Flow-based Segmentation of Seismic Data

Kari Ringdal

June 2012

Master Degree Thesis

Department of Informatics

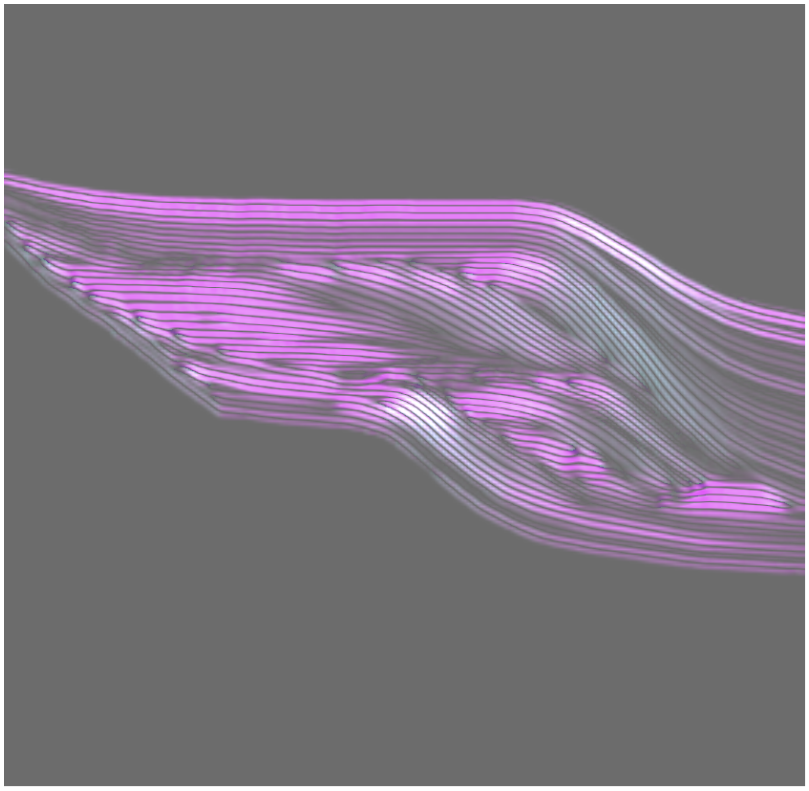
University of Bergen

Master Degree Thesis

Flow-based Segmentation of Seismic Data

by Kari Ringdal

June 2012



Supervised by Daniel Patel

Visualization Group
Department of Informatics
University of Bergen

ABSTRACT

This thesis presents an image processing method for identifying separation layers in seismic 3D reflection volumes. This is done by applying techniques from flow visualization and using GPU acceleration. In geology sound waves are used for exploring the earth beneath the surface. The resulting seismic reflection data gives us a representation of sedimentary rocks. Analyzing sedimentary rocks and their layering can reveal major historical events, such as earth crust movement, climate change or evolutionary change. Sedimentary rocks are also important as a source of natural resources like coal, fossil fuels, drinking water and ores. The first step in analyzing seismic reflection data is to find the borders between sedimentary units that originated at different times. This thesis presents a technique for detecting separating borders in 3D seismic data. Layers belonging to different units can not always be detected by looking in a local neighborhood. Our presented technique avoids the shortcoming of existing methods that work on a local scale by addressing the data globally. We utilize methods from the fields of flow visualization and image processing. We describe a border detection algorithm, as well as a general programming pipeline for preprocessing the data on the graphics card. Our GPU processing supports fast filtering of the data and a real-time update of the viewed volume slice when parameters are adjusted.

Table of Contents

Chapter 1

Introduction

1.1 Problem Statement

When the geologists analyze seismic data, one of the initial tasks is to identify the borders between sediments that originated at different times. This is time consuming, therefor attempts at automation has been done. However, previous automatic tools address the data only on a local scale. Locally run algorithms work fine for identifying borders where the layer pattern of the sediments has a different angularity on either side of the border (Figure 1.1 green circle), but comes short where the layer patterns are parallel (Figure 1.1 red circle). Our goal is to implement an automatic method that address the data globally to also detect borders in parallel neighborhoods. We want to do this by the use of methods from flow visualization, more specifically, by the use of particle seeding. We release massless particles in a flow field representing the sediment direction, trace the particles trajectories, and calculate the separation that occurs. The finite-time Lyapunov exponent is a quantity that characterizes the rate of separation of infinitesimally close trajectories. We use this quantity in our implementation. The first contribution of this thesis is a method for extracting borders in seismic data. Its novelty lies in the fact that it

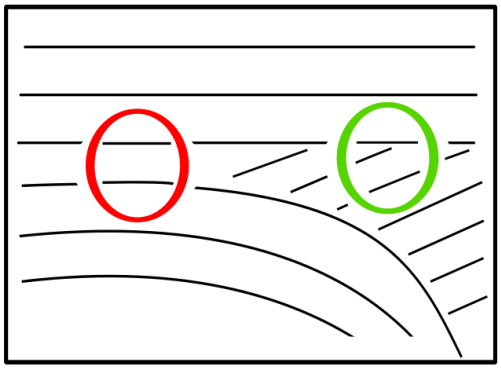


Fig. 1.1: Sediments of different units can be indistinguishable in local areas such as inside the circle.

addresses the data globally, and that it exploits interdisciplinary methods to do so.

The second contribution is a processing architecture for seismic data. The processing steps are divided into distinct modules each having a well defined area of operation. The modules operate on the data in series by GPU-acceleration, which allows the user to monitor every step and adjust parameters at interactive frame rates. This is important for finding optimal parameter settings so that a good segmentation of layers is achieved.

1.2 The Structure of this Thesis

Chapter 1 aims to introduce the reader to the background material which is assumed in subsequent chapters. Some stratigraphic concepts with an emphasis on unconformities are covered. We also take a look at the composition of seismic data, relevant seismic attributes and seismic data represented as a flow field.

Chapter 2 covers previous work relating to this thesis with sections devoted to unconformity detection in seismic data, image processing methods and methods for semantic segmentation of flow. The image processing methods are, more specifically, methods for extracting and smoothing a vector field that we have utilized on images of seismic cross sections. Related work on semantic segmentation of flow addresses two concepts well know within the field of flow visualization - Lagrangian coherent structures and the finite-time Lyapunov exponent.

Chapter 3 describes our general processing architecture. The proposed pipeline for processing seismic data on the GPU and the pipeline for unconformity detection in seismic data is also explained.

Chapter 4 gives implementation details. A short introduction to VolumeShop, which has functioned as the framework for our implementation, and to GPU programming is given before our plugins and shaders are accounted for.

Chapter 5 presents the results of our implementation. The data has been processed on the GPU according to our pipeline and our unconformity detection method is tested. The different parameter settings are discussed and performance measurements are presented.

Chapter 6 concludes and summarizes the findings and limitations of this thesis, and points out some areas for future work.

1.3 About Stratigraphy, Unconformities and Hydrocarbon Traps

Fig. 1.2: Three stratified sediments. The sedimentary sequences are separated by unconformities.

Stratigraphy is the study of rock layers deposited in the earth. A stratum (plural: strata) can be defined as a homogeneous bed of sedimentary rock. Stratigraphy has been a geological discipline ever since the 17th century, and was pioneered by the Danish geologist Nicholas Steno (1638-1686). He reasoned that rock strata were formed when particles in a fluid, such as water, fell to the bottom [Ker03]. The particles would settle evenly in horizontal layers on the lake or the ocean floor. Borders

between different layers are called *horizons*. Through all of Earth's history, layers of sedimentary rock have been formed as wind, water or ice has deposited organic and mineral matter into a body of water. The matter has sunk to the bottom and consolidated into rock by pressure and heat. A break in the continuous deposit results

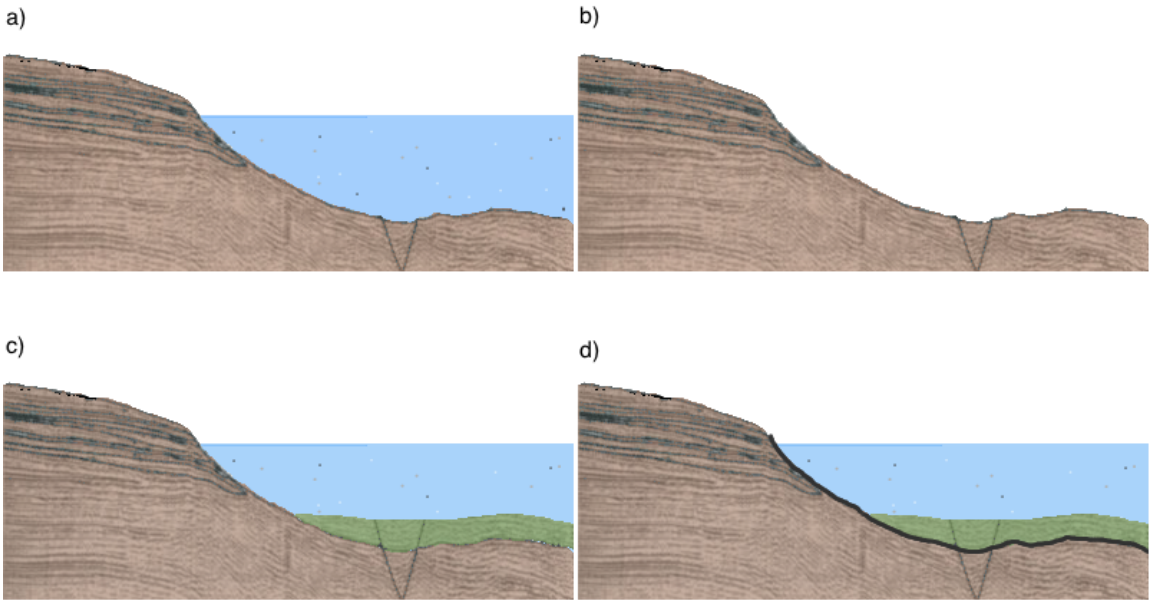


Fig. 1.3: Forming of an unconformity. a) Minerals and organic matter consolidate into rock at the bottom of a body of water. b) A time of lower water levels expose the sediments to erosion. c) New sediment form when the water levels rise again. d) An unconformity is the border between a sequence of young and old sediment layers.

in an *unconformity*, in other words, a surface where successive layers of sediments from different times meet. Thus, an unconformity represents a gap in the geological record. One of the goals of this thesis is to detect such borders in seismic data. Figure 1.2 is a picture of stratified rock where the unconformities are easily seen.

An unconformity usually occurs as a response to change in the water or sea level. Lower water levels expose strata to erosion, and a rise in the water level cause new horizons to form on top of the older truncated layers (see Figure 1.3).

The geologists analyze the pattern around an unconformity to decode the missing time it represents. Above and below an unconformity there are two types of termi-

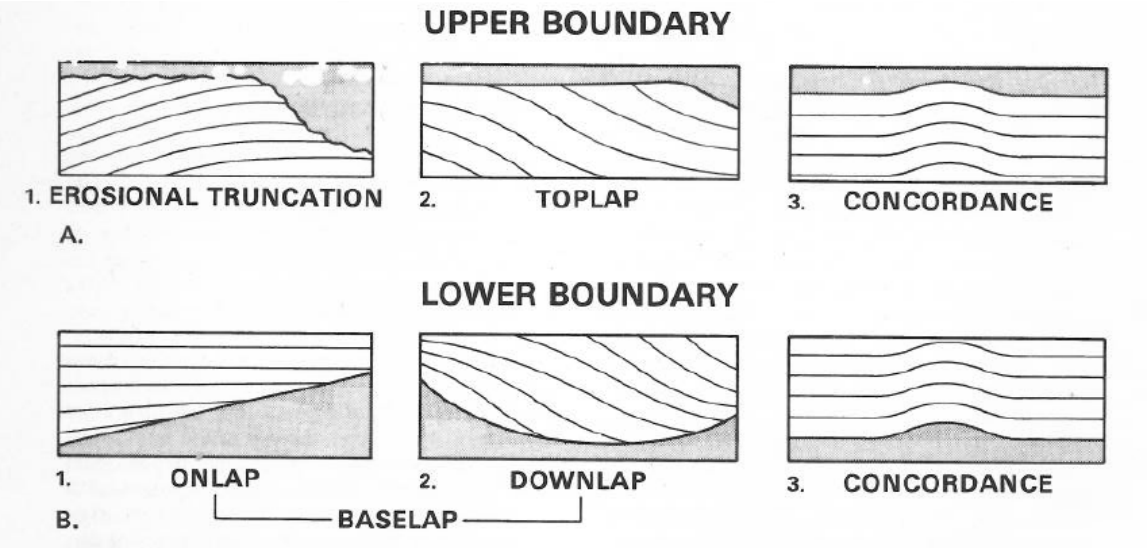


Fig. 1.4: Strata relates to an unconformity in different ways. The top three images show patterns that occur below an unconformity, and the bottom three images show patterns that occur above an unconformity.

nating patterns and one non-terminating pattern. *Truncation* and *toplap* terminates at the unconformity above. Truncation is mainly a result of erosion and toplap is a result of non-deposition. Strata above an unconformity may terminate in the pattern of *onlap* or *downlap*. Onlap happens when the horizontal strata terminates against a base with greater inclination, and downlap is seen where younger non-horizontal layers terminates against the unconformity below. *Concordance* can occur both above and below an unconformity and is where the strata layers are parallel to the unconformity. An illustration of these concepts can be seen in Figure 1.4.

An unconformity can be traced into its *correlative conformity*. In contrast to an unconformity, there is no evidence of erosion or non-deposition along the correlative conformity and the around laying pattern is of type concordance.

A *seismic sequence* - also called a sedimentary unit or facies, is delimited by unconformities and their correlative conformities. The fact that sediments belonging to different seismic sequences can be indistinguishable in greater parts of the picture, as illustrated in Figure 1.1, calls for global analyzing tools.

1.3.1 Structural and Stratigraphic Hydrocarbon Traps

Hydrocarbon exploration companies look for unconformities in the seismic data to identify possible hydrocarbon traps. Hydrocarbons migrate through porous sediments to areas of lower pressure, and they will move upwards until they are above ground if there is a way. Oil and gas reservoirs are therefore dependent on certain subsurface formations that form a geological trap. *Structural traps* are formed as a result of changes in the sediment structure due to folding and faulting. A fold is where the horizons are bent or curved, and it can form under varied conditions of stress, e.g. from hydrostatic pressure. A fault is planar fracture or discontinuity in the sediments. Movements of the earth crust may result in faulting. Figure 1.5 illustrates two structural traps that are created by a fold and a fault respectively. The trapped hydrocarbons are drawn in red, the yellow layer is porous rock, and the green layer is solid and impermeable. Another type of traps are *stratigraphic traps*. They accumulate oil due to changes of rock character rather than folding or faulting. Lateral and vertical variations in the thickness, texture, porosity or lithology of the sedimentary rock may lead to this type of trap. The rock characteristics are not prominent in seismic data the same way the subsurface structures are, and there may be a sequence of impermeable layers functioning as a stratigraphic trap within seemingly uniform areas. Evidently, the stratigraphic traps are harder to detect than the structural traps. Caldwell et al. [CCvB+97] suggest that hydrocarbons trapped in structural traps are almost all in production, and most of the remaining oil and gas are to be found in stratigraphic traps. Our unconformity detection algorithm, that also

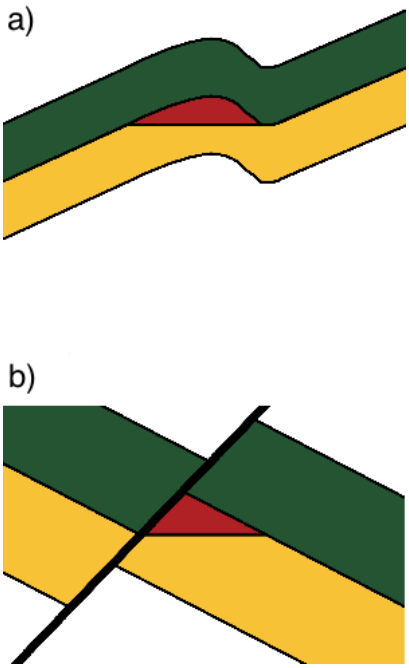


Fig. 1.5: Structural traps a) due to folding and b) due to faulting.

work in parallel areas, can help identify such stratigraphic traps.

For more information on unconformities, sedimentary sequences and other stratigraphic concepts, the reader is referred to Nichols book on sedimentology and stratigraphy [Nic09] and Catuneaus book on sequence stratigraphy [Cat06].

1.4 Seismic Data

Seismic data is generated by processing reflected seismic waves to capture a picture of the rock layers beneath the earth surface. The seismic waves are reflected from interfaces between the rock layers. The recorded return signal and its propagation time is used to determine the depth and angle of the reflecting surface. Many factors influence seismic reflection coefficients, e.g. mineral composition, porosity, fluid content, and environmental pressure.

The data sets may be of two, three or four dimensions with one to four measured values at each data point. 4D seismic data is 3D data acquired at different times over the same area. A seismic data set is typically large and noisy. Wind, swell and direct and scattered surface waves are among the many noise origins that challenge interpretation of the data. A seismic cross section is seen in Figure 1.6. Horizontal sediment layers can be seen on the top as well as more curved layers deeper down.

