@ :@ :@ @:::@:::@:::: :::::#::::::@:::::@::
     |                          @@:@ :@ :@ @:::@:::@:::: :::::#::::::@:::::@::
     |                         @@ :@ :@ :@ @:::@:::@:::: :::::#::::::@:::::@::
     |                       @:@@ :@ :@ :@ @:::@:::@:::: :::::#::::::@:::::@::
     |                       @:@@ :@ :@ :@ @:::@:::@:::: :::::#::::::@:::::@::
     |                      @@:@@ :@ :@ :@ @:::@:::@:::: :::::#::::::@:::::@::
     |                     :@@:@@ :@ :@ :@ @:::@:::@:::: :::::#::::::@:::::@::
     |                    @:@@:@@ :@ :@ :@ @:::@:::@:::: :::::#::::::@:::::@::
     |                    @:@@:@@ :@ :@ :@ @:::@:::@:::: :::::#::::::@:::::@::
     |                :@::@:@@:@@ :@ :@ :@ @:::@:::@:::: :::::#::::::@:::::@::
     |        :::@:::::@: @:@@:@@ :@ :@ :@ @:::@:::@:::: :::::#::::::@:::::@::
     | @@::::::: @: :::@: @:@@:@@ :@ :@ :@ @:::@:::@:::: :::::#::::::@:::::@::
   0 +----------------------------------------------------------------------->Gi
     0                                                                   148.2
```

(A)



```
    GB
1.151^                                              ##
     |                             @    ::          #                       :
     |                             @    :     :     #     :     :     :     ::
     |                            @@    :     :    :#     :    ::     ::   :::
     |                           @@@    :    ::    :#   :::    ::   ::::  ::::
     |                           @@@  :::    ::    :#   : :    ::   : ::  ::::
     |                         @@@@@ :: :  ::::  :::#  :: :  ::::  :: :: :::::
     |                         @ @@@ :: :  : ::@ : :#  :: :: : ::: :: :: :::::
     |                         @ @@@::: : :: ::@:: :# ::: :::: :::::: :::@::::
     |                        @@ @@@::: : :: ::@:: :# ::: :::: :::::: :::@::::
     |                        @@ @@@::: : :: ::@:: :# ::: :::: :::::: :::@::::
     |                        @@ @@@::: : :: ::@:: :# ::: :::: :::::: :::@::::
     |                       :@@ @@@::: : :: ::@:: :# ::: :::: :::::: :::@::::
     |                   :::@:@@ @@@::: : :: ::@:: :# ::: :::: :::::: :::@::::
     |              @@::::: @:@@ @@@::: : :: ::@:: :# ::: :::: :::::: :::@::::
     |          :::@@ :: :: @:@@ @@@::: : :: ::@:: :# ::: :::: :::::: :::@::::
     |       :::: :@@ :: :: @:@@ @@@::: : :: ::@:: :# ::: :::: :::::: :::@::::
     | @@@ :::::: :@@ :: :: @:@@ @@@::: : :: ::@:: :# ::: :::: :::::: :::@::::
     | @  :: :::: :@@ :: :: @:@@ @@@::: : :: ::@:: :# ::: :::: :::::: :::@::::
     | @  :: :::: :@@ :: :: @:@@ @@@::: : :: ::@:: :# ::: :::: :::::: :::@::::
   0 +----------------------------------------------------------------------->Gi
     0                                                                   118.1
```

(B)

FIGURE 5.30

5.5 Comparison to other feature tracking methods

In this section, we provide a high-level comparison of the developed feature tracking algorithm with other methods existing in the literature. Our approach is based on the algorithm for computing isosurfaces proposed by Sohn et al. in [39], which we improve to support field simplification procedures, and extend with a way of converting the contour correspondences to correspondences between contour tree arcs. Therefore, the algorithm we propose is based on spatial and field value ranges overlap, as well as on the nesting relationship of the features.

The region-based approaches, such as the ones described in [34] and [37], are effective in detecting the continuation of features across fields. However, as described in Subsection 2.9.1, detecting splits and joins between features (i.e. bifurcations and amalgamations) comes with extra complexity. In theory, an exponential number of combinations have to be considered in order to detect these events. However, in practice, certain heuristics can be applied in order to limit the search space. In contrast, our algorithm has a guaranteed polynomial complexity. Also, due to the nesting knowledge given by the contour tree, we produce tracking information for the entire field, while these approaches only focus on the flat hierarchy of features. On the other hand, in contrast to the region-based approaches, the current algorithm is not able to detect continuation of features whose scalar value ranges are disjoint in two consecutive time steps. Although representing an additional restriction, this is not an issue under the assumption that the temporal sampling is sufficient. Moreover, if this assumption holds, a drastic change in the value range should not occur for one feature, so this could lead to the conclusion that two features should not be matched. Another overlap based approach is described in [45], where contour trees are used for feature extraction. The features corresponding to contour tree arcs are then pairwise tested for spatial overlap, and the correspondence information is accumulated down the contour tree.

In [18], the idea proposed is to use 4D isosurfacing for tracking field features, where one dimension is the time. In [32], the correspondences between consecutive time steps are calculated using the global knowledge provided by all available field snapshots. Due to their static nature, these approaches lack the capability of being integrated in-situ. In contrast, we are going to demonstrate the feasibility of integrating our tool in-situ, in Section 7.3.1.

In [32], Saikia et al. highlight the sensitivity to noise of the approaches based on Feature Flow Fields, proposed in [41] and [44]. Although topological simplifications of the input fields are feasible, there is currently no work evaluating this idea. We demonstrated how our feature tracking algorithm can be applied on data sets simplified by the means of contour tree pruning.

5.6 Tool evaluation

Sections 5.1 and 5.2 have demonstrated the capability of the tool to visualize and track features across multiple snapshots of a time-dependent scalar field. Also, in Section 5.4, we have seen the memory usage graph of one of the common flows of the tool - reading the contour trees and the scalar values, building the local grids for the contour tree arcs, computing the feature tracking information, and launching the graphical interface.

From the perspective of the user, the graphical interface has the following capabilities:

- The ability to interact with the contour tree, and to select the desired features for visualization.
- The ability to hover over contour tree nodes, in order to see their associated scalar value.
- The ability to resize the two screen regions containing the field visualization and the contour tree, respectively, by dragging.
- The ability to change the aspect ratio of the contour tree, by a key press.

On the other hand, it can be seen that the tool lacks other data manipulation features, such as displaying cutting planes, coloring features by field properties, etc. However, during this project, we decided to focus more on the feature tracking algorithm behind the visualization, while the addition of further functionality in the tool's user interface could be the subject of another project.

5.7 Summary

This chapter presented an analysis focused on the feature tracking algorithm and on the tool implemented as part of this project.

Evaluating the tracking algorithm was done using a number of demos of increasing difficulty, representing both artificial test data and fluid flows corresponding to CFD simulations. In addition, we presented a number of metrics, which quantify the impact of the temporal sampling of the data, as well as the impact of the overlap significance constant threshold set. A performance analysis of the algorithm was also conducted on test data of different sizes. Section 5.5 concluded the analysis of the tracking algorithm, by presenting a high-level comparison of it with other techniques existing in the literature.

Chapter 6

Analyzing streaks in the turbulent plane Couette flow

6.1 Motivation

The analysis of turbulent flows in the presence of walls represents an important aspect in the field of fluid dynamics, as this kind of environments are highly relevant for practical engineering problems: designing quieter aircrafts with lower fuel consumption [17] or optimizing the shape of a wind turbine blade [48] are examples of industry problems which frequently involve well-developed turbulent flows. In order to make progress in understanding these environments, researchers proposed a number of canonical flows that are subject to relatively simple boundary conditions. Examples of canonical wall bounded flows include the Couette flow, the pressure driven channel flow, and the Poiseuille flow.

The formation of turbulence in these flows has been an elusive problem for a long time and extensive studies have been performed in order to understand the transition process from the laminar boundary layer to the turbulent boundary layer, as well as turbulent statistics associated with well-developed turbulent flows.