It is common to represent volumetric data by an equally spaced grid of voxels. A voxel is in a sense a 3D pixel. Where pixels are the building blocks of a 2D image, the voxels are the building blocks of a 3D volume. Each voxel is addressed by a (x, y, z) coordinate.

In this thesis the seismic data will represented as a voxel volume, and it will be processed on the GPU. Seismic data can be large, and a full seismic data set will frequently exceed available GPU memory. A solution is to use stream-processing of the data. It divides the entire data volume into smaller parts that are processed in

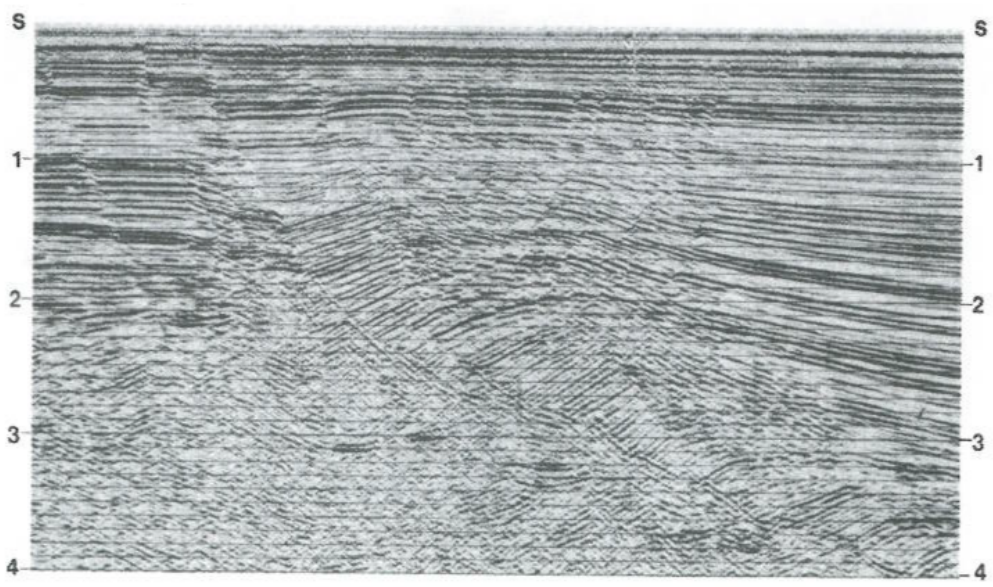


Fig. 1.6: A seismic cross section.

series. We do not support this, but conceptually, it is not hard to extend our method to perform this type of processing. Stream-processing on the GPU, with its limited memory capacity, will still be faster than processing the data on the CPU which has a larger memory space.

1.4.1 Seismic Attributes from Seismic Data

Many techniques to highlight interesting properties of seismic data have been developed. A seismic attribute is a generic term for any information derived from seismic data that aid the interpretation process. Examples include seismic amplitude, envelope, rms amplitude, frequency, bandwidth, coherency and spectral magnitude [Tan01, CM05]. While some attributes are directly sensitive to changes in seismic impedance, others are sensitive to the layer thicknesses or to seismic textures and morphology.

Dip and *azimuth* are attributes that describe the dominating orientation of the strata locally. Dip gives the vertical angle and azimuth the lateral angle. In our work, we use the dip/azimuth vectors to create a flow field representation of the data set where the

flow "moves" along the sediment layers. Through such a representation, we consider the global structure of the sediments when each voxel is analyzed for its probability to be on an unconformity.

1.4.2 Seismic Data Represented as a Flow Field

Vector fields can be used to represent many different data sets. Flow, electricity, magnetism, stress and strain are examples where each point in space can be assigned a direction and magnitude. Such data are well defined by a vector field. As already mentioned, we want to utilize a flow field representation of the seismic data to extract unconformities. As our 3D seismic data represents a snapshot of the subsurface, there is no change over time, so our flow representation can be considered a steady flow. The underlying vector field is defined as a mapping: If U is an open set of the Euclidean space R^n, then a vector field v on U is given by $v : U \rightarrow R^n$. In our case $n = 2$, and we use normalized vectors that are everywhere tangent to the horizons. It is possible to associate such a vector field with a flow as it can be assumed that there is a continuous variation from one point to another.

Our method utilizes the flow field to let particles move along the horizons. In 3D, the horizons constitute surfaces, and along a surface there are several possible particle paths (see Figure 1.7 left). For that reason, when we work with 3D volumes we extract a 2D flow field for each slice of the 3D data. On a 2D slice, the horizons become lines with only two possible path directions. For uniqueness we choose one direction constantly. Basically, we choose the ones going to the right and thereby get a unique 2D vector for each 3D point. We get the tangent vectors by first finding the gradients (they point in the direction of the greatest change and are everywhere orthogonal to the horizons), and then rotating all gradients by 90°. We make sure the flow vectors are aligned by replacing any vector with a negative x component by the opposing vector. Consequently, our method analyses each 2D slice of the 3D data

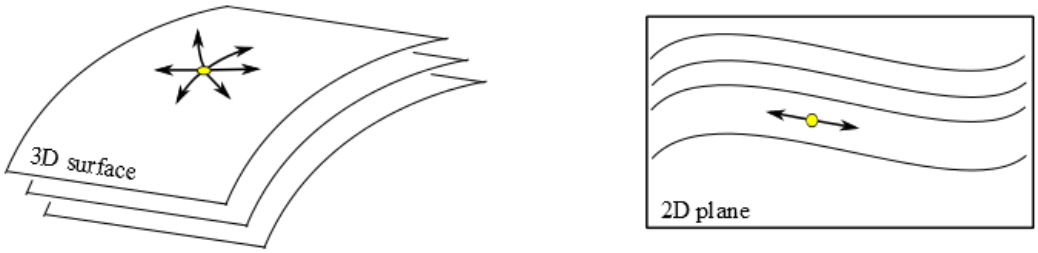


Fig. 1.7: On a surface a particle can move along an infinite number of paths. Along a line, the particle path is well defined.

separately. A true 3D vector analysis would probably detect more features, however it is not clear how the algorithm should be defined. We have further thoughts on this in the section 6.1 concerning possible future work. Anyways, interpretation of seismic data volumes is often done one slice at the time. By processing the volume slice by slice, we use the same data basis as many existing seismic methods.

A trajectory, or *streamline*, of a flow field is the line formed by a massless particle released in the flow. We calculate a streamline by integrating over the flow field (moving along the vector directions in small steps) for a finite number of steps. From any position, a streamline can be found in both the forward and the backward flow. The backward flow is the flow field with all vectors turned in the opposite direction. A streamline starting at a point where the vector field is 0, is a single point. According to seismic data, zero vectors may occur where the reflections are weak, e.g. in uniform areas without detectable horizons. These zero vectors must not be mistaken for singularities in a flow field. A *singularity*, or stationary point in a flow field, defines the behavior of the vectors in its surrounding neighborhood. It can be found, for example, in the center of a vortex. Zero vectors ascribable to an area of weak reflections, on the other hand, is just a missing part of the flow. Therefore we perform smoothing to avoid and minimize such areas.

Our presented algorithm depends on a vector field that represents the intrinsic properties of the seismic data accurately. There exists efficient methods for extracting dip/azimuth vectors from seismic data. They further improve the accuracy by

smoothing the data along its structures [HF02, Mar06]. Our algorithm can take as input a flow field of dip/azimuth vectors extracted by established methods for instance found in Petrel [201]. However, we have reimplemented this calculation to achieve better control and for speed up by adapting it to our GPU pipeline.

Chapter 2

State of the Art

Interpreting seismic data is a time consuming task, and extensive work has been done in developing computer-assisted methods. Chopra and Marfurt give a historical overview and summarize the main events of the trends in seismic attributes [CM05, CM08]. Because of the dense nature of seismic data, traditional visualization of 3D seismic data has been done by cross sections. In the last years, effort has been invested to produce better volumetric visualizations to provide new insight and accelerate the interpretation process [Pat09, Pat10, CM11]. Petrel [201] is one of several commercial softwares used by the oil industry for interpreting and visualizing seismic data.

This chapter will focus on previous approaches on finding unconformities, and also look into methods within the fields of image processing and flow visualization that relate to the new technique presented in this thesis. We review orientation field extraction from image processing since it relates to our vector field extraction of seismic data. The field of flow visualization is discussed as it uses methods relevant for identifying boundary surfaces.

2.1 Unconformity Detection in Seismic Data

One method for detecting sequence boundaries, or unconformities, is the method of Randen at al. [RRSS98]. This method first calculates estimates of dip and azimuth. This is done by applying a multi-dimensional derivative of a Gaussian filter followed by directional smoothing. Then, starting at a sample in the extracted orientation field, a curve is traced out in the direction of the local angle (see Figure 5.14). The

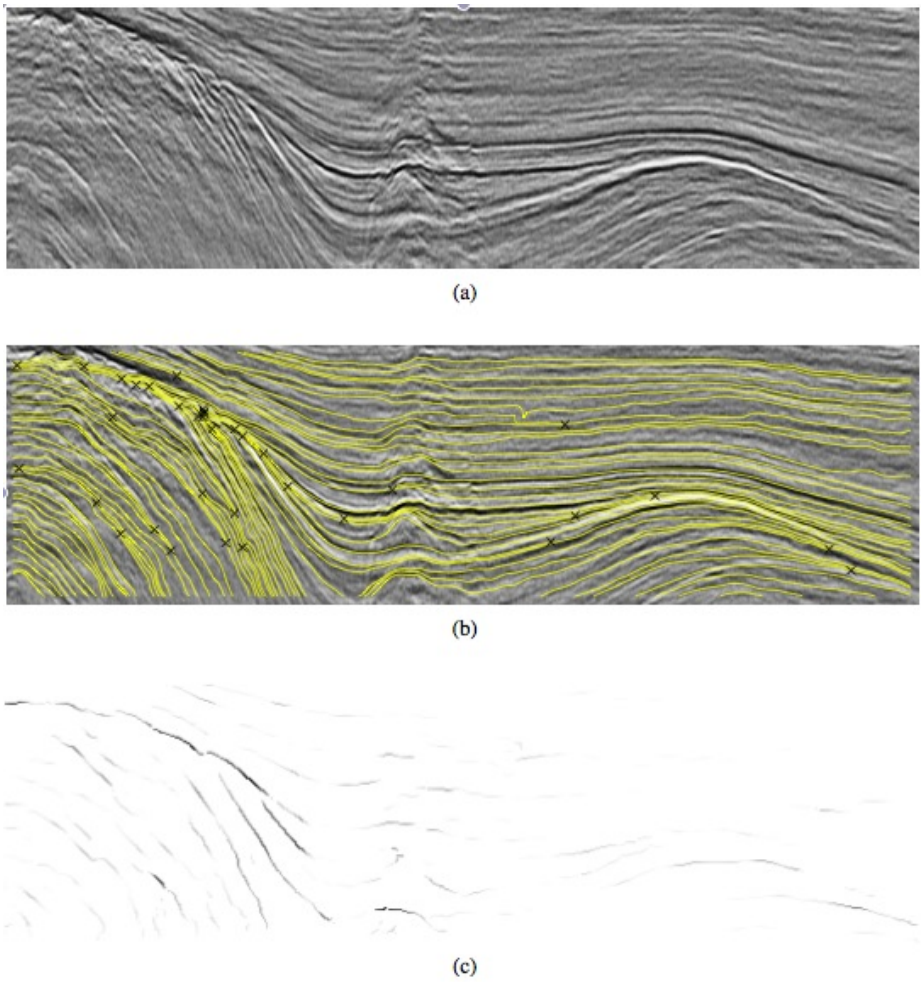


Fig. 2.1: a) Seismic cross section. b) Streamlines and their terminations (marked with X) extracted from the cross section. c) Stratigraphic surface probability from the detected terminations. Darker lines indicate a higher probability.

curves form flow lines along the orientation field. Intersection points of such flow lines are detected (marked with X). These points are likely to be on an unconformity. Brown and Clapp attempted a different approach that locally compares the data to a template that represents a neighbourhood around an unconformity [BC99]. Another method to find lateral discontinuities (e.g. lateral unconformities and faults) in seismic data is that of Bahorich and Farmer called the coherence cube [BF95]. Coherency coefficients are calculated from a 3D reflection volume and displayed as a new volume. Coherence is a measure of the lateral change of the form of the reflected wave along the local orientations in the data. Since the coherence cube first appeared in 1995 it has been improved several times. Chopra gives an overview of the develop-

ment of this method [Cho02]. Unconformities are often seen as discontinuities in the data, but not always. It can happen that there are no obvious signs of erosion and the layers on either side of an unconformity are parallel. This type of unconformity, sometimes called paraconformity or correlative conformity, would not be detected by the coherence cube or any of the above methods.

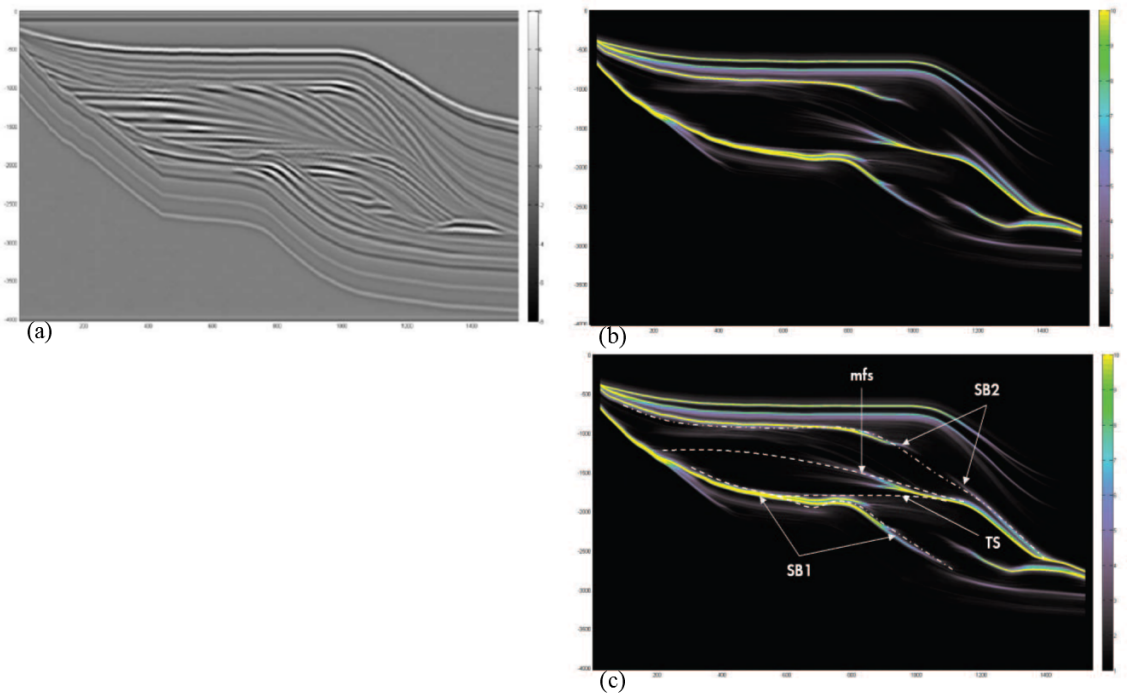


Fig. 2.2: a) Input data. b) The data is processed by an unconformity attribute by Van Hoek et al. [vHGP10]. c) The unconformities are manually enhanced by dotted lines.

A recent paper by van Hoek et al. [vHGP10] describes a new method for finding unconformities. Gaussian derivative filters are used to estimate the dip/azimuth field. The orientation field is then analyzed. The structure tensor field is calculated and regularized, and the principal eigenvector of the structure tensor is extracted. From this, the dip field is studied to see whether the vectors in local neighborhoods diverge, converge or are parallel. Output from their unconformity attribute is shown in Figure 2.2 b). Figure 2.2 a) is the input data, and c) shows the unconformities manually reinforced. Van Hoek et al. recognize the problem that previous methods address seismic data on a local scale, and they attempt to find a more global approach with their unconformity attribute. However, their method measures the conformability

of the dip field in a neighborhood of a predefined size and is therefore still a local method that does not capture events taking place outside its neighborhood.

The unconformity detection methods by van Hoek et al., as well as the method presented in this paper, use the reflector dip and azimuth of the seismic data as input. Much work has been done to extract this accurately. Complex trace analysis [Bar00], discrete vector dip scan [MKF98], and gradient structure tensor [BvVV99] are commonly used for the task. Marfurt presents a refined method for estimating reflector dip and azimuth in 3D seismic data and gives a good overview of the work previously done in this area [Mar06].

2.2 Orientation Detection in Images

In the field of image processing, a repetitive pattern is referred to as a texture, and a linear pattern as an oriented texture. Numerous algorithms are used for enhancing or segmenting textured images. When it comes to processing images digitally for tasks such as edge detection or pattern recognition, there is no algorithm generic enough to be a good choice at all times. It is necessary to choose the right algorithm for the right data and desired achievement.

A finger print, wood grain or a seismic image are examples of oriented textures. These textures have a dominant orientation in every location of the image. They are anisotropic, and each point in the image has a degree of anisotropy that relates to the rate of change in the neighborhood. This is often represented by the magnitude of the orientation vectors. Extraction of the orientation field of a texture has been well researched in the field of image processing. Extraction algorithms are often based on the gradient from a Gaussian filter [RS91, KW85]. In addition to automatic pattern recognition and some edge detection algorithms, directional smoothing of an image requires the orientation field to be calculated. Like in seismic data evaluation, it is

essential that the extracted orientation field represents the properties of the image.

Non photo-realistic rendering (NPR) concerns with simplifying visual cues of an image to communicate certain aspects more effectively. Kang et al. [KLC09] suggests a new NPR method for 2D images that uses a flow-based filtering framework. An anisotropic kernel that describes the flow of salient image features is employed. They use a bilateral filter (an edge-preserving smoothing filter) for the construction of what they call an edge tangent field (ETF). This is a vector field perpendicular to the image gradients. The initial gradient map is obtained by a Sobel operator. The vector adapted bilateral filter takes care to preserve salient edge directions, and to preserve sharp corners in the input image. The filter may be iteratively applied to smooth the ETF without changing the gradient magnitude. Figure 2.3b is an image of an ETF presented by line integral convolution (LIC). The LIC image shows how the vectors of the ETF follows the edges of the input image. Kang et al. present this

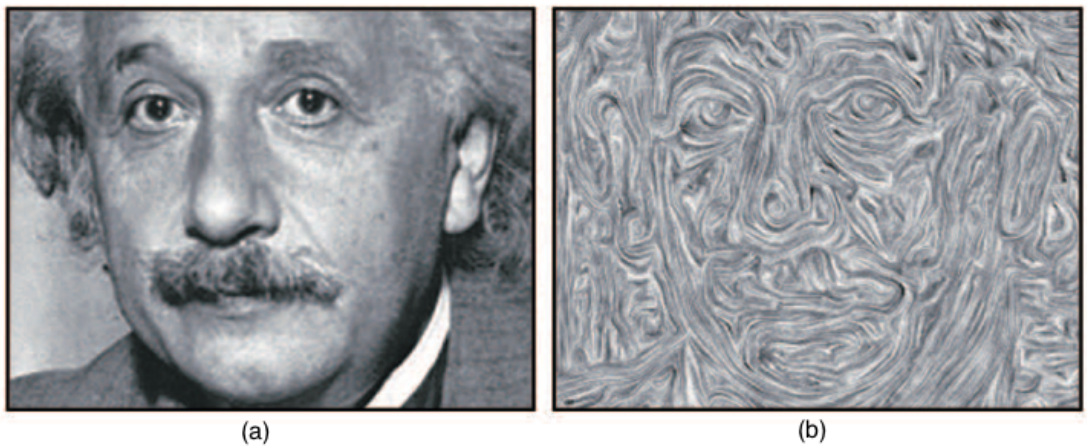


Fig. 2.3: a) Input image. b) The extracted edge tangent field [KLC09].

vector field extraction method as a base for extracting image features. A different vector field extraction method is proposed by Ma and Manjunath [MM00]. They find edge flow vectors by identifying and integrating the direction of change in color, texture and phase discontinuity at each image location. The vector points in the direction of the closest boundary pixel.

2.3 Lagrangian Methods for Semantic Segmentation of Flow

Flow visualization is a subfield of data visualization that develops methods to make flow patterns visible. A *Lagrangian* approach to study fluid flows uses particle trajectories to reveal the nature of a flow. Here, a way to understand a time dependent, or unsteady flow is by dividing it into regions where the flow behaves similarly, and study how these regions deform and move over time. Regions that have a similar flow pattern are called *Lagrangian coherent structures* (LCS). A flow can be seen as a set of LCS.

Methods for finding LCS can also be used on steady flows [Hal01, SP09]. Sadlo and Peikert [SP09] show comparable results from LCS methods and the previous established vector field topology (VFT) methods applied to 3D steady flows. VFT, as introduced to the field of visualization by Helman and Hesselink [HH89], includes methods to identify and visualize the topological structure of a vector field. Mainly by detecting zero gradients (singularities) and the connections between them. In contrast to VFT, the LCS methods are not based on detecting singularities. Haller [Hal01, Hal02] have demonstrated that the *height ridges* of the finite-time Lyapunov exponent field (see Section 2.3.1) work as LCS boundaries. A height ridge can be defined as the major axis of symmetry of an elongated object. In 3D are the LCS boundaries flow separation and attachment surfaces, i.e. surfaces along which the flow abruptly moves away or attaches.

2.3.1 The Finite-time Lyapunov Exponent (FTLE)

The finite-time Lyapunov Exponent (FTLE) characterizes the rate of separation between infinitesimally close trajectories in a flow. It is typically found by calculating a set of trajectories starting from very close positions. The trajectories are used to find

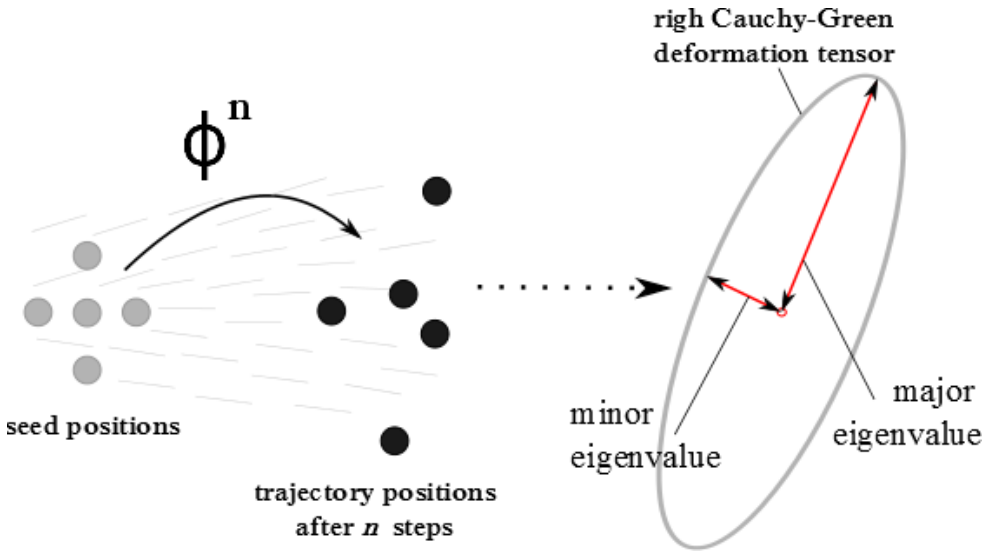


Fig. 2.4: Close trajectories in a flow field, advanced for a set number of steps, are used to calculate the right Cauchy-Green deformation tensor. The FTLE is derived from the major eigenvalue of this tensor.

the flow map gradient. This gradient is left multiplied with its transpose resulting in a tensor called the right Cauchy-Green deformation tensor (see Figure 2.4). At each location, the FTLE is derived from the maximum eigenvalue of the calculated deformation tensor. Since the 1970's the FTLE has been well established as a mean to analyze the chaos and predictability in dynamical systems [PPF+10]. Haller pioneered the use of FTLE on velocity fields of fluid flow, and showed the relation between FTLE and LCS [Hal01, Hal02]. The notion of the height ridges of FTLE as LCS have been further studied by Shadden et al. who proved that the flux across these height ridges is small and most often negligible [SLM]. FTLE has since been applied to a large variation of fluid flow problems within many different fields [PD10]. Also within the visualization community, FTLE and LCS has played an important role. Much work has been done building on these concepts. For an overview, the reader is referred to a recent article by Pobitzer et al. that give the state of the art on visualization of unsteady flow [PPF+10]. As far as we know, this thesis is the first attempt to exploit LCS for detecting unconformities in seismic data.

Finding LCS in 3D flows is computationally intensive, and this has inspired efforts to optimize the computation of FTLE [GGTH07, GLT+09, LM10, SRP11]. Sadlo

and Peikert suggest incorporating adaptive refinement of the scalar mesh into the ridge extraction procedure, and only calculate trajectories in the regions containing ridges [SP07]. Garth et al. show that finding LCS on a cross section of a 3D flow, may often be sufficient for visualizing and understanding the flow [GGTH07]. They also use GPU accelerated path line integration to speed up FTLE computations on 2D flow [GLT+09]. Because our flow from seismic data is not time dependent and we work in the 2D domain, we get sufficient efficiency by GPU accelerated calculations of FTLE without using complex optimization methods.

In a recent paper, Haller and Sapsis show that the smallest eigenvalue of the right Cauchy-Green tensor field relates to the rate of attraction between the trajectories [HS11]. They prove that in a time interval $[t_0, t]$, the maximum eigenvalue can be used to find repelling LCS at t_0 and the minimum eigenvalue can be used to find attracting LCS at time t.

Chapter 3

The Processing Architecture

Complex algorithms running on large datasets take time to complete. In our case, the algorithms require several input parameters. The optimal parameter choices depend on the data itself and are often obtained through trial and error. We speed up the trial-and-error cycle by designing an architecture that gives immediate previews on the results, in real-time, as the user changes the parameters. The basic idea behind our processing architecture is thus to process 3D data efficiently with the possibility for a user to intervene along the pipeline. To do this we have broken the process down into smaller processing segments and constructed a pipeline of modules. This chapter will explain our processing architecture. First we present our pipeline for general processing of volumetric data on the GPU, and then we show how we have specialized it into detecting unconformities in seismic volume data.

3.1 Our Pipeline for Processing Seismic Data on the GPU

The use of GPU-accelerated methods is rapidly increasing in sciences requiring calculations, and can dramatically increase the computing performance on seismic data. The highly parallel structure of modern GPUs is ideal for efficiently processing large blocks of data provided the calculations could be done in parallel. GPUs support programmable *shaders*, that manipulate geometric vertices and fragments and output a color and transparency (RGBA) for every pixel on the screen. Instead of output to

the screen, the RGBA-vectors can be written to a 2D or 3D texture. A texture is in this sense a block of memory located on the GPU.

In our pipeline, we use two 3D textures of equal size and alternate on reading and writing to and from these textures in a ping-pong fashion. The input data is read from one texture, and the output is written to the other. The next procedure moves the data in the opposite direction. If we used only one texture, and did the reading and writing to and from the same memory locations the data would get out of sync and consist of a mixture of processed and unprocessed voxels. The problem would arise when the processing of one voxel requires information from surrounding voxels. Then the sampled information would sometimes be output information instead of input. This would happen although the processing is done in parallel, as reading and writing is not necessarily done at the exact same time for all processed voxels. The original volume is kept as a separate texture on the GPU to provide the input in case of reprocessing (see Figure 3.1).

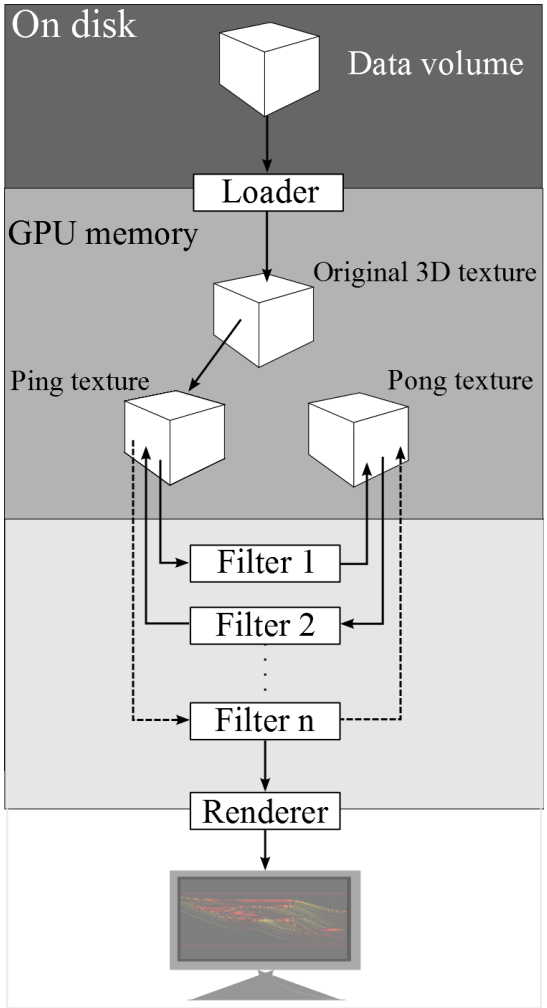


Fig. 3.1: Data is alternately read and written between two 3D textures located on the GPU while filters are applied iteratively. Filtering is done in parallel one slice at the time. Any output slice from any of the filters can be rendered to the display to check the result.

This set up allows us to divide the processing into smaller modules that can be executed one after the other on the GPU. The processing steps can be seen as filters that

perform a small and well defined operation on the data. The filters are implemented as shaders. Each shader processes the data on the GPU in parallel, and the output may be viewed slice-wise. The pipeline can operate in a preview mode where a subset of the volume slices is processed by each filter. This mode is beneficial for adjusting the filter parameters as it gives a real-time update of the displayed slice for smaller subsets. The displayed slice is the *current slice*. There is a global current slice slider that tells all the filters which slice to process. Each filter also has a radius parameter that sets how many slices around the current slice it will process. This parameter is added to allow 3D filtering in preview mode. When 3D filters are used, the radius at every filter must be large enough to provide the needed number of processed input slices for the following 3D filters. After the desired parameter setting are found in preview mode, the operation to process the whole volume may be executed. Then all the volume slices are processed at each filtering step before the next filter is executed.

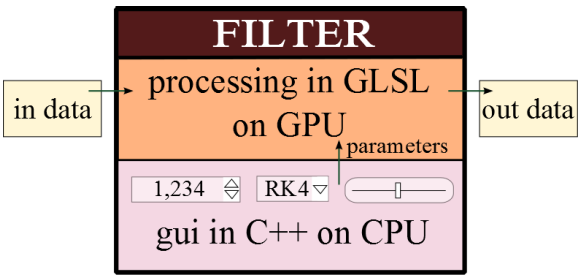


Fig. 3.2: A filter has a shader part and a GUI part. The desired shader and its parameters are set from the GUI.

A filter consists of a GLSL shader that take care of all calculations, and a GUI written in C++ (see Figure 3.2). The shader responsible for the desired filter operations can be chosen from the GUI. The filter parameters are adjusted by GUI sliders that connect to variables of the chosen shader.

Our processing pipeline should meet the needs of a fast seismic processing work flow. In seismic data processing, a series of routines are applied to large seismic data. The calculations of each routine are typically executed for every voxel in the data volume and can be done in parallel. Every routine also depend on one or several parameter settings that needs to be adjusted to best suit the data set in question. As we have seen above, our pipeline comply with these requirements.

3.2 The Pipeline Adapted for Unconformity Detection in Seismic Data

An automatic method for finding borders between sediments in seismic data would greatly reduce manual interpretation times. The goal of this thesis is to create such an automated method. We built the architecture described in the previous section, to address the problem. To identify unconformities in seismic data, we have constructed a pipeline of three distinct GPU-accelerated modules. Output from one module is input to the next. The three modules can be summarized as:

- Vector field extraction from the seismic data
- Mapping of stratigraphic surface probability
- Visualization of surfaces

Figure 3.3 gives an overview of the pipeline. This section will elaborate on each of the modules. This thesis focus on the mapping of stratigraphic surface probability using concepts from flow. In this step lies the novelty of this thesis.

The pipeline is implemented as plugins in the VolumeShop framework [BG05]. A separate *load-plugin* takes care of loading the data onto the GPU and initializing the ping-pong textures. The first two modules of the pipeline are performed by several instances of a *filter-plugin* with different settings for each instance. A volumetric visualization of the extracted surfaces (module 3) is done by plugins already existing in the VolumeShop framework. VolumeShop provides good visualizations of 3D volumes that have been sufficient for examining outcomes of our algorithm. The plugins needed by the rest of the pipline, are our implementations.

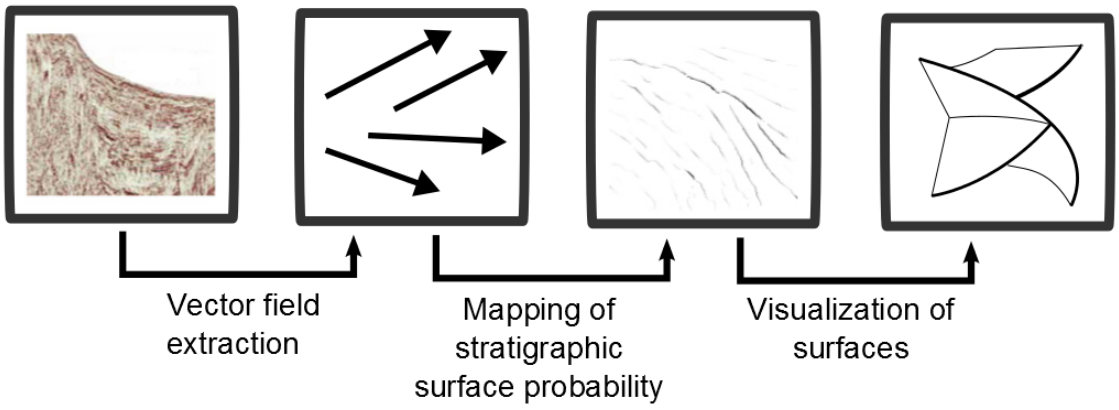


Fig. 3.3: Pipeline for unconformity extraction.