One of the distinctive features observed in turbulent wall-bounded flows is the formation of streaky patterns in the streamwise velocity component in the near-wall region. Some of the earliest observations about these motions were made by Kline et al. in [20], but have been confirmed by multiple further studies [7]. Streaks represent an alternating pattern of regions with high or low streamwise velocity: an example can be seen in Figure 6.1. The formation of streaks is considered to be the result of the *lift-up tendency* of vortices after they appear in the immediate proximity of the wall, tendency explained by the energy transfer between the vortices and the base flow [16]. In [20], Kline et al. demonstrate the contribution of low-velocity streaks in the production of turbulent kinetic energy. They observe the violent ejection of fluid with low streamwise velocity from the regions very near to the wall. They then suggest that the instability produced by these fluid bursts (i.e. bursting process) is essential for the transportation of kinetic energy to the outer regions of the boundary layer, producing turbulence.

Another important research aspect in turbulent flows are coherent vortex

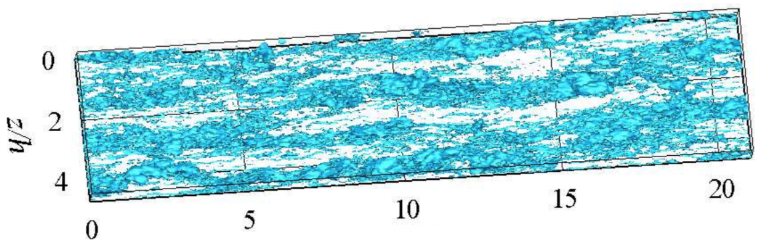


FIGURE 6.1: Visualization of the streamwise velocity field in a channel flow [16]

structures which can be identified across wider ranges of flows. The identification of coherent vortex structures represents an attempt of the researchers to break down the chaotic motion that characterizes turbulence into more elementary, organized motion. Two important examples of coherent structures are hairpin vortices and (quasi)streamwise vortices [1]. The former are characterized by their hairpin (or horseshoe) shape, and are often easily observed during the transition stage of the flow from the laminar to the turbulent state. The streamwise vortices are characterized to be long and thin structures, that are moving in the direction of the mean flow. Although coherent structures play a key role in the theoretical understanding of many flows, in practice they are easily disrupted by noise, and often absent from actual flows. Moreover, in [1], Adrian et al. state that "only motions that live long enough to catch our eye in a flow vizualization and/or contribute significantly to time-averaged statistics of the flow merit the study and attention we apply to organized structures".

The analysis of streaks has been conducted using the method of spatial filtering [5], which supports the understanding of the flow motions using global statistics. One such study is [23], where Lee et al. explore the population-level properties of large- and very large-scale motions in streaks. In [6], Brandt et al. analyse the streak interactions by the means of conditional sampling, also by looking at their statistical properties. However, none of these methods focuses on individual vortex structures, but rather on spatial or temporal global averages. On the other hand, in a context different from streak analysis, Choi et al. compute in [12] Lagrangian statistics of a turbulent channel flow by tracking a number of particles in a direct numerical simulation. They release many particles at several wall-normal locations and track them spatially and temporally, quantifying the evolution of properties associated with each fluid particle.

This chapter aims to highlight the value of tracking flow features for streak analysis. Motivated by the Lagrangian statistical analysis in [12], we will consider vortex structures instead of fluid particles. We will focus on observing how the motions of individual vortices contribute to maintain the streak phenomena in the turbulent flow. This provides additional insight into the mechanism of maintenance and evolution of the streak structure, in comparison to the global statistical methods employed by [23] and [6]. We demonstrate the capability of tracking the motion of vortices that have certain spatial properties, with the streamwise vortices being focused throughout the chapter. The tool we propose can be used to investigate the contribution of streamwise vortices included in low streamwise velocity streaks to the evolution of the turbulent plane Couette flow. The first question we would like to answer is the existence of streamwise vortices in low-velocity streaks. Then, we would be interested in observing the motion of these structures. From the best of our knowledge, no researcher previously looked at the individual contribution of each streamwise streak vortex to the evolution of the streaks and to the enhancement of turbulence in flows.

We chose to focus on streamwise vortices, as they usually preserve their shape better than hairpins in the fully developed stages of turbulent flows. Detecting features with other specific shapes or properties (such as hairpins) could be the subject of a different project. Here we want to highlight how tracking vortex structures with different properties can potentially lead to a better understanding of certain flows.

In the rest of the chapter, section 6.2 aims to explain the filtering method used to detect the features of interest, while section 6.3 presents the results of running the tool on a sequence of steps from the turbulent plane Couette flow. Note that details about the Couette flow are given in Appendix A.

6.2 Filtering vortices

As mentioned above, the vortex structures chosen for analysis are characterized by their velocity vector values, orientation and elongated shape. In order to assess this, we propose a *FeatureFilter* that performs the following steps:

1. Compute the centroid of the local grid defining the feature. The distance from the centroid to the closest moving wall should be between a lower and an upper bound. This ensures that the feature is in the flow region where the streaky pattern is visible (see Figure 6.1).

2. Average the velocity vector x (i.e. streamwise direction) components of the voxels belonging to the mesh region defining the feature, in order to obtain a characteristic velocity representing the streamwise speed of the vortex structure. In order to remove features belonging to high velocity streaks, the absolute value of this velocity should be large. This is because the low-velocity condition for flow regions considers the streamwise velocity relative (not absolute) to the mean velocity profile of the flow [16], which, for simplicity, we assume to be the linear (laminar) profile. Since the velocity of the features is high in the proximity of the wall, the absolute velocity of the current feature is expected to be large as well. In other words, the large streamwise velocity in the present flow field is corresponding to the low speed streaks based on the reference frame associated with the moving wall.

3. Extract the surface of the vortex, using a VTK ContourFilter. This is done because, when assessing the spatial properties of the features, we want to consider the shape of one isosurface that defines it, rather than its entire corresponding mesh region.

4. Perform a 3D orthogonal distance regression on the points belonging to the surface, in order to obtain the 3D line which minimizes the sum of the squared distances from the points to the line. Intuitively, in the case of an ideal streamwise vortex, this line should pass through the center of the shape.

5. Compute the projections from the points defining the surface on the regression line just computed. Using the projections, also compute the distances from the points to the line. After computing the distances:

   - Compute their average, $dist_{avg}$. Also compute the distance between the furthest two projections $dist_{max}$. In order to enforce the elongated shape of the vortex, we want the ratio $\frac{dist_{max}}{dist_{avg}}$ to be large.
   - Compute their standard deviation, $dist_{sd}$. In order to enforce a regular shape of the surface, we want the ratio $\frac{dist_{sd}}{dist_{avg}}$ to be small.

6. Consider the normalized direction vector d of the line. In order to enforce the bulk flow orientation of the feature, we check that the d_y and d_z vector components are close to 0.

6.2.1 Performing an orthogonal distance regression

An orthogonal distance regression computes the best-fit line which minimizes the sum of the squared distances from the input points to it. Therefore, given a set of points in space of the form $\{p_1, p_2, ..., p_n\}$ where the elements $p_i = (x_i, y_i, z_i)$ are 3D points, the computed line l minimizes the value of the following expression:

$$\sum_{i=1}^{n} distance^2(p_i, l)$$

where $distance$ refers to the 3D orthogonal distance between a point and a line.

The computation of the best-fit line is done using the idea presented in [27] and consists of the following steps:

1. Compute the centroid of the set of points. The best-fit line will pass through the centroid.

2. Subtract the centroid from the points, and with the resulting points build the 3 x N matrix where each column corresponds to one of the points.

3. Calculate the singular value decomposition of this matrix. The least singular value will correspond to the normal on the best-fitting plane.

By taking the perpendicular vector on this direction, we obtain the direction of the best-fit line.

4. We know a point on the line (the centroid), and its direction vector. Therefore, we managed to express the 3D regression line.

In our implementation, we used the singular value decomposition implementation available in the NumPy library [14]. This was used from our C++ code using a system call.

6.2.2 Choosing the thresholds

Most of the filtering conditions included earlier in the section are expressed in terms of specific thresholds. These were computed using a trial and error approach in order to obtain sensible results, as in the literature there was no attempt to strictly define the shape parameters of coherent structures.

The results presented in the next section use the following thresholds, for each filtering step:

- Step 1: The centroid should be between 0.15 and 0.25. This is the wall-distance interval where the streaky pattern is visible. In addition, we only consider the streaks from the bottom part of the flow. Similar results would be obtained if we considered the regions at the same wall-normal distance to the other moving wall.
- Step 2: The absolute average velocity of the features should be larger than 0.1.