3.2.1 Preprocessing - Vector Field Extraction

There are many seismic attributes that need the orientation vector field of the seismic data as input in the calculation. One important is structure oriented filtering (SOF). SOF is used to improve the quality of seismic data. Because of the importance in seismic attribute calculations, many vector extraction methods already exist [LMADA02, LPG+07, CM07]. We could easily run our data through such a method, using for instance Petrel [201]. However, we have implemented one ourselves to get it into our pipeline giving the user real-time access to all parameters. Another reason we implemented this ourselves is that we want to avoid zero vectors in areas with no structure. Here we instead want a vector describing the direction from a larger perspective since our unconformity detection algorithm depend on a vector field that is defined in every voxel.

SOF for smoothing seismic data has been in use since the nineties, and the used methods are closely related to the image processing methods for edge preserving (anisotropic) smoothing. SOF for smoothing seismic data is done by estimating the local orientation of the reflections and performing smoothing along them. However, smoothing across the reflection terminations must be avoided. Reflection terminations occur mainly at faults, and can be found with edge detection meth-

ods [RPS01, AT04]. Figure 3.4 is from the article of Höcker and Fehmers [HF02] and shows a seismic slice before and after structure oriented smoothing using an anisotropic diffusion filter.

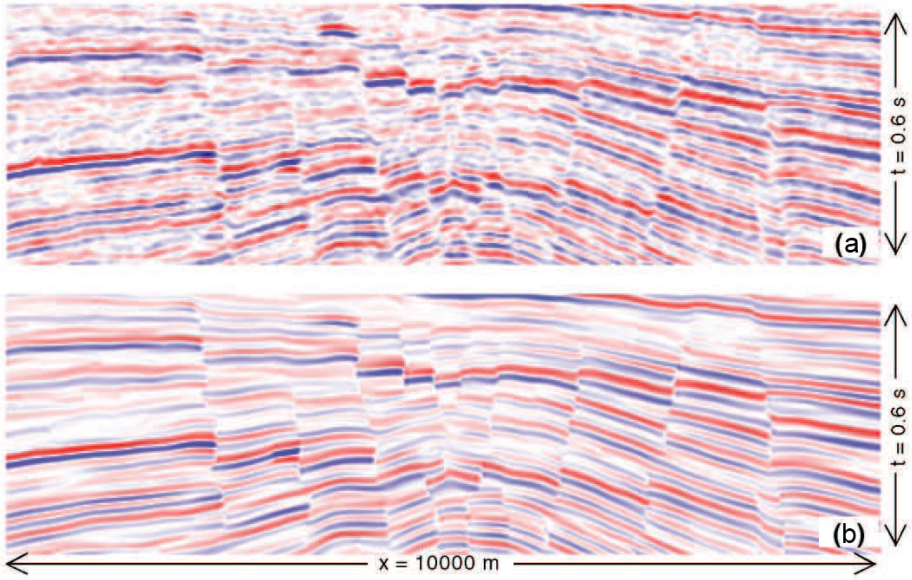


Fig. 3.4: a) Input slice, b) smoothed through SOF by Höcker and Fehmers [HF02].

The first step of our vector field extraction implementation is to find the derivatives by calculating the central difference in each voxel. This gives us the gradient vectors. For a less noise sensitive procedure for gradient extraction, we have also implemented a Sobel filter as an alternative. The extracted gradients are perpendicular to the reflection layers. For reasons explained in Section 1.4.2, we only use the x- and y-components of the gradient vectors. The vectors are rotated around the z-component and normalized to constitute a 2D flow field that is everywhere tangent to the horizons. Smoothing will in most cases improve the flow field representation of the data. Since the filters for calculating the gradients are sensitive to noise, we have implemented a 7×7 Gaussian filter and a mean filter where the mask size can be set by the user. These filters may be used to eliminate some noise before the gradients are found. After finding the vectors, we also utilize the flow-based, SOF method by Kang et al [KLC09] to smooth the vectors constituting the flow field. The vectors representing each sedimentary unit should be aligned and gradually changing accord-

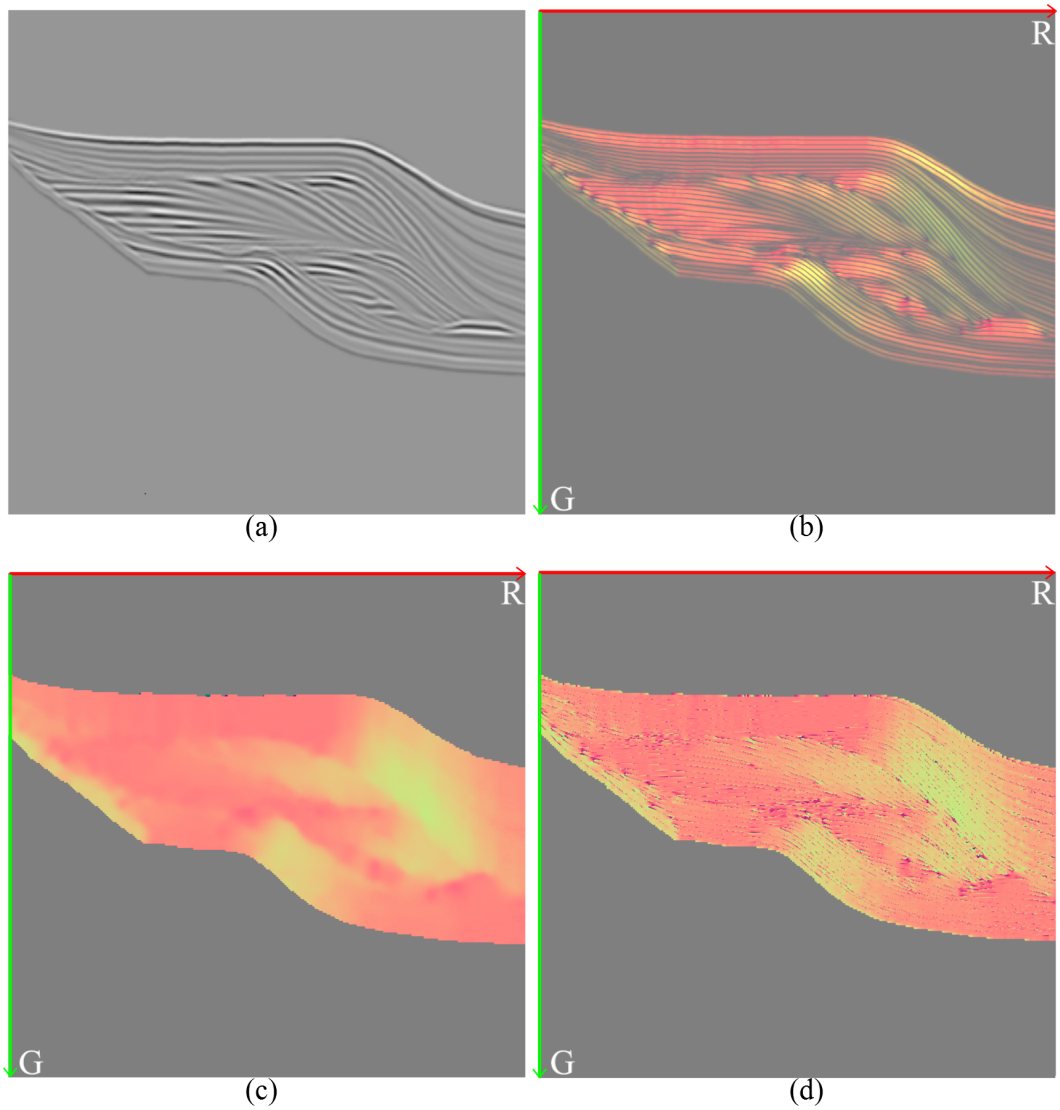


Fig. 3.5: a) Seismic data. b) The flow field found by rotating gradient vectors. c) The flow field after 3 iterations of Kang et al.[KLC09] smoothing filter. The vectors are normalized. d) For comparison, the normalized vectors of b) without any smoothing.

ing the the underlying horizons. However, a gradual change across unit borders is not desirable unless it is intrinsic to the data. The SOF method efficiently smooths the flow vectors without diminishing important information by smoothing across the borders. A flow field, extracted and smoothed by our pipeline, is seen in Figure 3.5. Closer implementation details will be covered in Section 4.3.4.

3.2.2 Processing - Mapping of Unconformity Surface Probability

Our unconformity detection algorithm is implemented as a filter-plugin in our framework. The technique uses particle seeding, as is common in the field of flow visualization. The trajectories of the particles are not visualized themselves, but used to calculate a scalar value describing the degree of separation at the seedpoint. Figure 3.6 is a simplified illustration of the algorithm. For every voxel in our flow field we

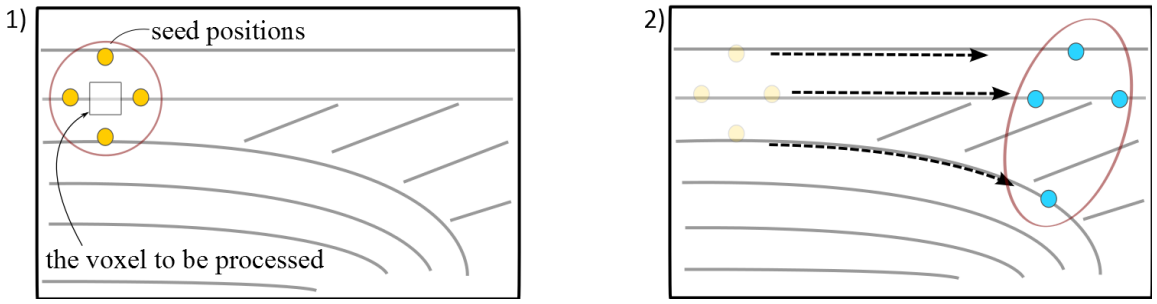


Fig. 3.6: Illustration of the border detection algorithm. Massless particles are released in a flow field. The yellow dots represent the seed points. The rate of separation between the trajectories are calculated form the trajectory end positions (blue dots) and stored in the center voxel in Fig. 1).

seed four particles from its neighboring voxels. Their trajectories are are advanced for a finite number of steps, before the separation between the trajectories are calculated. The rate of separation is mapped to the position of the starting voxel, i.e. the center voxel of the seed points. This is done for every voxel on a slice, and slice by slice throughout the volume. The output is then a 3D volume containing the probability each voxel has for being on an unconformity. Because of the parallel nature of seismic data, two close trajectories will end up in close proximity to one another if their seed points are within the same sedimentary unit (see Figure 3.7). If, instead, their seed points lay on different sides of an unconformity, it is likely that the trajectories will separate when advanced along the flow field. Therefor, with this method, unconformities are detected in areas with parallel horizons where local methods would fail.

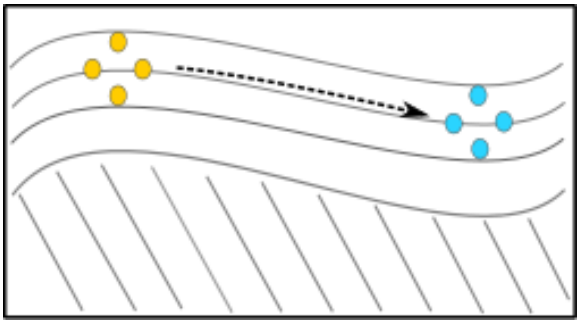


Fig. 3.7: Particles seeded within the same sedimentary unit will end up in close positions.

For more robustness of the algorithm, we trace particles in both directions of the flow. In the forward flow and in the opposed backward flow (see Figure 3.8). The trajectories may separate only in one of the flow directions, and we are interested in mapping the separation no matter if it occurs in the forward or the backward flow. The calculated separation for both flow directions are compared, and the greatest

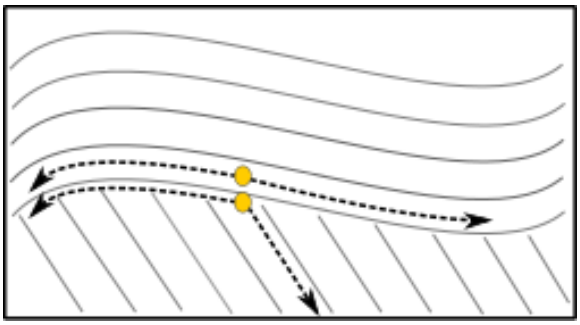


Fig. 3.8: For each voxel a set of particles are traced in both flow directions.

value is saved as this indicates the probability of a separator.

Successful and Failing Cases

Although our method can detect unconformities not detectable by current methods, we are not able to detect all. Therefore, for the rest of this section we will take a closer look at the outcome of the algorithm in a few synthetic scenarios, and determine where it succeeds and where it fails. Figure 3.9, 3.10, 3.11 and 3.12 illustrates stylized seismic cross sections where the black lines represent the pattern of the seismic data. The colored lines indicate the results from applying our algorithm. A green line

indicates an unconformity rightfully detected by our method, an orange dotted line indicates a false negative (i.e. an unconformity not detected by our method), and a red line indicates a false positive (i.e. an unconformity detected by our method which is not actually an unconformity).

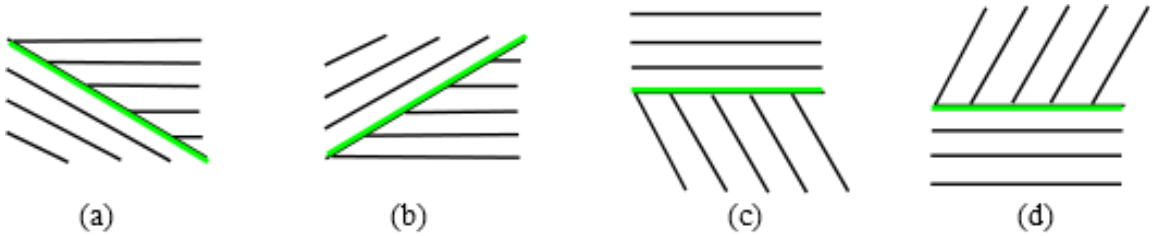


Fig. 3.9: Our algorithm will have a successful outcome in the situations of onlap a) and b), toplap c) and downlap d).

Figure 3.9 illustrates the seismic patterns of onlap (a) and (b), toplap (c) and downlap (d). Our algorithm will detect a separation in one of the flow directions between the trajectories starting on, and close to, the unconformity. In other words, we get a successful outcome in these situations. Local methods would successfully find the unconformities in all cases in Figure 3.9.

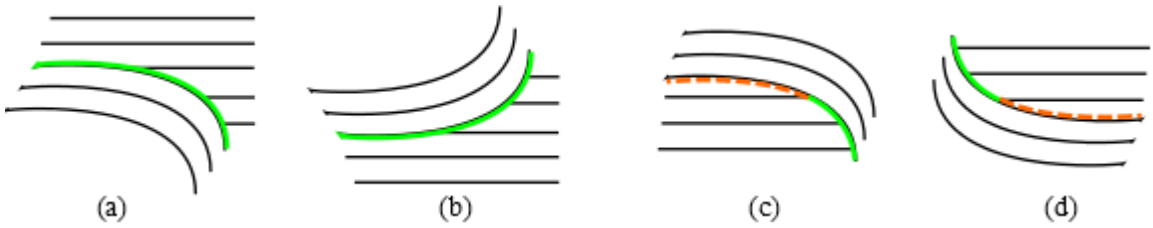


Fig. 3.10: The algorithm will have problems detecting an unconformity in parallel areas where where the horizons go from parallel to converging.

When the horizons show a parallel pattern in some areas, as illustrated in Figure 3.10, a successful outcome is expected in the situation shown in a) and b). A particle set starting from voxels around the unconformity will separate as they move along the horizons. In c) and d), on the other hand, we will have attracting trajectories as they move from the unconformity in the parallel area into the non-parallel area. That is, when the particle set reach the non-parallel area, some of the particles will continue along the same trajectory. If we only look for the separation, the unconformity will

not be detected in the parallel area of Figure 3.10 c) and d).

Figure 3.11 shows a data sets containing different patterns. The pattern of Figure 3.10 a) can be seen to the right and 3.10 d) to the left of this illustration. However, we do not get any false negatives here. The reason is that we get separating trajectories in one flow direction which is easy to detect, even though they converge in the opposite direction which we do not detect.

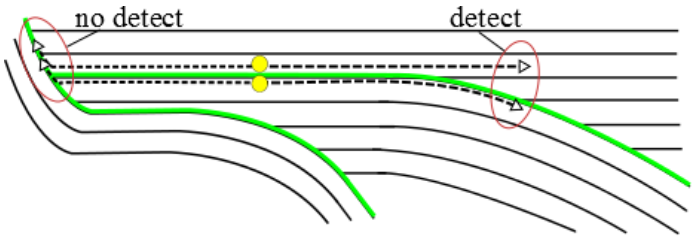


Fig. 3.11: A parallel area where the horizons are converging at one end and diverging at the other. Our method will detect the unconformity throughout the parallel and the non-parallel area.