- Step 5: The ratio $\frac{dist_{max}}{dist_{avg}}$ should be larger than 15, and the ratio $\frac{dist_{sd}}{dist_{avg}}$ should be lower than 0.5.
- The absolute value of d_y and d_z in the normalized direction vector should be lower than 0.1.

6.3 Results

The first step was to select features satisfying the first two filtering steps described in 6.2. Therefore, all vortex structures belonging to low-velocity streaks should be detected. The result of this experiment is shown in Figure 6.2. A visualization of the streamwise velocity in the same snapshot, obtained using Paraview at a clipping plane parallel to the wall and within the range used to bound the position of the vortex centroid can be seen in Figure 6.3. We can see how features are predominant within the low-velocity streaks, colored blue in the Paraview visualization. Note that a low-velocity region is not necessarily correlated with the presence of strong vortices, as observed in the bottom right part of the flow: from that low-velocity region no vortex structure was extracted. This indicates that the vortex structures with the above filtering properties (i.e. convecting with low velocity) are not dominant in the present low-speed streaky region.

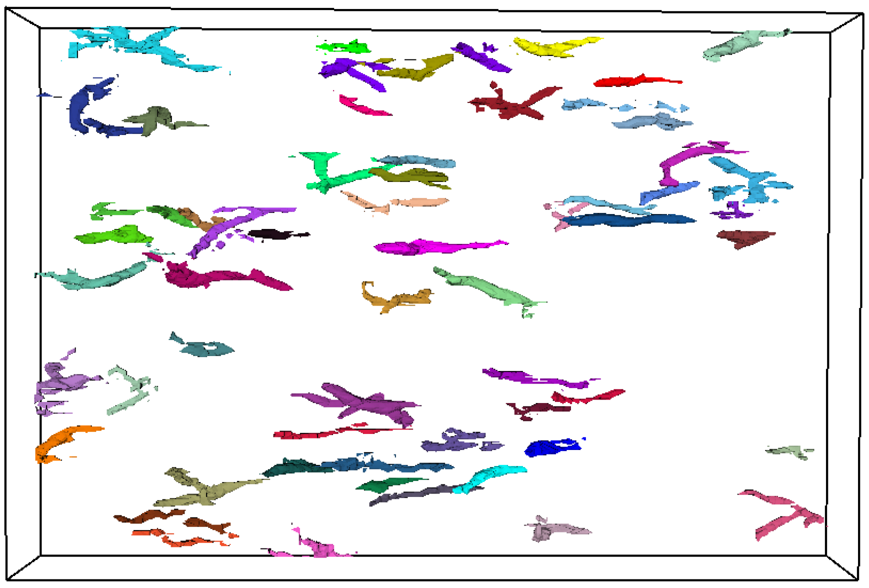


FIGURE 6.2: Visualization of low-velocity streaks features belonging to a time snapshot of a fully developed plane Couette flow

We continued by enabling the shape filtering steps, that enforce the selected features to be ideal streamwise vortices (steps 3-6). The resulting visualization is included in Figure 6.4.

We see that a number of 10 streamwise vortex structures belonging to low-velocity streaks are detected in the used snapshot from the plane Couette flow. Therefore, the number of vortex structures that also satisfy the shape condition is significantly decreased. This indicates that most of the vortex structures appearing in the low-speed streaks do not have an ideal streamwise vortex shape, and appear distorted in the field. By refining the thresholds used, the feature filtering rigidity could be modified.

The motion of the selected features can now be tracked temporally. This is included in Figures 6.5. We can see how the vortex structures have a motion tendency in the streamwise direction. Also, in steps 2 and 4 we can see how two pairs of streamwise vortices join (the magenta and the light blue vortices, as well as the pink and green ones). Their streamwise shape is however still visible in the later time steps.

Apart from tracking the features visually, in future the tool could be used to support the statistical analysis of flows, by taking averages of individual vortex properties such as velocity fluctuations and Reynolds stress. Compared to the existing studies, we would be able to quantify the contribution of each individual vortex structure to the global properties, as well as to filter the vortices with specific properties.

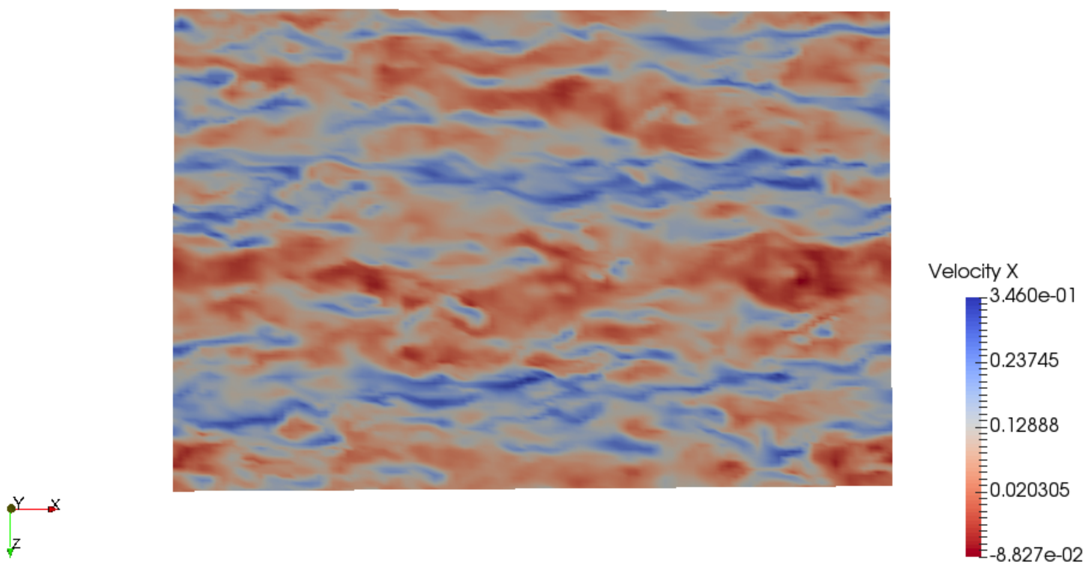


FIGURE 6.3: Paraview visualization of low-velocity streaks in a time snapshot of a fully developed plane Couette flow

6.4 Summary

The important results of this chapter are the following:

- We have demonstrated how the developed tool can be used to obtain better insights into flows. We use the ability of tracking individual features to solve real problems in the field of computational fluid dynamics.
- We have exemplified how the tool could be extended with feature filtering capabilities and, for the specific example of streamwise vortices in low-velocity streaks, motivated its relevance to the fluid dynamics research.
- We proved that ideal streamwise vortex structures exist in the plane Couette flow.
- We demonstrate the usefulness of including more than one scalar in the analysis of flow fields. While so far we only considered one scalar of the field (usually the Q-criteria), we now combine the Q-criteria with the velocity information.