In the last illustration (Figure 3.12), we see horizons that curve along a bell shape below the unconformity with horizontal horizons above. The algorithm will in this case detect a non-existing unconformity (the red line in Figure 3.12) that joins the top of the bell-shape. This is because trajectories starting on, and close to, the red line

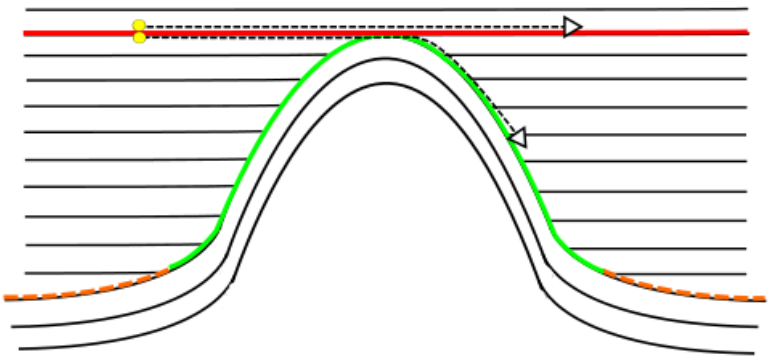


Fig. 3.12: Illustration of a situation where the unconformity detection algorithm will identify a non-existing unconformity (red line).

will separate when hitting the top of the bell-shape. The lower trajectories will move downwards along the bell-shape while the other trajectories continue horizontally. Their starting location is then wrongfully identified as an unconformity.

We have identified some scenarios where our presented algorithm is expected to have a successful outcome and some where it is more likely to fail by false positives or false negatives. Our algorithm is unique in the sense that no other existing unconformity attribute address the data on a global scale. This, together with the fact that it shows promising results in some of the above cases, makes our unconformity detection method well worth further research.

3.2.3 Suggestions on Postprocessing - From Volumes to Geometric Surfaces

Ideally, the output volume of our pipeline for detecting unconformities in seismic data is a collection of geometric surfaces. Such data could be input to a modeling tool for creating reservoir models. The final stage of a professional pipeline should include connecting the mapped points, outputted by the previous module, into polygon meshes. This is a compact representation and quick to display compared to a volume. Well established methods such as voxel labeling [MTHC03] or connected component analysis [BGS00] would be adequate for this task. As mentioned in the beginning of the chapter, we have not implemented this final stage of the pipeline, instead we display the output of the algorithm directly using direct volume rendering through raycasting.

Chapter 4

Implementation

This chapter will detail the implementation of our program. It is implemented as a set of plugins to VolumeShop. VolumeShop is an interactive prototyping framework for visualization developed by Bruckner and Gröller [BG05]. Our plugins are written in C++ and GL shading language (GLSL) and utilize OpenGL. The following section (4.1) will address the VolumeShop framework, and Section 4.2 will briefly cover the topic of programmable shaders which are used for controlling the GPU. The implementation details of our plugins and shaders are given in Section 4.3.

4.1 Description of The VolumeShop Framework

VolumeShop is a dynamic GPU-accelerated system for visualizing volumetric data at interactive frame rates. Many advanced illustrative rendering techniques are made with this framework. Its plugin architecture makes it an efficient environment for developing new visualization methods. Users create visualization components by implementing plugins. Plugins are compiled against the VolumeShop core, which defines the plugin interface and handles communication between the plugins.

Figure 4.1 shows a screenshot of the VolumeShop interface. The currently loaded plugins can be seen to the right in the plugin tab, and an interactive visualization of the data to the left. We set up this environment by adding a viewport, which show the graphics, to the interface. A viewer holds information needed to translate a 3D volume to the 2D viewport, and a compositor controls the assembling of the volumetric objects that are displayed. The background is in itself a volumetric ob-

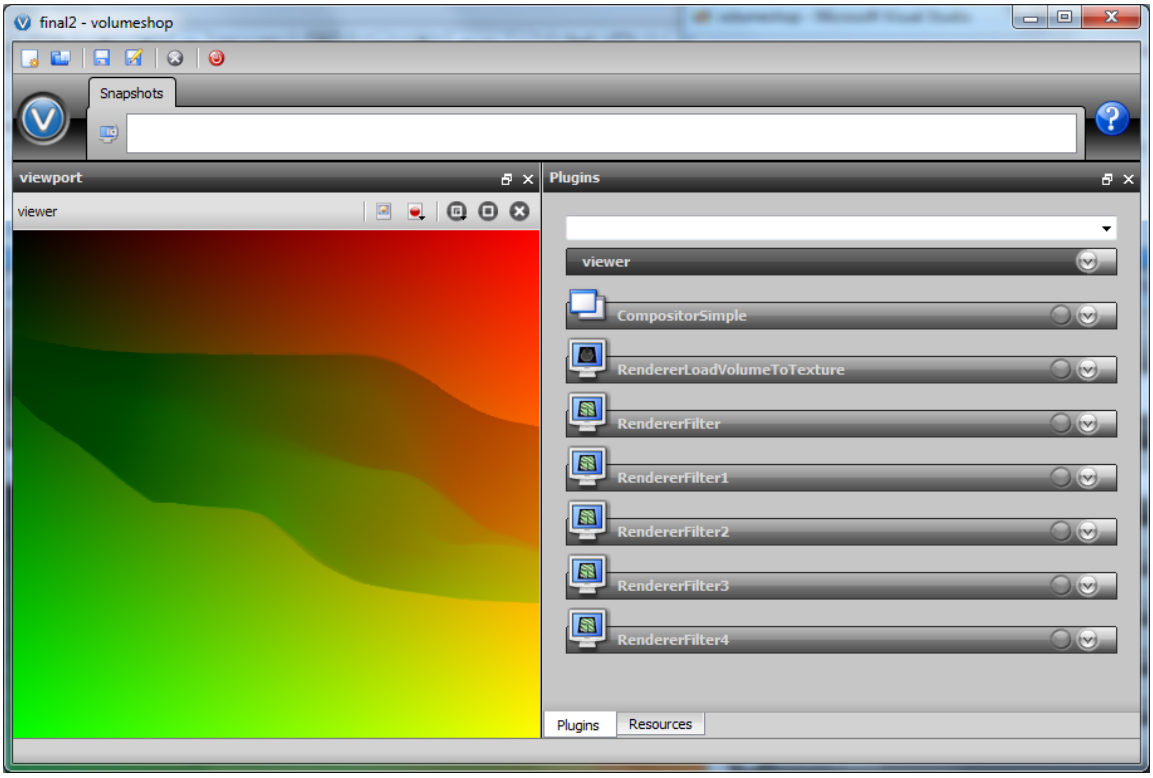


Fig. 4.1: Screenshot of the VolumeShop interface.

ject [BG05]. The other plugins seen in Figure 4.1 are our plugins and we will return to their functionality later in this chapter. All plugins added to the interface are executed in succession from top to bottom, and their renderings are displayed in the viewport in the same order.

4.2 GPU Programming

The use of GPUs for rendering is well known, but their massive computational power may also be used for general parallel computations. Parallel algorithms running on the GPU can achieve a great speed-up over similar CPU algorithms. Microsoft's DirectX and the open standard OpenGL are APIs for 2D and 3D rendering that take advantage of hardware acceleration. CUDA and OpenCL are some of the existing frameworks used with these APIs to write programs that are to be executed on the GPU.

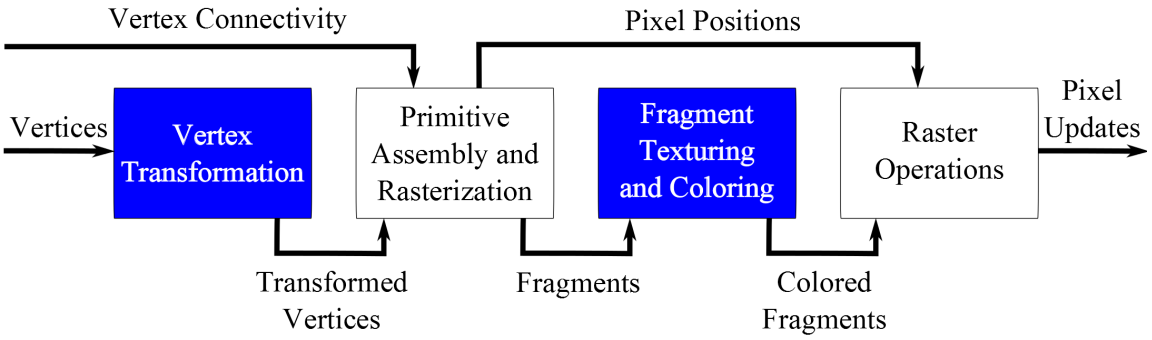


Fig. 4.2: The graphics pipeline.

Modern graphics cards process data according to the graphics pipeline. The pipeline contains the necessary steps to transform a representation of 3D primitives into a 2D raster image that is rendered to the screen. The performed actions include vertex transformations, color and connectivity computations, rasterization and interpolation. Figure 4.2 is a simplified diagram of the graphics pipeline showing the parts relevant to this thesis. The functionality of the stages shown in blue can be defined by programmable shaders.

The vertex shader is executed in the first stage of the pipeline. The input is the vertices constituting the 3D scene that is to be displayed. A vertex in this context is a set of attributes such as location, color, normal, and texture coordinates. The output of the vertex shader is used to assemble primitives in the next stage of the pipeline. The primitives are rasterized and the fragment information is passed on to the fragment shader. The fragment shader calculates a final color for every pixel on the screen. When textures are used, the fragment shader is responsible for combining pieces of the input fragment with texture samples. Typically, the fragment shader will calculate lighting effects, but is not limited to this. The fact that it is executed for each pixel simultaneously makes the fragment shader efficient for any type of calculations that can be done in parallel. A color and opacity (RGBA) is output for each fragment. The resulting image can be rendered to the display or to a texture via a framebuffer. A framebuffer is a hardware-independent abstraction layer between the

graphic operations and the video output. It reads from a memory buffer containing a complete frame of data.

We load our seismic data volume onto the GPU into a 3D texture, hereafter referred to as the data texture. To render our results, we input to our vertex shader four 3D texture coordinates and four 3D vertex positions. The texture coordinates correspond to the corners of a slice of the data texture. They are associated with the vertex positions positioned in the four corners of the viewport. By setting the right texture coordinates in the corners, we can render any slice through the 3D seismic data. We use the fragment shader for performing complex calculations on the data in the seismic slice.

4.3 Our Plugins and Shaders for Unconformity Detection

We have implemented plugins and shaders that do the following operations: load the data from file to the GPU, extract vector fields, smooth the data according to its structures, map the surface probability, detect height ridges, render the images, and export the data back to file. This section will first cover some memory management details, and then address the load-plugin and the generic filter-plugin. The filter-plugin contains several shaders and is responsible for the greater part of the functionality of our implementation.

4.3.1 Memory Management

We use OpenGL 3D textures to hold the data during the execution of the pipeline. A texture contains up to four storage slots (called R, G, B and A channel) for every voxel. Depending on the internal format, data put in a texture are formatted to

integers in the range $[0, 255]$ or with floating point precision in the range $[0, 1]$. We use the internal format of GL_RGBA32F. This means that every voxel has four channels, each with a float represented with 32 bits precision. This is the highest precision we can have for a texture. Although our pipeline outputs a 1-component selection volume where the voxels are either on an unconformity or not, we use 4-channel textures during processing. This is because the filter-plugins need as input, and outputs more than one value. The R and G channels hold the $x-$ and y-components of our flow field vectors, and the B channels are used for various information at each filtering stage. The A channels are set to the maximum value 1 throughout the pipeline. A full opacity value assures that we can see the output from each filter when it is displayed.

To acquire a data value from a texture, a 3D position is given for sampling. The $x-$, $y-$ and z-components of this position is in the range $[0, 1]$. If the given position does not coincide with a data grid point in the texture, OpenGL will use the closest data values around this position and interpolate them before the calculated value is returned. Our textures are set up for trilinear interpolation.

4.3.2 The Load-plugin

The load-plugin loads the data from disk into the GPU memory. The input may be a 1 to 4-component volume of floats or unsigned chars (integers in the range $[0, 255]$). The load-plugin also initializes the ping-pong textures. These are the two 3D textures that hold the data during the following processing steps. When the ping-pong textures are generated, they each get assigned an integer id that specify their location in the graphics memory. This information is needed by other plugins that will access the ping-pong textures for reading and writing. Therefore, we let the load-plugin output the memory location integers in its interface. Variables in the plugin interface slots can easily be linked across the plugins. The following filter-

plugins will be linked to the texture id slots of the load-plugin and thereby informed of the memory locations of which to read and write.

4.3.3 The Generic Filter-plugin

The filter-plugin is a generic plugin for processing a seismic slice. The specific filters are implemented as shaders and added to this plugin. Only one shader can be chosen for every instance of the filter-plugin. Therefore, one filter-plugin is added to the VolumeShop interface for every filter applied to the volume. The memory locations of the ping-pong textures must be linked in the interface to the right slots in the load-plugin. It is important that the linking is done in an alternating manner for each filter-plugin to ensure ping-ponging of writing. The out-texture in one filter-plugin must be the in-texture in the next.

Figure 4.3 shows the user interface of the filter-plugin. The user chooses a filter from a pull-down menu. This activates the respective shader. When the filter-plugin is executed, the in-texture is bound and its memory location information is passed on to the shader where the sampling occurs. The out-texture is bound to a framebuffer in the rendering to texture procedure. The processed texture can be viewed slice-wise, and the displayed slice is chosen by a slider. This is the current slice. The sliders appear in the interface when the user clicks on a value (integer or float number). Below the current slice, there are three more sliders for setting different filter parameters. The meaning of the sliders depend on the chosen filter.

If the option to render to texture is off, only the current slice is processed. If the option is activated, as many slices of the data texture as specified is processed. When less than all slices are to be rendered, the filter-plugin will process consecutive slices around the current slice. The option to process a subset of slices at each filter-plugin, allows 3D filtering in preview mode. For fast processing in preview mode, we want to process a minimum of slices. The minimum of slices that needs to be processed

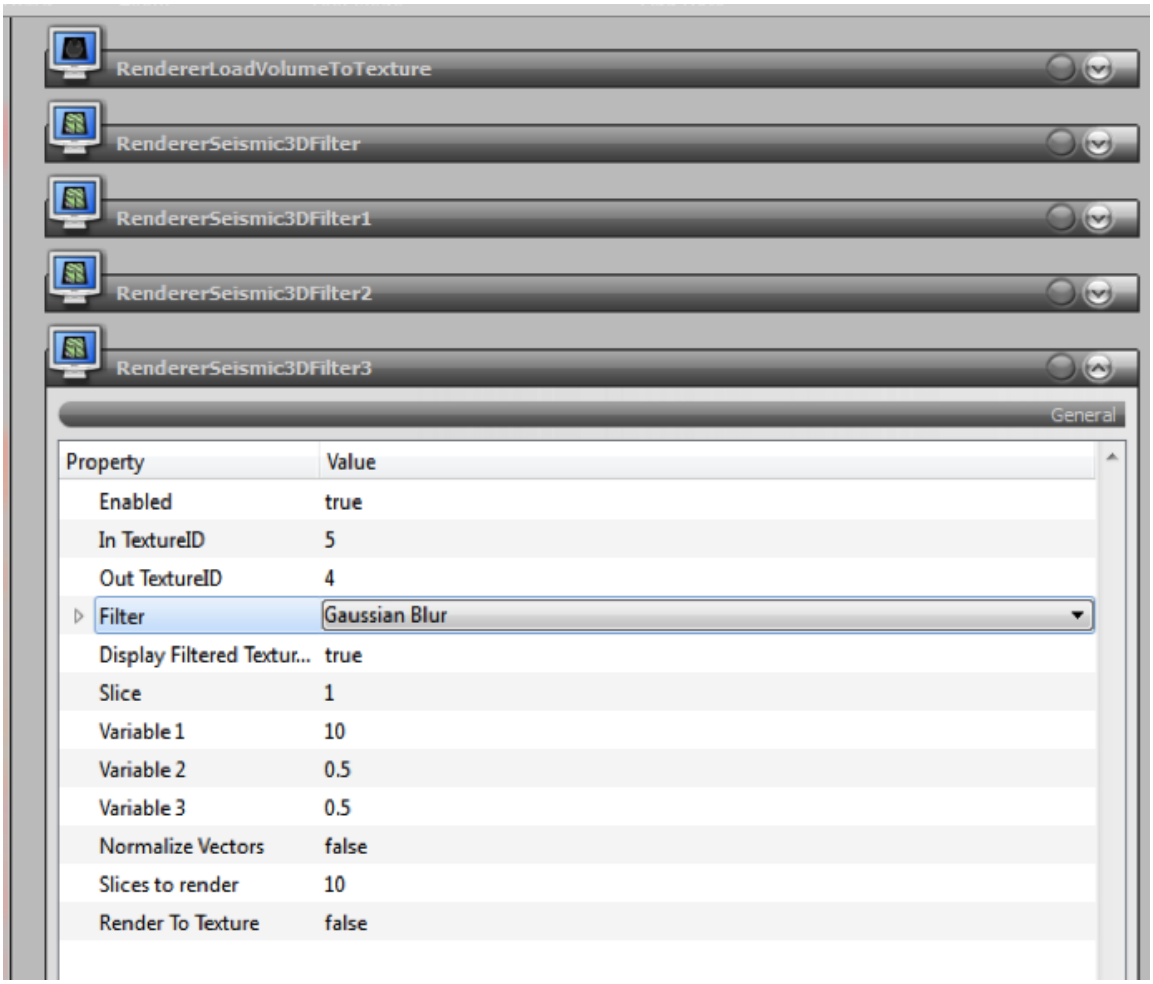


Fig. 4.3: The interface of the filter-plugin.

at each step varies when applying 3D filters. If two box filters of size 3^3 is used, the first filter take 5 input slices to output 3, and the second filter use them to output 1 processed slice.