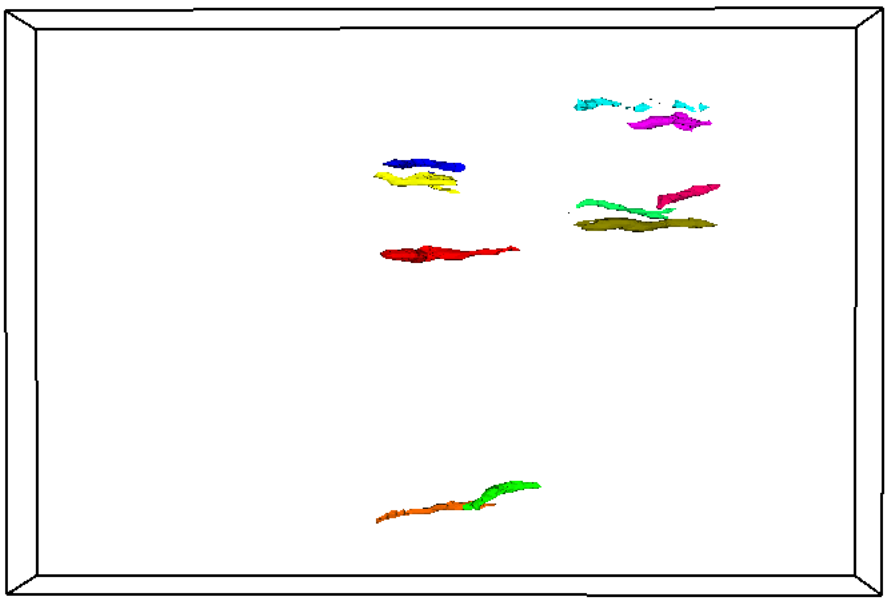
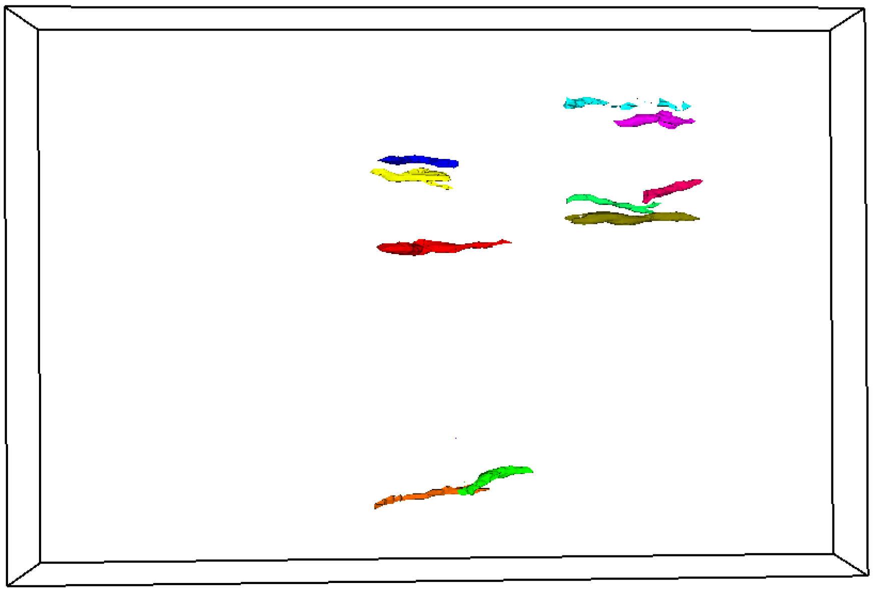
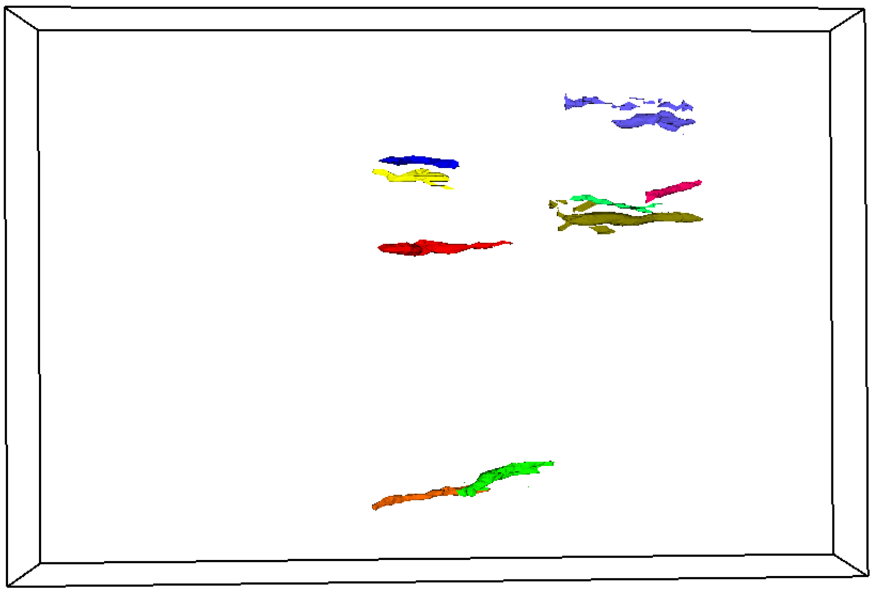
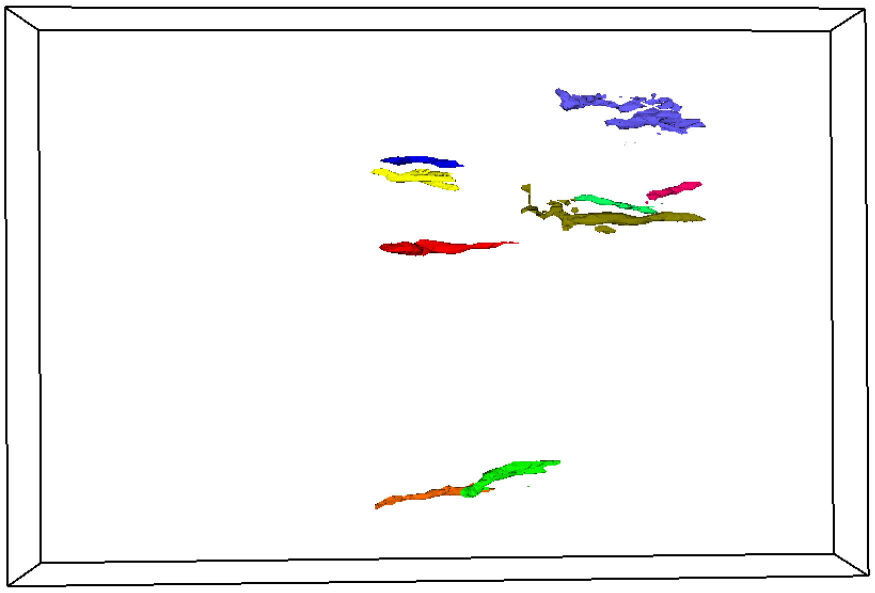
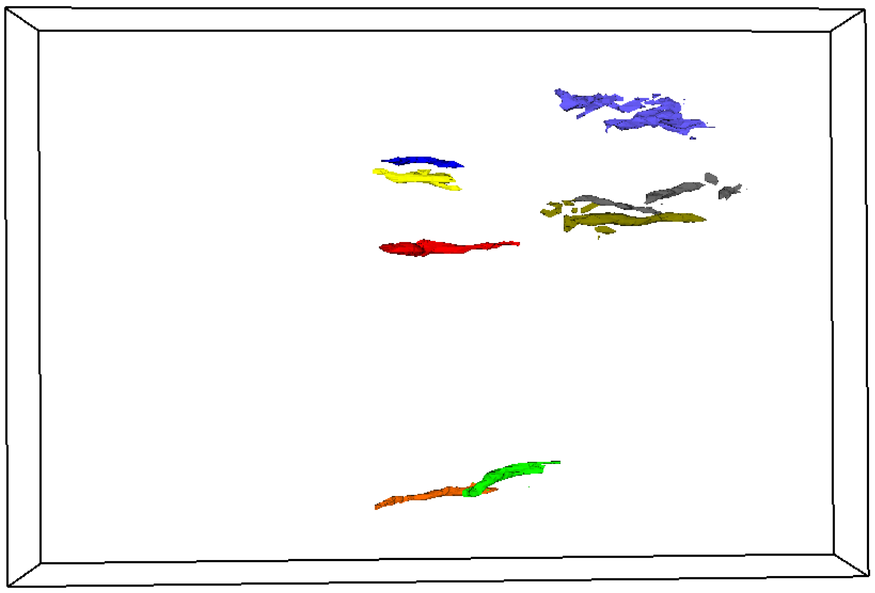


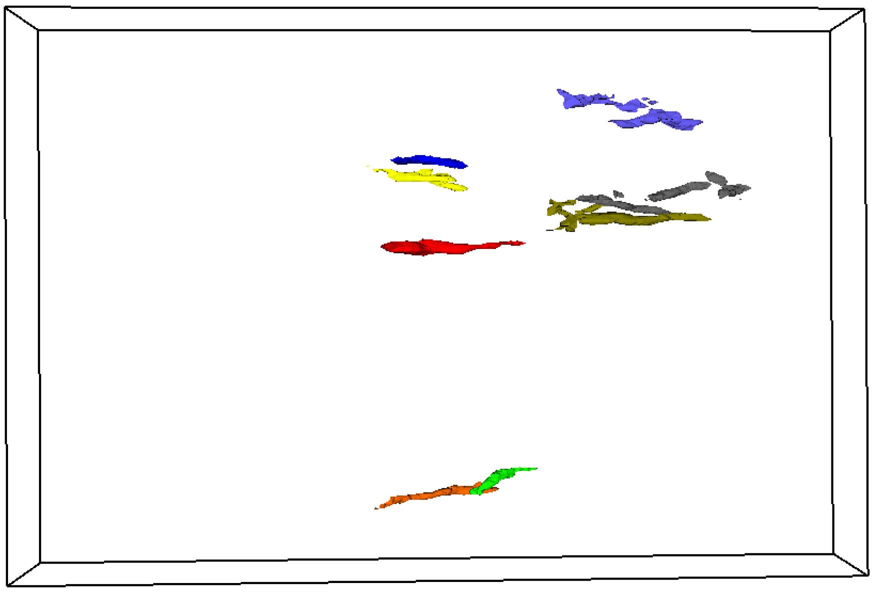
FIGURE 6.4: Visualization of low-velocity streaks features belonging to a time snapshot of a fully developed plane Couette flow


(A) Step 1

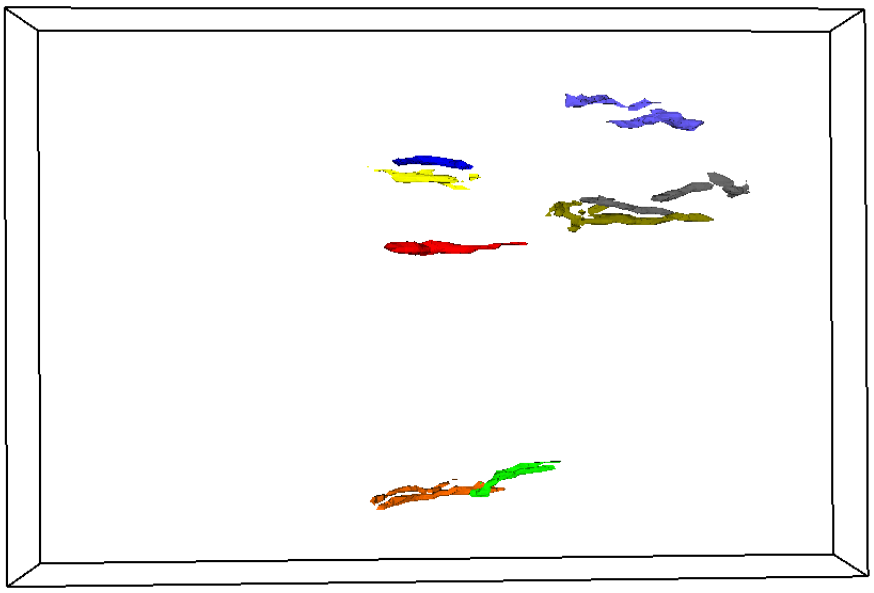

(B) Step 2


(C) Step 3

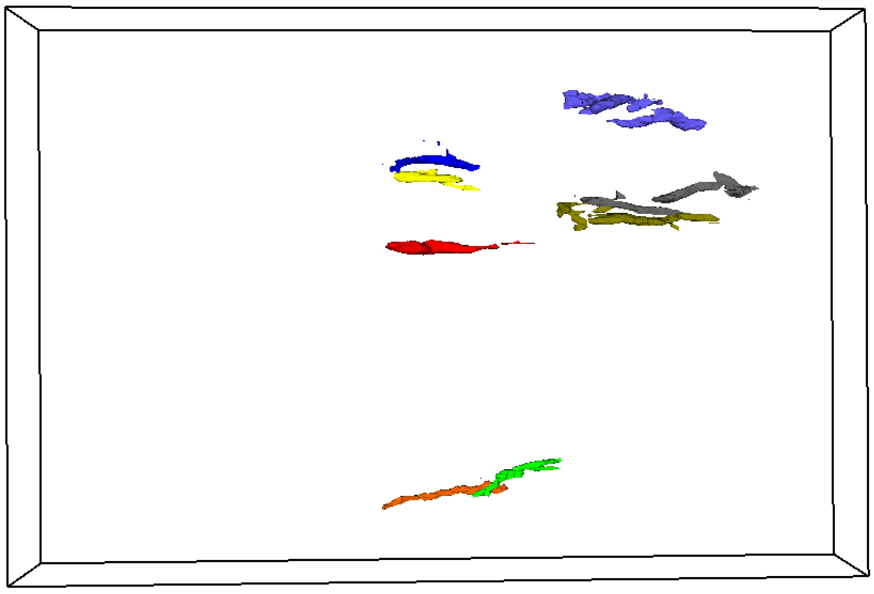

(D) Step 4



(E) Step 5



(F) Step 6



(G) Step 7

FIGURE 6.5: Temporal tracking of streamwise vortices belonging to low-velocity streaks.

Chapter 7

Conclusions

In this project, we proposed a new algorithm for tracking features in scalar field, described in Chapter 3. Chapter 4 describes the tool we implemented in order to enable the temporal visualization of features in time-dependent fields. The efficiency of the algorithm and of the wrapping tool was then evaluated in Chapter 5. In Chapter 6, we demonstrate how the implemented tool can be used to facilitate the understanding of the streak phenomena in turbulent wall-bounded flows. Finally, in Section 7.3, we present a plan describing how the feature tracking algorithm could be integrated in an in-situ pipeline.

7.1 Reflection

The project combined knowledge from a wide range of fields, including graph algorithms, data visualization, geometry or statistics. In some of them, much research effort has been put in order to understand the current state of the literature. A thorough understanding of the existing algorithms for feature tracking was required in order to objectively classify their advantages or drawbacks. The proposed tool was evaluated in the context of computational fluid dynamics simulations, so the project involved the understanding of many specific concepts and problems existing in the field; from this point of view, the chapter concerned with the analysis of streaks in the plane Couette flow was particularly challenging to realize. In addition, the performance of the tool developed was also an important aspect in the perspective of integrating it in-situ, but also in the context of the large-sized test data we used for evaluation.

7.2 Deliverables

The main achievement of this project is the proposal and implementation of an algorithm for detecting correspondences between features extracted using contour trees, in time-varying fields. We demonstrated its efficiency in tracking vortex structures in fields belonging to fluid dynamics simulations, such as the Taylor Green Vortex or the turbulent plane Couette flow.