4.3.4 The Shaders for Flow Field Extraction and Smoothing

An extracted flow field should be everywhere parallel to the reflection layers in the data. We use rotated gradients as the underlying vectors of our flow (as explained in Section 1.4.2). Our flow field is 2D, so it has been sufficient to implement 2D filters for calculating the gradients and smoothing the flow field. However, 3D filtering is possible in GLSL, and our processing pipeline supports 3D filtering.

All calculations are saved with float precision in the R, G, B and A channels of the 3D textures. The value of the vector components may vary in the range $[-1, 1]$ during calculations, but are scaled to the $[0, 1]$ color range before it is rendered. This is done so we can directly visualize the RGBA-vectors holding the flow field. A direct visualization of the flow field is seen as a color field that varies according to the flow vectors.

The Flow Field Extraction Shaders

We have implemented two alternative shaders for extracting the flow field. The first shader finds the gradients using the central difference method. It looks up four samples for every voxel. Two samples to calculate the $x-$component, and two samples to calculate the y-component of the gradient. The gradient of position (x, y) is found by:

$$\nabla f(x,y) = \begin{bmatrix} \frac{d(x-r,y)-d(x+r,y)}{2} \\ \frac{d(x,y-r)-d(x,y+r)}{2} \end{bmatrix} \tag{4.1}$$

r is the distance between the samples. This parameter can be set by the user. A greater distance can help avoid zero vectors in uniform areas. The simplicity and the few samples makes this filter fast but quite sensitive to noise.

Our second flow field extraction shader utilizes a Sobel filter. This is more robust than the previous method with respect to noise. It uses six samples in both the x and y direction to calculate a gradient. The masks looks as follows:

$$\nabla y\text{:} \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \qquad \nabla x\text{:} \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \tag{4.2}$$

Both of our flow field extraction shaders, rotate the calculated gradients. First a 90° counterclockwise rotation is performed, $(x, y) \Rightarrow (-y, x)$. If the resulting

$x-$component is negative, the vector is again rotated 180°, $(x, y) \Rightarrow (-x, -y)$.

The tangent is saved in the R and G channel for every voxel, and the gradient magnitude is saved in the B channel. The reason for explicitly storing this magnitude is that the initial gradient magnitude is used later by the structure oriented smoothing filter which runs in several iterations. While the gradients change during the structure oriented smoothing, their initial magnitudes are easily accessible in the b-components of the flow field vectors.

The Shader for Structure Oriented Smoothing

To smooth the extracted vector field, we have implemented the bilateral filter of Kang et al. [KLC09]. Kang et al. use this filter on images for extracting a vector that field follows the edges in the image. They obtain the initial vector field by rotating gradients 90°, as we do. Their filter is defined as follows:

$$\mathbf{t}'(\mathbf{x}) = \frac{1}{k} \int \int_{\Omega_\mu} \phi(\mathbf{x}, \mathbf{y}) \mathbf{t}(\mathbf{y}) w_s(\mathbf{x}, \mathbf{y}) w_m(\mathbf{x}, \mathbf{y}) w_d(\mathbf{x}, \mathbf{y}) d\mathbf{y} \tag{4.3}$$

k is the vector normalizing term. $\mathbf{x}$ is the position of the kernel center ($\mathbf{x} = (xpos, ypos)$), and $\mathbf{y}$ represents all positions in the kernel. $\mathbf{t}'(\mathbf{x})$ is the smoothed tangent vector output for position $\mathbf{x}$. $\phi(\mathbf{x}, \mathbf{y})$ is a sign function that temporarily reverse the direction of the tangent $\mathbf{t}(\mathbf{y})$ if the angle between $\mathbf{t}(\mathbf{x})$ and $\mathbf{t}(\mathbf{y})$ is greater than $\pi/2$. The weight functions w_s, w_m and w_d relates to space, gradient magnitude and tangent direction respectively.

The spatial weight function is defined by

$$w_s(\mathbf{x}, \mathbf{y}) = \begin{cases} 1 & \text{if } \|\mathbf{x} - \mathbf{y}\| < \mu \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

where μ is the radius of the filter. This sets the size of the filter mask. A small mask

will be more accurate in preserving edges than a large mask, but it will smooth the vectors less. In our implementation, we connect the mask size to one of the GUI sliders. The user can then decide the mask radius r with a maximum of $r = 11$ samples (voxels).

w_m is a weight in the range $[0, 1]$ that relates to the initial gradient magnitudes. $\mathbf{g}(\mathbf{x})$ is the initial gradient of position $\mathbf{x}$, and w_m is found by

$$w_m(\mathbf{x}, \mathbf{y}) = \frac{\|\mathbf{g}(\mathbf{y})\| - \|\mathbf{g}(\mathbf{x})\| + 1}{2} \tag{4.5}$$

This indicates that neighboring points with a greater gradient magnitude than that of the center position $\mathbf{x}$, gets bigger weights. It ensures the preservation of the dominant edge directions. Every iteration of the filter uses the initial gradient magnitudes to find w_m. We continue to store this magnitude in the $b-$component of every voxel. After the last iteration, the $b-$component will be set to zero.

The final weight w_d, is the absolute value of the dot product of the normalized tangents:

$$w_d(\mathbf{x}, \mathbf{y}) = |\mathbf{t}(\mathbf{x}) \cdot \mathbf{t}(\mathbf{y})| \tag{4.6}$$

Here, bigger weights are given the closer the two tangents align (with a maximum of 1), and a weight of 0 is given when they are orthogonal. This promote smoothing among similar orientations, and thereby preserve the corners in the underlying image. This weight will also prevent smoothing across zero vectors. See [KLC09] for a more detailed description.

Pseudecode for our implementation of the filter by Kang et al can be found in the Appendix. The shader containing this filter executes a double for-loop that considers every position within the filter mask. For every mask position, the according tangent vector is sampled from the flow field and the calculations are done with regard to the center position of the mask. The mean of the smoothed vectors at all mask positions

is the new and smoothed tangent for the current position. This tangent is stored in the R and G channel of the current voxel. The initial gradient magnitude (sampled from the B channel of the center position), is again put in the B channel of the current voxel.

The Shaders for General Smoothing

We have implemented two other shaders for smoothing for experimenting and identifying the best one. One consist of a Gaussian blur filter and the other is a basic mean filter. The Gaussian mask is of size 7×7 and the mean filter is a $n \times n$ mask with odd sides. The user can set the size of the mean filter mask in the user interface and can increase the Gaussian mask by using several filters after each other. These filters can be used to smooth the data before the gradients are calculated. It will eliminate some noise and minimize areas with zero gradients. The type and amount of noise matters to which of these filters are the better choice. The filters may also be used to smooth the resulting vector field. This is faster than the structure oriented smoothing, but less accurate.

4.3.5 The FTLE_Unconformity Detection Shaders

The FTLE_unconformity detection shader holds the implementation of our unconformity detection algorithm. In short, it calculates four trajectories in forward and backward directions of the flow field, finds the greatest separation of the trajectories by FTLE, and maps it to the R channel of center voxel of the seed positions. We will now look at our methods for finding the trajectories, before our method for the FTLE calculation are explained.

The Methods for Calculating Trajectories in a Flow Field

Particles are advanced along the flow field by integration, and we are using the Runge Kutta 4th order (RK4) integration method [JST81]. When choosing an integration method, it is necessary to weigh the accuracy with the expenses of computation. If the step size of the RK4 method is set to h, the error per step is in the order of h^5, while the total accumulated error has order h^4. Because of the error accumulation, a small step size is desirable. The user can set the step size and the number of steps the particles advance along the vector field in the GUI and see the results immediately. The balance between the accuracy of the method and the performance speed lies in the choice of these two parameters. They also affect to what extent the data is addressed locally or globally.

We have implemented a method called $RK4$ that uses the Runge Kutta 4th order method to calculate the next position of a trajectory. The input is a position and a step size. If the step size is positive, the next position is calculated from the forward flow, if it is negative the flow vectors are turned in the opposite direction. Before the $RK4$ method returns the new position, it checks if the new position is within the boundaries of the flow field. If not, it returns its input position.

To find the last position of a trajectory, we have implemented a method called $GetEndOfPath$. It uses the number of steps and the step size parameters set by the user. As input it takes a starting position and a flow direction. The direction is either -1 or 1, and this parameter is multiplied with the step size. The $GetEndOfPath$ method calls the $RK4$ method to find the next position of the trajectory. It continues to do this until the set number of steps is achieved or the $RK4$ method returns its input position. Only the end position of the trajectory is needed in the further calculations, so all the intermediate positions are discarded. The final output is a 3D vector containing the end position and the number of performed steps. Pseudocode for the $GetEndOfPath$ method can be seen in the Appendix.

The Method for Calculating the FTLE

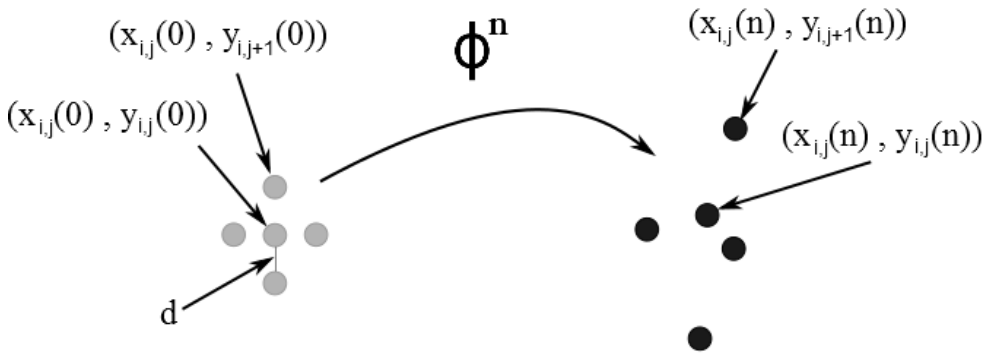


Fig. 4.4: A flow map gradient is calculated from trajectory positions after n number of trajectory steps. Trajectories are found by utilizing the Runge-Kutta 4^{th} order method.

The finite-time Lyapunov exponent (FTLE) is a scalar value characterizing the rate of separation between trajectories in a flow field. We briefly explained how the FTLE is calculated in Section 2.3.1. We will repeat it here, more in depth, and specify our implementation of the procedure. Figure 4.4 illustrates how the distance between particles seeded in a flow field may change as they move with the flow over time. A steady flow has a constant flow structure, and a particle seeded at time t moves along the same trajectory as a particle seeded from the same point at time $t + n$. In our method we seed four particles for each voxel of the volume. They are seeded in the formation of the gray dots in Figure 4.4 but without the center dot. After the particles are advanced along the flow field for an equal number of steps, their new positions are used to calculate a flow field gradient. The first-order partial derivatives constituting this gradient is the Jacobian matrix

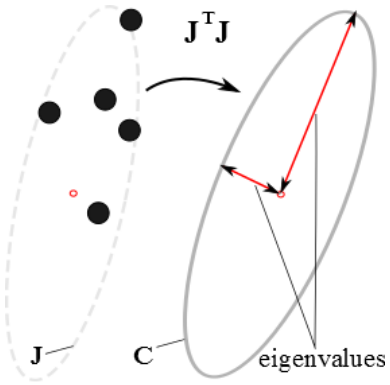


Fig. 4.5: The right Cauchy-Green deformation tensor is found by left-multiplying the Jacobian with its transpose.

$\mathbf{J}$. For position (u, v), $\mathbf{J}$ is defined as:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial u}{\partial \mathbf{x}} & \frac{\partial u}{\partial \mathbf{y}} \\ \frac{\partial v}{\partial \mathbf{x}} & \frac{\partial v}{\partial \mathbf{y}} \end{bmatrix} \tag{4.7}$$

In Figure 4.4 is this gradient related to the position $(x_{i,j}(n), y_{i,j}(n))$. We will instead relate it to the center voxel of the seed points. Therefore, we do not need to calculate a trajectory from this center voxel. We calculate the Jacobian matrix as follows:

$$\mathbf{J} = \frac{1}{2d} \begin{bmatrix} u_x^1 - u_x^2 & v_x^1 - v_x^2 \\ u_y^1 - u_y^2 & v_y^1 - v_y^2 \end{bmatrix} \tag{4.8}$$

where u^1 and u^2 are the end positions of the trajectories seeded close to the center voxel in the x direction, and v^1 and v^2 the end positions of the trajectories seeded close to the center voxel in the y direction. The x and y subscripts refers to the $x-$ and $y-$components. d is the distance showed in Figure 4.4.

Next, we left-multiply $\mathbf{J}$ with its transpose. This results in a rotation-independent tensor called the right Cauchy-Green deformation tensor $\mathbf{C}$. In mathematical terms, $\mathbf{C} = \mathbf{J}^T\mathbf{J}$. It is symmetric, and, by the spectral theorem, its eigenvalues are real and its eigenvectors are mutually orthogonal. These eigenvectors define the principal axes of $\mathbf{C}$. The eigenvalues and the principal axis draw an ellipse as illustrated in Figure 4.5. We are interested in the major eigenvalue. If

$$\mathbf{C} = \begin{bmatrix} a & c \\ c & b \end{bmatrix} \tag{4.9}$$

the major eigenvalue $\lambda_{max}(\mathbf{C})$ is found by

$$\lambda_{max}(\mathbf{C}) = \frac{t}{2} + \sqrt{\frac{t^2}{4} - h} \tag{4.10}$$

where $t = a + b$ and $h = ab - c^2$.

The major eigenvalue is used to calculate the FTLE as follows:

$$FTLE = \frac{1}{2n} \ln \lambda_{max} (\mathbf{C}) \tag{4.11}$$

n is the number of steps in the trajectories.

We have implemented the calculations above in one shader. Trajectories are calculated and the FTLE is found for both directions of the flow. The maximum of the two FTLE values is stored in the R channel of the current voxel. Pseudocode for our FTLE calculation method is to be found in the appendix.

The Methods for Trajectory and FTLE Calculations Extended to 3D

We have extended the implementation above to a method that work on the data in 3 dimensions. We have implemented a shader, with a few amendments of the above methods, that can be used for FTLE calculations in a 3D flow field. This shader calculates six trajectories in each flow direction, instead of four. That is, two more in the z-direction. Extending the Jacobian and the deformation tensor calculations from 2D to 3D is straight forward, but finding the major eigenvalue is more complex. For this we have used a GLSL method by Mario Hlawitschka of the OpenWalnut Community [200]. The FTLE is calculated as in the 2D implementation (see Equation 4.11).

Optimized Method for Calculating the FTLE

Our 2D FTLE method calculates four trajectories for every voxel. This means that the trajectories are recalculated many times (see Figure 4.6). We have made an optimized implementation of our algorithm that calculates the trajectories only once in each flow direction for every voxel. In this implementation, the trajectory calcu-

lations and the FTLE calculations are done in separate shaders. The first shader stores in every voxel the end position of the trajectory starting from that voxel, and the number of steps. Next, the shader responsible for the FTLE calculations samples the end positions from the in-texture and use them directly in the FTLE calculations. There are three values for each flow direction that must be stored when the trajectories are calculated. The $x-$ and $y-$component of the end position and the number of steps. We have only the four channels available. The solution has been to choose either the forward or backward flow direction when running the optimized implementation. The flow direction may be chosen in the filter-plugin interface. The disadvantage is that you only see "half the picture" when the FTLE is found in just one flow direction. However, it is a good implementation for parameter adjusting as it updates the displayed slice instantly.

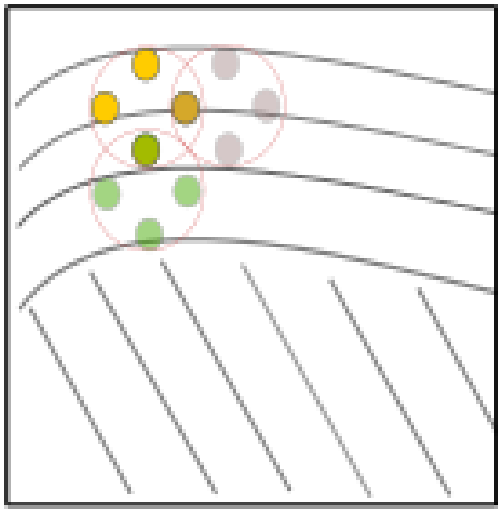


Fig. 4.6: There are many overlapping trajectories when four particles are seeded for every voxel.

4.3.6 The Shader for Extracting Height Ridges

To say that the FTLE is a mapping of the probability each voxel has to be on an unconformity, is a partial truth. The Lagrangian coherent structure (LCS) borders is defined as the height ridges of the FTLE field [Hal02]. These borders coincides with unconformities in our flow field. Therefore, in local neighborhoods the FTLE values can be seen as an unconformity probability. The height ridges will not have the same values throughout the data, so we have implemented a simple height ridge extraction shader. For each voxel the shader fetches two samples in the y direction. One on

either side of the current voxel. The distance between the sampled positions can be set from the user interface. The shader tests if both of the samples hold a smaller FTLE value than the current voxel. If not, the same is done in the x direction. If the test is positive in either the y or x direction, the current voxel is on a height ridge and therefore stored as a selected voxel. After processing the FTLE field volume by this shader we end up with a selection volume where the voxels are either on or off (selected or not selected). The selected voxels constitute the detected unconformities. Pseudocode for this method is found in the Appendix.

Chapter 5

Results

In this chapter we show our results by looking at test sets processed by our pipeline. For testing our algorithm, we have created 3D data sets from 2D seismic images since we have not been able to obtain real seismic data volumes containing unconformities. We have implemented image processing methods for smoothing the data and for extracting a vector field. The test sets have been useful for testing our algorithm, and for getting a feel of how our methods actually perform. Although we do not have real 3D seismic data, our pipeline readily supports it.

Our test sets are created by stacking copies of seismic cross section images into volumes. Our algorithm work on one slice at the time, and without a variation in z-direction, we do of course get the same result throughout the volume. However, producing test volumes instead of just running 2D image tests, assures that our implementation is ready for 3D data. The first test sets resemble the images of the thought cases in Section 3.2.2. We will check if our methods behave in practice how we have predicted in Cahpter 3. For the second test, we have created a data volume from the seismic image in van Hoek et al.'s article [vHGP10]. To determine the accuracy of our algorithm, we need to know what we ideally should detect for a real seismic example. Van Hoek et al. have included a manual interpretation of the seismic data which will function as a reference. In the same section (Section 5.2) we show output from each processing step, as well as results for different parameter settings. For a basic test set with a variation in z direction, we procedurally created a 3D vector field where the flow moves in different directions above and below a delimiting surface. The result is presented in Section 5.3. The last section of this

chapter give some performance details.

5.1 Synthetic Tests

We would like to see if our algorithm behaves as we expect in some simple situations. On that account, we have created data sets resembling the synthetic scenarios of Section 3.2.2, namely the figures 3.9 a) and c), 3.10 a) and d), and 3.12. For our synthetic test sets we have used black lines to conceptually represent the reflection layers. We have also blurred the images and thereby enabling them as input to our pipeline. The blurring produces an image with representative gradients in every voxel from the conceptual images. The following figures show data slices processed by our pipeline. We have blended the output slice with the input slice. The green lines are detected unconformities. The image each data set represents is included in the figures over the appropriate output slice.

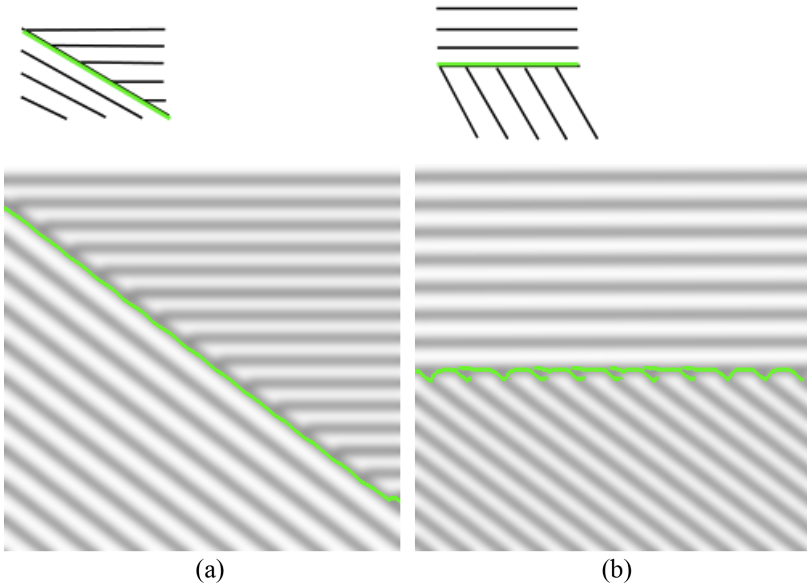


Fig. 5.1: The unconformity is rightfully detected in both tests.

The first two tests gave a successful outcome. Output slices are seen in Figure 5.1. A separation is detected when the trajectories are calculated in the flow moving along the lines from left to right. When the calculations are done in the opposite flow

direction, no separation is detected. Our algorithm looks for a separation in both flow directions, so the unconformities are detected in these tests. The FTLE equation (Eq. 4.11) contains a normalizing factor, and therefore we also detect the unconformity in the area close the the right edge of each data set. In b) the unconformity is detected, although slightly wavy instead of straight. The reason is that the blurring of the original image has created black circular areas at the bifurcations.

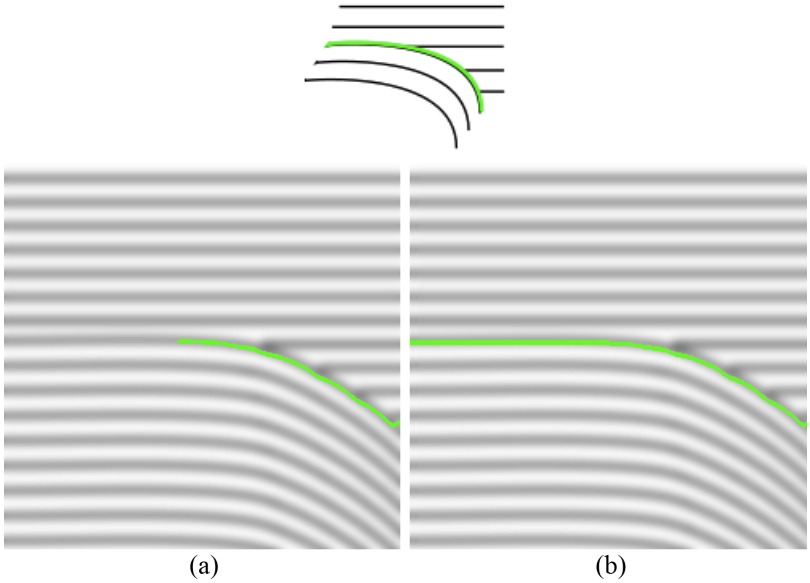


Fig. 5.2: a) The unconformity is rightfully detected in the parallel neighborhood. b) A higher number of trajectory steps will result in an unconformity detection further into the parallel area.

Figure 5.2 shows output slices from our next test set. As expected, the unconformity were detected in the parallel area. In a) we let our algorithm calculate shorter trajectories than in b). The results show that with enough number of trajectory steps, the unconformity is detected throughout the parallel neighborhood. To detect the full length of unconformities, we can fix the number of trajectory steps as a function of the image resolution and the step size. This would assure a global addressing of the data.

In the next two test sets, seen in Figure 5.3, we expected different unsuccessful outcomes. In the first test, we expected our algorithm to result in a false negative by not detecting an existing unconformity in a parallel neighborhood. Figure 5.3 a)

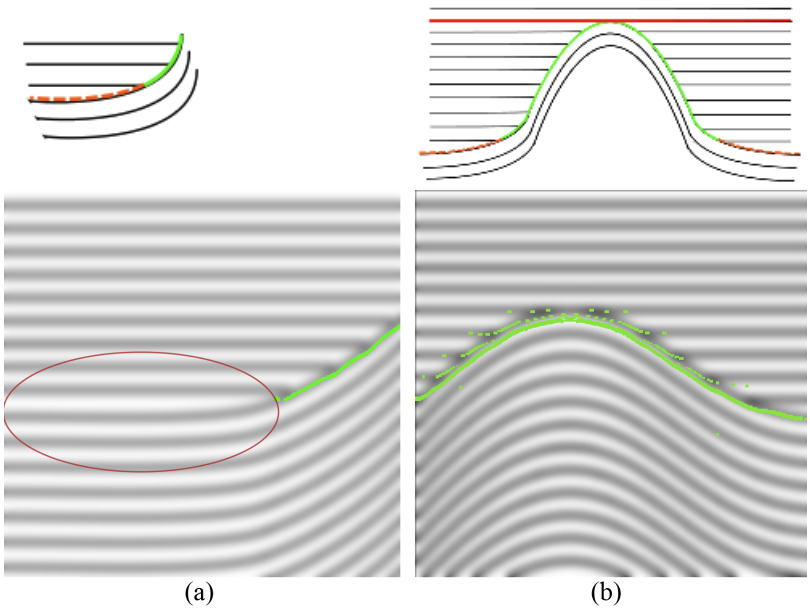


Fig. 5.3: a) The unconformity is wrongfully not detected in the parallel area indicated by a circle. b) A more successful outcome than expected. A false unconformity line is not detected in this data set.

shows that our presumptions were right. Trajectories starting from positions close to the unconformity in the parallel area will not separate in either flow direction. Only the part of the unconformity laying within the non-parallel neighborhood was detected in this test.

In the second test, we expected a false positive. This data set contains lines representing bell-shaped reflection layers below an area of horizontal layers. We predicted our algorithm to wrongly detect a non-existing horizontal unconformity joining the top of the bell-shape. Surprisingly, our method rightfully detected the unconformity without additionally detecting the expected false positives (see Figure 5.3 b)). There are some minor false positives in form of dots above the unconformity, but no false unconformity lines. The dots looks to coincide with the black areas of the bifurcations and are likely to be attributed to the way we have created the synthetic data.

To investigate this further, we created a vector field similar to that extracted by our pipeline in last test set above (Figure 5.3 b)). This vector field does not contain the artifacts at the bifurcations that the initial blurring has created in the above test sets.

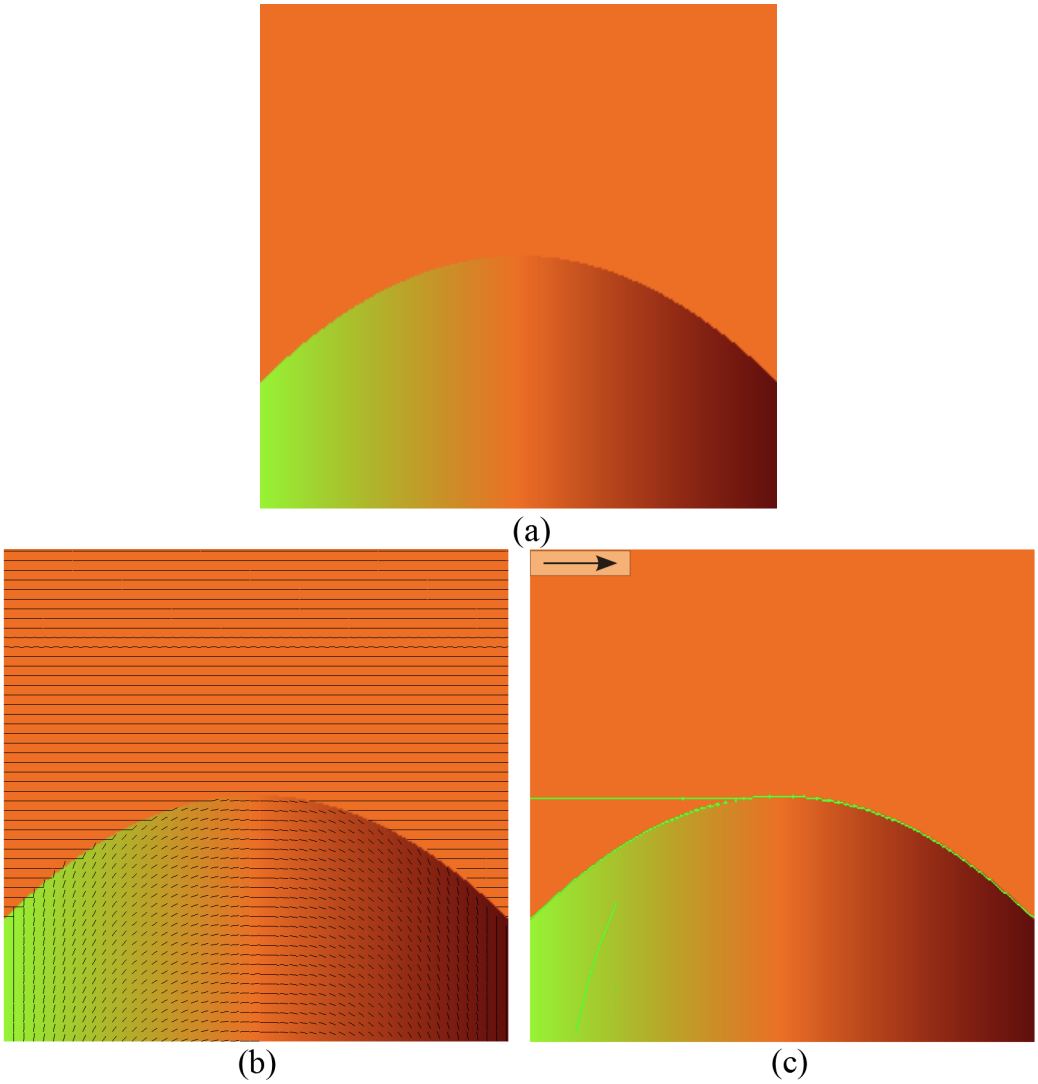


Fig. 5.4: a) The input vector field. b) The underlying vectors indicated by short lines. c) Output from our unconformity detection algorithm in one flow direction (indicated by an arrow). The unconformity is rightfully detected. In addition, false positives above and within (green area) the half circle is wrongfully detected.

Figure 5.4 a) shows a slice of the volume containing our vector field. The vectors are stored in the R and G channels of a texture, and are therefore seen as colors. In b) we have added short lines that indicate the underlying vector field. c) shows a resulting slice from processing this data set by our unconformity algorithm in on flow direction. The unconformity is rightfully detected. Since the data is processed in only one flow direction, we conclude that close particles moving upward along the edge of the half circle will separate after they have reached the highest point.

Because of the interpolation that occurs when we sample the data, close particles

may become slightly separated in this vector field. With two horizontally aligned particles, the left most particle is more affected by vectors to the left than the right particle, which is more affected by the vectors to the right. Within the half circle, this will result in a slight separation. Particles seeded in the green area will move upward, and many will reach the half circle edge and continue along it. The false positive we see in this area is detected since close particles seeded along the false positive line will separate slightly on there way towards the half circle edge. There they will continue along trajectories that separate after the highest point of the edge.

The wrongfully detected unconformity seen as a horizontal line, shows the output we expected in Section 3.2.2. Here, it is the vertically aligned particles, seeded close to the false positive, that are responsible for the detected separation. The lower particle will hit the top point of the half circle and continue downward along its edge, while the higher particle will move horizontally.

5.2 Seismic Test

Fig. 5.5: A seismic cross section containing unconformities.