The algorithm was included in a novel visualization tool, which supports the understanding of time-varying fields with the support of their corresponding contour trees. We demonstrate the usability of the tool in the context of real fluid dynamics problems, using a case study about how it supports the understanding of the streak phenomena maintenance and evolution in the turbulent plane Couette flow. In addition, we present a discussion about how the algorithm could be integrated in an in-situ environment, particularly relevant as a result of the impossibility to store the huge amounts of data produced by the CFD solvers.

7.3 Future Work

7.3.1 In-situ feature tracking

The main purpose of the project was to implement a feature tracking algorithm, having in mind its eventual applicability for tracking features in-situ. In this section we analyse the feasibility of the developed algorithm for this task, from the perspective of potentially integrating it into PyFR, and anticipate some of the challenges associated to performing the tracking at the time of the simulation.

Scalability and hardware

PyFR solves the equations describing the dynamics of fluids on unstructured meshes, supporting a variety of hardware platforms. The sizes of the meshes used to describe the space of the simulations are sometimes of the order of hundreds of millions of cells, so scalability becomes an essential aspect for the value of the software. The scalability of PyFR is analysed in [42], where the benchmarks are using a large scale simulation of flow over a low-pressure turbine blade cascade. The simulations were hosted by the Piz Daint and Titan supercomputers available in the Swiss National Supercomputing Centre and the Oak Ridge National Laboratory, respectively. The largest benchmark included in the paper used a mesh containing 180331250 elements, and ran on 15000 GPUs.

The scalability of PyFR is due to its parallelism capabilities. A simulation can be performed in parallel, by dividing the mesh into connected subregions of cells, and assigning them to different computation nodes. For a mesh consisting of N_e elements and for N_p processing units available, the mesh is going to be divided into regions of size N_e/N_p. The neighboring nodes then have to communicate between them in order to manage flow regions close to the common boundaries.

The complex mathematical systems defining flows are heavily based on matrix multiplications and other algorithms that involve many Single Instruction Multiple Data (SIMD) operations. This fact makes it an ideal candidate to be run on GPUs and, in many simulations (including the one described in [42]), the CUDA backend of PyFR is used to exclusively target the GPUs of the system. Therefore, in the case of heterogeneous architectures

(architectures that pair CPUs with GPUs), the CPUs are going to remain idle for the majority of the simulation time [42].

Tracking features in distributed memory

Naturally, the sequence of snapshots produced by PyFR as part of a simulation has a very high temporal sampling. Running the tracking algorithm between every pair of consecutive fields would result in a very high tracking accuracy. On the other hand, this may not be feasible, depending on the computational cost of the feature tracking. In addition, this may not be even needed, since the feature tracking algorithm can still have a very high accuracy even if a fixed number of time steps are skipped between any two snapshots we compute the correspondence information for. Therefore, there is a fundamental trade-off between the accuracy of the tracking (reflected into how often we compute the correspondences), and the computational cost incurred by the runs of the algorithm.

One difficulty in visualizing the feature tracking in-situ comes from the computational nodes performing their simulation parts locally, using distributed memory. Therefore, a global copy of the state of the flow does not exist. One facile way to overcome this would be to use a central machine that has this knowledge: if this machine was available, we would then be able to obtain the desired visualization and feature tracking by just running the tool we proposed as part of this project. However, this approach comes with a number of problems:

- The I/O problem of transferring the entire scalar field often enough to the central machine is unfeasible. As an example, during the simulations described in [42], global copies of the field were only written 9 times to disk, and each write took around 7 minutes. Although, depending on the performance of the communication channel, sending the entire mesh to the central machine is likely to take less, this is still unfeasible, as even a few seconds of copying would be too much for ensuring a good tracking accuracy.
- The run time of the feature tracking algorithm will be of the order of tens of minutes for meshes with hundreds of millions of cells.
- The approach lacks scalability. While the size of CFD simulations is likely to break the barrier of petabyte-sized data, this way of obtaining the visualization remains a purely theoretical idea.

We clearly need to exploit the simulation's computation parallelism in order to design a scalable way of tracking and visualizing features in-situ. In addition, we mentioned before that, in heterogeneous architectures, the CPUs are mostly idle while the GPUs are carrying out the simulation. A valuable idea is to use this available CPU power to perform the data visualization since:

- The memory used by the CPU and its pairing GPU is shared between the two units. Therefore, the CPU can read the local simulation without any data copying overhead.

- The feature tracking algorithm is better suited for CPUs rather than GPUs.
- The interference between the visualization and the simulation itself would be minimal.

Since a computational node (a CPU) will be responsible for the same mesh portion during the entire run of the simulation, the problem of tracking features inside this region can be solved using the implementation we developed as part of this project. However, the additional difficulty is represented by the preservation of tags for features that cross boundaries between mesh regions owned by different machines. In order to handle this case, we propose the following alternatives:

- Each process would compute and prune its local contour trees, and would conduct the calculations of correspondences between subsequent trees locally. Processes sharing boundaries should then communicate to share their information about the features on the boundaries of their mesh subregions. For example, in order to propagate colored feature tags for a real time visualization, when a tagged feature hits a boundary we have two cases:

    1. The feature continuation was pruned from the contour tree of the neighbouring computation machine.
    2. There is a contour tree region which corresponds to the continuation of the current feature. In this case, we want to assign the same tag to the arcs in that contour tree region.

  Note that the pruning criteria should be adjusted such that features of interest are not pruned from both contour trees while moving across boundaries. Otherwise, their trace can be lost. Also, it is useful to note that, due to the contour tree based nature of the feature tracking algorithm, the features that join or split do not represent special cases. Another aspect is that the fields will often contain features that span the subregions corresponding to more than two processes; therefore sharing the knowledge about tags in a second communication step could be necessary. The total size of the messages between neighbouring machines is proportional to the number of features in the field, so to the total size of the locally computed contour trees. Compared to the dimension of the field, this is much smaller assuming that the trees have been sensibly simplified.

- In the literature, there are several proposals of methods to compute contour trees in parallel, an example being the divide and conquer method proposed in [31] by Pascucci et al.. They make use of a procedure for merging two upper object trees, included in Figure 7.1, which runs in time proportional to the size of the boundary between the regions defining the two trees. For each pair of neighbouring regions, we could assign one of the processes to compute a combined contour tree after the join and split trees are locally computed by each machine. The issue here is that the size of the (unpruned) upper object trees communicated over a boundary can be close to the size of the subregion defining them. Whether this is acceptable or not depends on the in-situ configuration. If it is not acceptable, the merging procedure in Figure 7.1 should be adapted to support the prior pruning of the upper object trees.

```
MergeJT($JT_1, JT_2$)

1   $JT$= NewTree()
2   UF = NewUF()
3   $k \leftarrow$ MergeNodesSorted($JT_1, JT2$)
4   for each node $i = 0$ to $k - 1$ do:
5      AddNode($JT, i$)
6      if IsMin($\mathcal{F}|_{\mathcal{M}_1 \cap \mathcal{M}_2}, i$) then NewSet($UF, i$)
7      for each edge $v_i v_j$ with $j < i$ do:
8         $i' \leftarrow$ Find($UF, i$)
9         $j' \leftarrow$ Find($UF, j$)
10        if $j' \neq i'$ then AddArc($JT, i', j'$)
11        Union($UF, i', j'$)
12  return $JT$
```

FIGURE 7.1: An algorithm for merging two join trees [31]

Computing a shared contour tree for each pair of adjacent processes would inherently solve the discussion about features hitting boundaries presented at the previous point. However, we can see that it comes at a higher computational cost. Also, handling features that span more than two subregions brings additional complexity, as multiple contour trees computed for pairs of adjacent machines have to be considered in order to temporally track these features.

These approaches involve the computation of feature tracking correspondences between fields of the size of one or two mesh subregions, since every mesh subregion is assigned to a different process. Considering the benchmarks described in [42], we can see that the size of the subregions assigned to the GPUs is around 12000. On fields of these sizes, the runtime of the proposed feature tracking algorithm is usually below 0.2 seconds. Combining it with the cost of computing contour trees, the communication delay between processes, and other data processing overhead, we expect the total to be below 1 second. Since PyFR does not generate new snapshots more often than once every 0.1 seconds, in the currently assumed environment we are able to compute correspondences once every 10 snapshots. Due to PyFr's high temporal sampling, the feature tracking is usually accurate enough even when run once every 100 time snapshots, aspect which we tested for the turbulent plane Couette flow. Therefore, we consider that, by using the fundamental simulation parallelism of PyFr, the feature tracking algorithm developed as part of this project is feasible to be run in-situ, as long as it is improved with the techniques for handling boundaries described above.

Making the tracking information valuable

So far, we have analysed the feasibility of tracking features in-situ. Although possible, simply storing the tracking information on disk is not valuable in the absence of the flow fields. These cannot be easily saved due to them being generated orders of magnitude faster than the time taken to be written on disk. We are going to mention a number of other opportunities generated by the ability to track features at the time of the simulation:

1. Visualize how selected features evolve in real time. Paraview Catalyst [3] represents a library for in-situ data analysis, which allows visualization tasks on distributed memory. The library is based on VTK, so its integration with our code may be easier than expected. This way, we could obtain a tool similar to the one we developed as part of this project, but which runs in real time.

2. Obtain spatial and temporal statistics for specific classes of features without storing the simulation on disk. As an example related to Chapter 6, statistics can be obtained for streamwise vortex structures belonging to low-velocity streaks, if they are automatically detected in-situ. Note that the code we developed for detecting them is not immediately applicable to distributed meshes.

3. Mediate the I/O bottleneck of storing CFD simulations. During an in-situ simulation, we could track the evolution of features, highlighting what regions of the field are indeed useful for understanding the flow. This information could be then used during a second run of the simulation in order to only write to disk the interesting regions of the fields.

7.3.2 Dealing with data periodicity

In computational fluid dynamics, the unbounded flows are usually reduced to a (bounded) simulation space, which is then assumed to be periodic with respect to the infinite dimensions. This aspect is further exemplified in Appendix A.

At the moment, we ignore the periodicity of the fields. Therefore, in the case of a feature crossing boundaries of a periodic simulation space, we will wrongly treat it as more than one, independent, features. This is also going to be reflected in the structure of the contour tree, which will have multiple arcs corresponding to one real feature.

To address this, the brief idea we propose is to update the contour tree of the field after it is constructed by ignoring periodicity. We can look at pairs of upper leaf arcs which are attached to the same simulation space boundary (with regards to periodicity). For a match found (according to the significance of the boundary overlap), we are going to remove one of the arcs, and add its associated voxels in the list of the remaining arc. The procedure should be applied repeatedly, to allow to removal of arcs that initially were not leaves in the contour tree. This approach is appealing in the context of feature tracking, as the feature tracking algorithm we developed would be immediately applicable on the contour trees updated this way.

7.3.3 Paraview plug-in

As mentioned before, with Paraview researchers are only able to visualize global isosurfaces, while temporal tracking of features has to be done by eye.

An idea would be to integrate our work in the widely used Paraview tool, by writing a plug-in for it that enables our additional functionality. There are already many plug-ins for Paraview that add new features to the tool, many of them specific to certain research fields. The main advantage of including our work in a Paraview plug-in would be its integration in more advanced pipelines that include other Paraview filters. For example, colouring the vortex structures we extract by field properties, or displaying clipping planes through our current visualization would represent immediate functionality offered by Paraview.

7.3.4 Designing a domain specific language for feature detection

We have seen how our tool can be extended with automatic feature detection logic. However, for every feature type we want to detect, a new C++ class has to be added to the project.

An idea would be to design a domain specific language that would facilitate the process of describing field features. A researcher could then write specifications in this language, while the tool would interpret this kind of descriptions and perform the feature detection accordingly. This way, the process of describing features will be much more concise, and not limited to the researcher's C++ knowledge.

In the particular example of stremwise vortices, we used the distances from the feature's surface points to their linear regression line, as a test for its shape. However, this is not generally applicable for other shapes. A more applicable test would be to consider the histogram of the distances from surface points to the centroid of the feature. By providing an example histogram of the distances, the similarity between it and the histograms of the field's features can be used as a shape test. This idea for detecting features with a specific shape is described in [32].

Appendix A

Fluid flows used

This appendix aims to introduce the reader to two types of flows investigated in the present report, which were used as testing data during the realization of the project.

Throughout this document, the governing equations of fluids are the compressible Navier-Stokes equations. In both flow simulations described, the Mach number is, however, set to be low ($Ma = 0.1$) in order to realize nearly incompressible conditions. Fluid flows simulations are uniquely determined by *initial conditions* and *boundary conditions*, which drive the evolution of the flows. The initial conditions are used to determine the state of the simulation space when it begins. On the other hand, the boundary conditions are used to model the state of the fluid in the immediate proximity of the boundaries. Another important aspect is that, in practice, certain dimensions of the simulation space can be regarded to be infinite (i.e. homogeneous), in the absence of bounding surfaces. To make this computationally feasible, simulations are considered to be periodic with regards to the infinite dimensions.

A.1 Taylor Green Vortex

The simulation space of the Taylor Green Vortex (TGV) is a cube, with the length of the edges L representing the characteristic length scale used in the expression of the Reynolds number Re (described in Section 2.2). The periodic boundary conditions are adopted in each spatial directions (x, y and z). Figure A.1 contains the isovalue at Q-criteria 0.001 taken in the initial stage of the TGV. Although 12 features are observed, in reality the field contains only 8 features, as a consequence of the periodicity of the cube.

The simulation we used was performed at $Re = 1600$, with the initial flow condition given by:

$$u = V_0 sin(\frac{x}{L})cos(\frac{y}{L})cos(\frac{z}{L})$$

$$v = -V_0 cos(\frac{x}{L})sin(\frac{y}{L})cos(\frac{z}{L})$$

$$w = 0$$

$$p = p_0 + \frac{\rho_0 V_0^2}{16}(cos(\frac{2x}{L}) + cos(\frac{2y}{L}))(cos(\frac{2z}{L}) + 2)$$

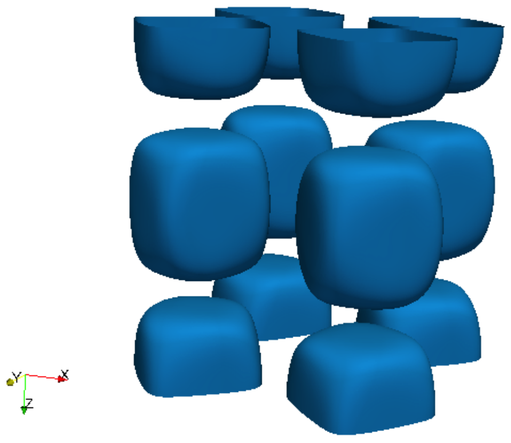


FIGURE A.1: A global isosurface at 0.001 generated using Paraview for the laminar stage of the Taylor Green Vortex simulation

A.2 Turbulent plane Couette flow

The turbulent plane Couette flow represents a canonical example of a wall-bounded flow. The flow simulated is bounded by two parallel walls, that are moving in opposite directions with constant velocity. Here the characteristic dimension used to express Re is given by h, half of the distance between the two moving walls. At the two walls a viscous wall condition is used. In the other two dimensions, the periodic boundary conditions are adopted. The initial condition is given by the mean velocity profile, perturbed by some noise to trigger the turbulent transition. A global isosurface of a snapshot belonging to the plane Couette flow can be seen in Figure A.2.

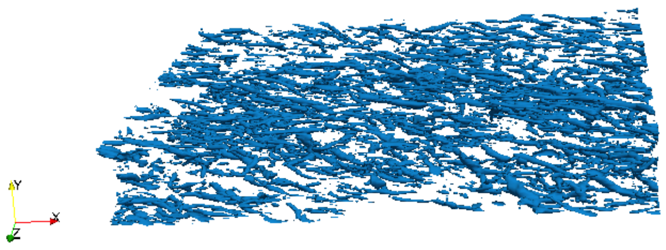


FIGURE A.2: A global isosurface at 0.3 generated using Paraview for a fully developed stage snapshot of the plane Couette flow

Bibliography

[1] R. J. Adrian and Z. C. Liu. "Observation of vortex packets in direct numerical simulation of fully turbulent channel flow". In: *Journal of Visualization* 5.1 (2002), 9–19. DOI: 10.1007/bf03182598.

[2] James Ahrens, Berk Geveci, and Charles Law. "ParaView: An End-User Tool for Large-Data Visualization". In: *Visualization Handbook* (2005), 717–731. DOI: 10.1016/b978-012387582-2/50038-1.

[3] Utkarsh Ayachit et al. "ParaView Catalyst: Enabling In Situ Data Analysis and Visualization". In: *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*. ISAV2015. Austin, TX, USA: ACM, 2015, pp. 25–29. ISBN: 978-1-4503-4003-8. DOI: 10.1145/2828612.2828624. URL: http://doi.acm.org/10.1145/2828612.2828624.

[4] C.l. Bajaj, V. Pascucci, and D.r. Schikore. "The contour spectrum". In: *Proceedings. Visualization '97 (Cat. No. 97CB36155)* (). DOI: 10.1109/visual.1997.663875.

[5] A. Bez Vidal et al. "On the Properties of Discrete Spatial Filters for CFD". In: *J. Comput. Phys.* 326.C (Dec. 2016), pp. 474–498. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2016.09.002. URL: https://doi.org/10.1016/j.jcp.2016.09.002.

[6] Luca Brandt and H. C. De Lange. "Streak interactions and breakdown in boundary layer flows". In: *Physics of Fluids* 20.2 (2008), p. 024107. DOI: 10.1063/1.2838594.

[7] G. J. Brereton and J.-L. Hwang. "The spacing of streaks in unsteady turbulent wall-bounded flow". In: *Physics of Fluids* 6.7 (1994), 2446–2454. DOI: 10.1063/1.868192.

[8] H. Carr, J. Snoeyink, and M. Van De Panne. "Simplifying flexible isosurfaces using local geometric measures". In: *IEEE Visualization 2004* (). DOI: 10.1109/visual.2004.96.

[9] Hamish Carr, Jack Snoeyink, and Ulrike Axen. "Computing contour trees in all dimensions". In: *Computational Geometry* 24.2 (2003), 75–94. DOI: 10.1016/s0925-7721(02)00093-7.

[10] Hamish Carr, Jack Snoeyink, and Michiel van de Panne. "Flexible isosurfaces: Simplifying and displaying scalar topology using the contour tree". In: *Computational Geometry* 43.1 (Jan. 2010), pp. 42–58. DOI: http://dx.doi.org/10.1016/j.comgeo.2006.05.009. URL: //www.sciencedirect.com/science/article/pii/S0925772109000455.

[11] Pinaki Chakraborty, S. Balachandar, and Ronald J. Adrian. “On the relationships between local vortex identification schemes”. In: *Journal of Fluid Mechanics* 535 (2005), 189–214. DOI: 10.1017/s0022112005004726.

[12] J.-I. Choi, K. Yeo, and C. Lee. “Lagrangian statistics in turbulent channel flow”. In: *Physics of Fluids* 16 (Mar. 2004), pp. 779–793. DOI: 10.1063/1.1644576.

[13] G. Haller. “An objective definition of a vortex”. In: *Journal of Fluid Mechanics* 525 (2005), 1–26. DOI: 10.1017/s0022112004002526.

[14] Christian Hill. “NumPy”. In: *Learning Scientific Programming with Python* (), 184–279. DOI: 10.1017/cbo9781139871754.006.

[15] H. T. Huynh. “A Flux Reconstruction Approach to High-Order Schemes Including Discontinuous Galerkin Methods”. In: *18th AIAA Computational Fluid Dynamics Conference* (2007). DOI: 10.2514/6.2007-4079.

[16] Yongyun Hwang. “Large-scale streaks in wall-bounded turbulent flows: amplication, instability, self-sustaining process and control”. Theses. Ecole Polytechnique X, Dec. 2010. URL: https://pastel.archives-ouvertes.fr/pastel-00564901.

[17] Jaiwant Jayakaran et al. “Computational Fluid Dynamics (CFD) Based Combustion Modeling”. In: *Industrial Combustion The John Zink Combustion Handbook* (2001), 287–325. DOI: 10.1201/9781420038699.ch9.

[18] Guangfeng Ji and Han wei Shen. *Feature Tracking Using Earth Mover’s Distance and Global Optimization,” Pacific Graphics 2006.*

[19] Guangfeng Ji, Han-Wei Shen, and R. Wenger. “Volume tracking using higher dimensional isosurfacing”. In: *IEEE Transactions on Ultrasonics, Ferroelectrics and Frequency Control* (). DOI: 10.1109/visual.2003.1250374.

[20] S. J. Kline et al. “The structure of turbulent boundary layers”. In: *Journal of Fluid Mechanics* 30.4 (1967), 741–773. DOI: 10.1017/S0022112067001740.

[21] Marc Van Kreveld et al. “Contour trees and small seed sets for isosurface traversal”. In: *Proceedings of the thirteenth annual symposium on Computational geometry - SCG ’97* (1997). DOI: 10.1145/262839.269238.

[22] Ailsa H. Land and Alison G. Doig. “An Automatic Method for Solving Discrete Programming Problems”. In: *50 Years of Integer Programming 1958-2008* (2009), 105–132. DOI: 10.1007/978-3-540-68279-0_5.

[23] Jin Lee et al. “Spatial organization of large- and very-large-scale motions in a turbulent channel flow”. In: *Journal of Fluid Mechanics* 749 (2014), 818–840. DOI: 10.1017/jfm.2014.249.

[24] *libtourtre. A Contour Tree library.* 2015. URL: https://github.com/sedillard/libtourtre (visited on 01/25/2017).

[25] William E. Lorensen and Harvey E. Cline. “Marching cubes: A high resolution 3D surface construction algorithm”. In: *ACM SIGGRAPH Computer Graphics* 21.4 (1987), 163–169. DOI: 10.1145/37402.37422.

[26] Subha Parvathy Mahaadevan. "Isosurface extraction in the visualization toolkit using the Extrema Skeleton algorithm". PhD thesis. 2003.

[27] John Mandel. "Use of the Singular Value Decomposition in Regression Analysis". In: *The American Statistician* 36.1 (1982), pp. 15–24. DOI: `10.1080/00031305.1982.10482771`. eprint: `http://www.tandfonline.com/doi/pdf/10.1080/00031305.1982.10482771`. URL: `http://www.tandfonline.com/doi/abs/10.1080/00031305.1982.10482771`.

[28] Chris Muelder and Kwan-Liu Ma. "Interactive feature extraction and tracking by utilizing region coherency". In: *2009 IEEE Pacific Visualization Symposium* (2009). DOI: `10.1109/pacificvis.2009.4906833`.

[29] J. Munkres. "Algorithms for the Assignment and Transportation Problems". In: *Journal of the Society of Industrial and Applied Mathematics* 5.1 (1957), pp. 32–38.

[30] Nicholas Nethercote, Robert Walsh, and Jeremy Fitzhardinge. ""Building Workload Characterization Tools with Valgrind"". In: *2006 IEEE International Symposium on Workload Characterization* (2006). DOI: `10.1109/iiswc.2006.302723`.

[31] Valerio Pascucci and Kree Cole-Mclaughlin. "Parallel Computation of the Topology of Level Sets". In: *Algorithmica* 38.1 (2003), 249–268. DOI: `10.1007/s00453-003-1052-3`.

[32] H Saikia and T Weinkauf. "Global Feature Tracking and Similarity Estimation in Time-Dependent Scalar Fields". In: *Computer Graphics Forum (Proc. EuroVis)* 35.3 (2017), accepted.

[33] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1990. ISBN: 0-201-50255-0.

[34] R. Samtaney et al. "Visualizing features and tracking their evolution". In: *Computer* 27.7 (1994), 20–27. DOI: `10.1109/2.299407`.

[35] D. Schneider et al. "Interactive Comparison of Scalar Fields Based on Largest Contours with Applications to Flow Visualization". In: *IEEE Transactions on Visualization and Computer Graphics* 14.6 (2008), 1475–1482. DOI: `10.1109/tvcg.2008.143`.

[36] William J. Schroeder and Kenneth M. Martin. "The Visualization Toolkit". In: *Visualization Handbook* (2005), 593–614. DOI: `10.1016/b978-012387582-2/50032-0`.

[37] D. Silver and Xin Wang. "Tracking and visualizing turbulent 3D features". In: *IEEE Transactions on Visualization and Computer Graphics* 3.2 (1997), 129–141. DOI: `10.1109/2945.597796`.

[38] Daniel Simig. "Turning Flows into Trees: Graph Analytics for Aerodynamic Flows". MA thesis. Imperial College London, 2016.

[39] B.-S. Sohn and Chandrajit Bajaj. "Time-varying contour topology". In: *IEEE Transactions on Visualization and Computer Graphics* 12.1 (2006), 14–25. DOI: `10.1109/tvcg.2006.16`.

[40] *Steve Hollasch's Web Pages*. URL: http://steve.hollasch.net/cgindex/color/colors.txt.

[41] H. Theisel and H.-P. Seidel. "Feature Flow Fields". In: *Proceedings of the Symposium on Data Visualisation 2003*. VISSYM '03. Grenoble, France: Eurographics Association, 2003, pp. 141–148. ISBN: 1-58113-698-6. URL: http://dl.acm.org/citation.cfm?id=769922.769938.

[42] Peter Vincent et al. "Towards Green Aviation with Python at Petascale". In: *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis* (2016). DOI: 10.1109/sc.2016.1.

[43] Hakon Wadell. "Volume, Shape, and Roundness of Quartz Particles". In: *The Journal of Geology* 43.3 (1935), 250–280. DOI: 10.1086/624298.

[44] T Weinkauf et al. "Stable Feature Flow Fields". In: *IEEE Transactions on Visualization and Computer Graphics* 17.6 (2011), 770–780. DOI: 10.1109/tvcg.2010.93.

[45] W. Widanagamaachchi et al. "Interactive exploration of large-scale time-varying data using dynamic tracking graphs". In: *IEEE Symposium on Large Data Analysis and Visualization (LDAV)* (2012). DOI: 10.1109/ldav.2012.6378962.

[46] Jane Wilhelms and Allen Van Gelder. "Octrees for faster isosurface generation". In: *ACM SIGGRAPH Computer Graphics* 24.5 (1990), 57–62. DOI: 10.1145/99308.99321.

[47] F.d. Witherden, A.m. Farrington, and P.e. Vincent. "PyFR: An open source framework for solving advection–diffusion type problems on streaming architectures using the flux reconstruction approach". In: *Computer Physics Communications* 185.11 (2014), 3028–3040. DOI: 10.1016/j.cpc.2014.07.011.

[48] Wang Xudong et al. "Shape optimization of wind turbine blades". In: *Wind Energy* 12.8 (2009), 781–803. DOI: 10.1002/we.335.