Figure 5.5 shows the seismic image we use in this test. The data in the center is a seismic cross section containing several unconformities. The first processing step is to

extract a flow field. We use a Sobel mask to obtain the gradients which we then rotate. In the gray, uniform area of this data set, the gradients will be zero. Hence, our flow field is not defined here. This is not a problem, as we are not looking to extract any information from this area. Within the part of the test set containing the seismic data, the gradients are well defined in most voxels. However, the pattern of the data may cause the Sobel mask to result in zero gradients in some voxels. This is a problem for our algorithm. Any trajectory hitting a single voxel with a zero vector will

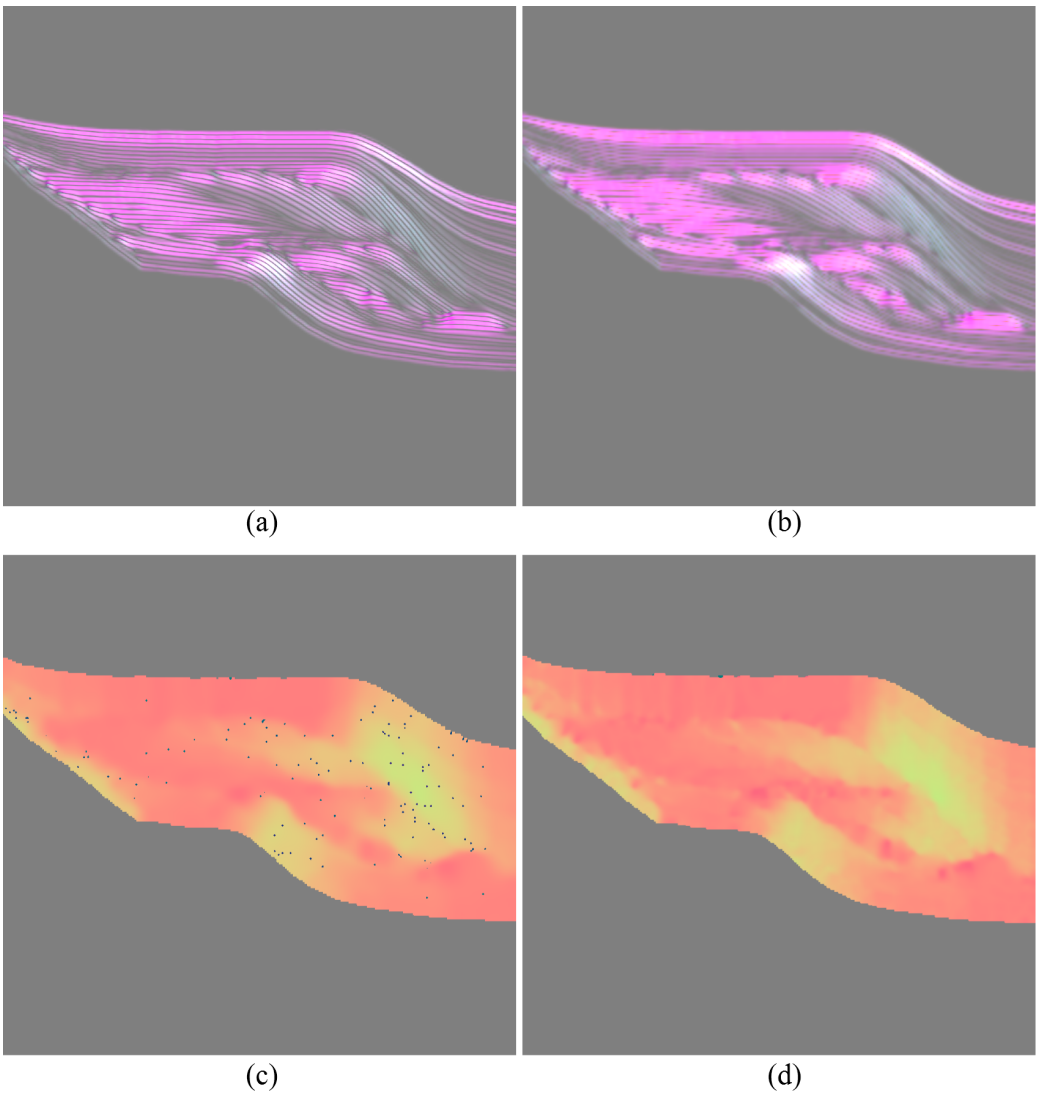


Fig. 5.6: a) The tangent vector field found by rotating vectors extracted by a Sobel filter. b) The tangent vector field after smoothing by a 3×3 mean filter. c) The result after performing SOF on a) and then normalizing all vectors to length 1. d) The result after performing SOF on b) and then normalizing all vectors to length 1.

stop. This will causes close trajectories to separate which results in false positives for our algorithm. We smooth our flow vectors by the structure oriented filter (SOF) by Kang et al., however, this filter does not smooth away the zero vectors. Therefore, we have found it necessary to smooth the extracted vector field by a 3×3 mean filter before we apply the SOF. The effect on the flow field, with and without the use of the mean filter, can be seen in Figure 5.6 c) and d).

The unnormalized tangents seen in Figure 5.6 a) and b) have a purple hue because the magnitude of every vector is explicitly stored in the B channel of the voxel holding that vector. The SOF is an iterative filter. It can be applied several times to get the wished result. Each iteration use the initial gradient magnitudes in its calculations, which is why we have stored these magnitudes in the B channels. After the last SOF iteration, the initial gradient magnitudes are no longer needed and the B channels are set to zero. The vectors are also normalized to 1. By this we avoid our algorithm to result in false positives attributed to a separation of close particles that move at different speeds. We achieve good results with three iterations of the SOF. Figure 5.7 show the output from applying the SOF one, two, and three times. We have used a mask size of 5×5 for every iteration.

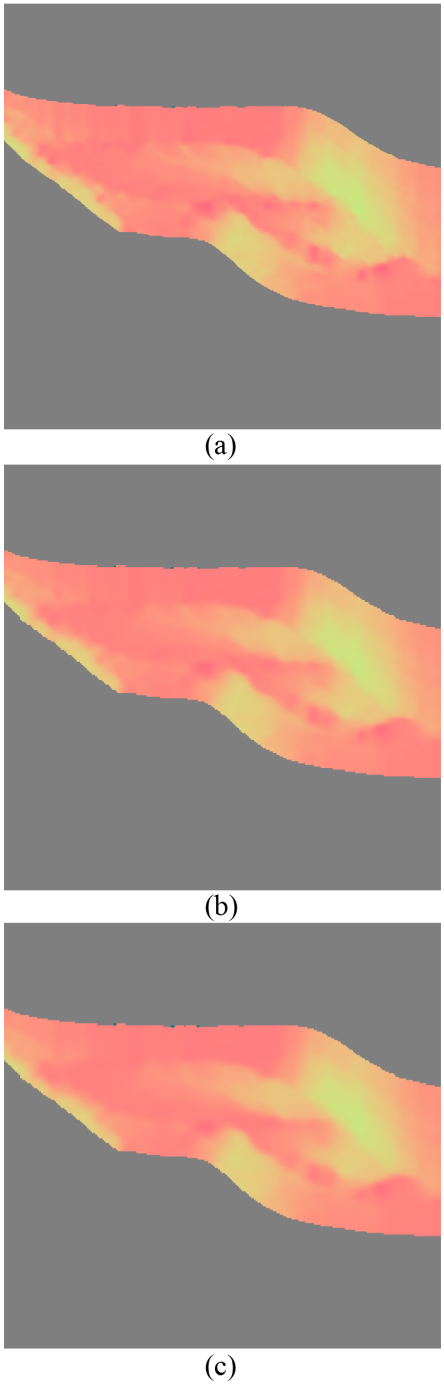
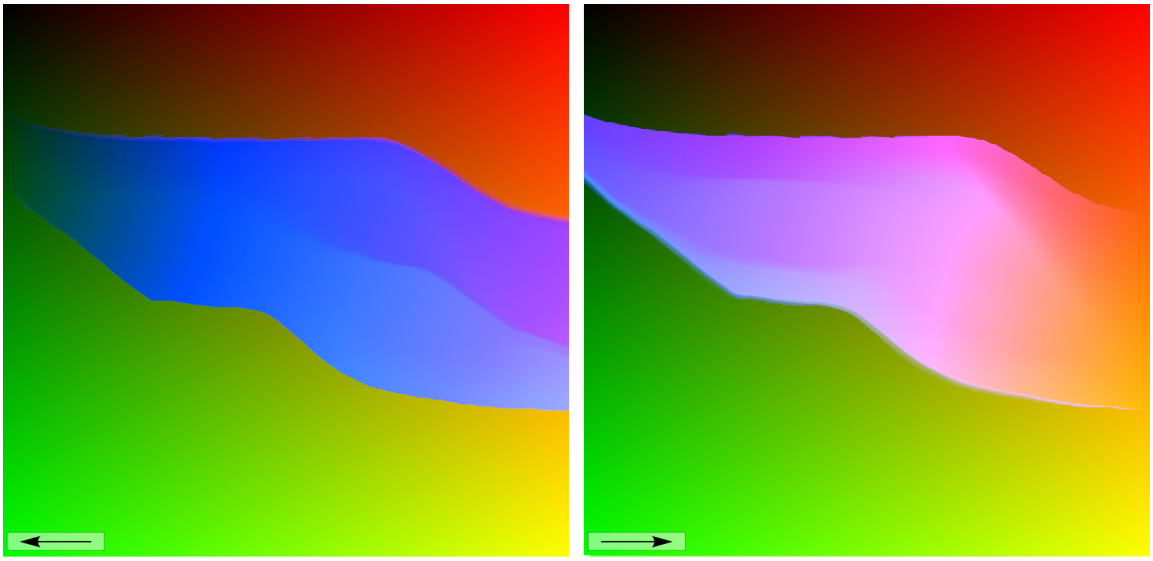


Fig. 5.7: a) Output after one iteration of the SOF. b) Output after two iterations of the SOF. c) Output after three iterations of the SOF.

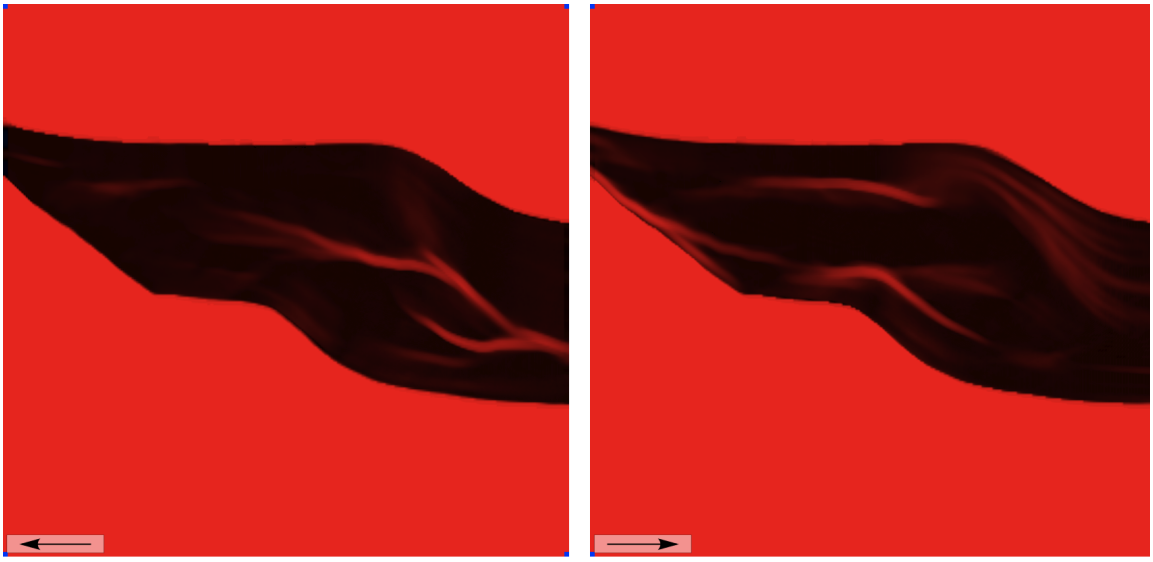


(a) (b)

Fig. 5.8: One trajectory is calculated from each voxel. The end position is mapped to the R and G channel of the starting voxel. The number of steps are mapped to the B channel. a) The flow moves from right to left. b) The flow moves from left to right.

We now have a well defined flow field in the area containing the seismic data. The next processing step is to calculate trajectories in this flow field. To be able to show output from this processing step, we have used our optimized implementation where the trajectory and the FTLE calculations are done in two different instances of a filter-plugin. We can then display the output texture holding the results from the trajectory calculations. In this implementation, the trajectories and the FTLE are found in one flow direction at the time. We show the output from each flow direction in separate images. The flow directions are indicated by arrows.

A trajectory is calculated from every voxel in the data. Figure 5.8 shows a color mapping of every trajectory's end position to its starting position. The (x, y) coordinates of an end position is put in the R and G channel of the trajectory's starting voxel. The number of trajectory steps is stored in the B channel. Outside the defined flow field, the seeded particles do not move. Here, the B channel is zero and the R and G channels contain the (x, y) coordinates of its own voxel. Consequently, the color map seen outside the flow field is the same as the side of a color cube. The black



(a) (b)

Fig. 5.9: The rate of separation of trajectories seeded one voxel apart is calculated by FTLE and mapped to the R channel a) in the backward flow and b) in the forward flow.

corner voxel stores the coordinates $(0, 0)$, and the $(1, 1)$ coordinates are stored in the yellow corner voxel. Before storing the number of performed steps in the B channel, we scale this number to the $[0, 1]$ range. Within the flow field, we thus get a bluish color. Towards the edge of the image, the blue color gets weaker as the trajectories will have a decreasing number of steps the closer to the edge of the flow field that particle is seeded. In Figure 5.8 the number of trajectory steps is set to 500, and the step size is 0.1. Although subtle, it is possible to see some variations in the color map of the flow field that differ from the otherwise continuous change.

Next in our algorithm, we calculate the FTLE. Figure 5.9 shows the output from the FTLE calculations mapped to the R channel of every voxel. The end positions are here sampled from the in-texture at a distance of one voxel. The number of steps used as a scaling factor of FTLE is sampled from the same position as the mapped voxel. A more intense red color means a greater separation between the trajectories starting from that voxel. We have set the R channel to 1 outside the defined flow field although no separation occurs here.

The borders of the Lagrangian Coherent structures (LCS) in a flow field is the height

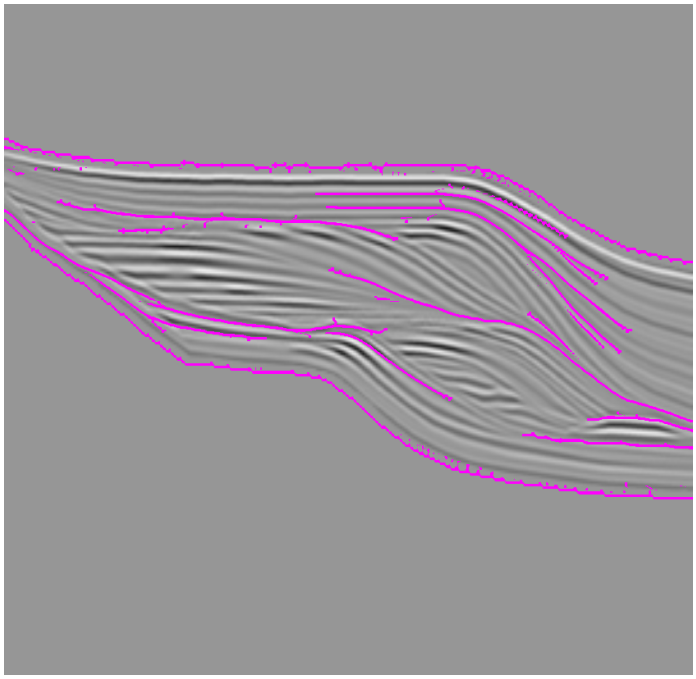


Fig. 5.10: Output from our unconformity detection pipeline.

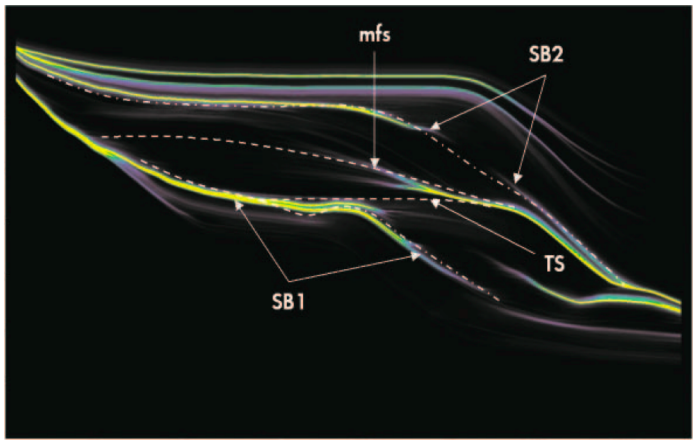


Fig. 5.11: Reference image. The unconformities are manually enhanced by dotted lines.

ridges of the FTLE field [Hal02]. In the color map above (Figure 5.9) we can easily see height ridges in the higher ranges of the FTLE values. However, it is not the voxels with the globally highest intensity that we look for, but the voxels with the highest intensity in a local neighborhood. Therefore we process the FTLE volumes by ridge extraction. For a co-visualization together with the input data, we output the height ridges blended with the original seismic image. A resulting slice can be seen in Figure 5.10. Extraction in both flow directions are present in this image. The reference image from van Hoek et al.'s [vHGP10] article is included for comparison

(Figure 5.11). We can see that the LCS borders of our flow field coincides with the unconformities in most voxels. A few false positives and false negatives are also seen. However, it is a good chance that a better height ridge extraction algorithm and optimized parameter settings will give even better results than our output in Figure 5.10. Our method for extracting the height ridges considers only three or five samples for each voxel, and it does not account for varying widths of the ridge structures. Therefore, it is possible that improving our height ridge method would result in a more accurate detection of the unconformities.

5.2.1 Different Parameter Settings

Different parameter settings for our unconformity detection algorithm gave different results. Figure 5.12 and 5.13 show the output slices of six different parameter settings. For a better overview, the different parameter settings are presented in a table.

Figure	5.12(a)	5.12(b)	5.12(c)	5.12(d)	5.13(a)	5.13(b)
Num. Steps	50	150	1000	1000	10	175
Step Size	0.8	0.8	0.1	0.1	5	0.5
Seeding Radius	0.5	0.5	5	0.5	0.1	1

The difference between the first two output slices is the number of trajectory steps, a) has a step number of 50 and b) has a step number of 150. The size of the data slice is 256×256. 150 steps with a step size of 0.8 means that the trajectories extend over an area of 120 voxels, about half the width of the image. Figure 5.12 b) show more rightfully detected unconformities than a), but also more false positives. However, in the area towards the lower right of the data in a) is an unconformity rightfully detected that is not present in b). It is probable that the higher number of steps has lead to a rejoining of particles that were separated at on point, and thus this unconformity is not detected in b).

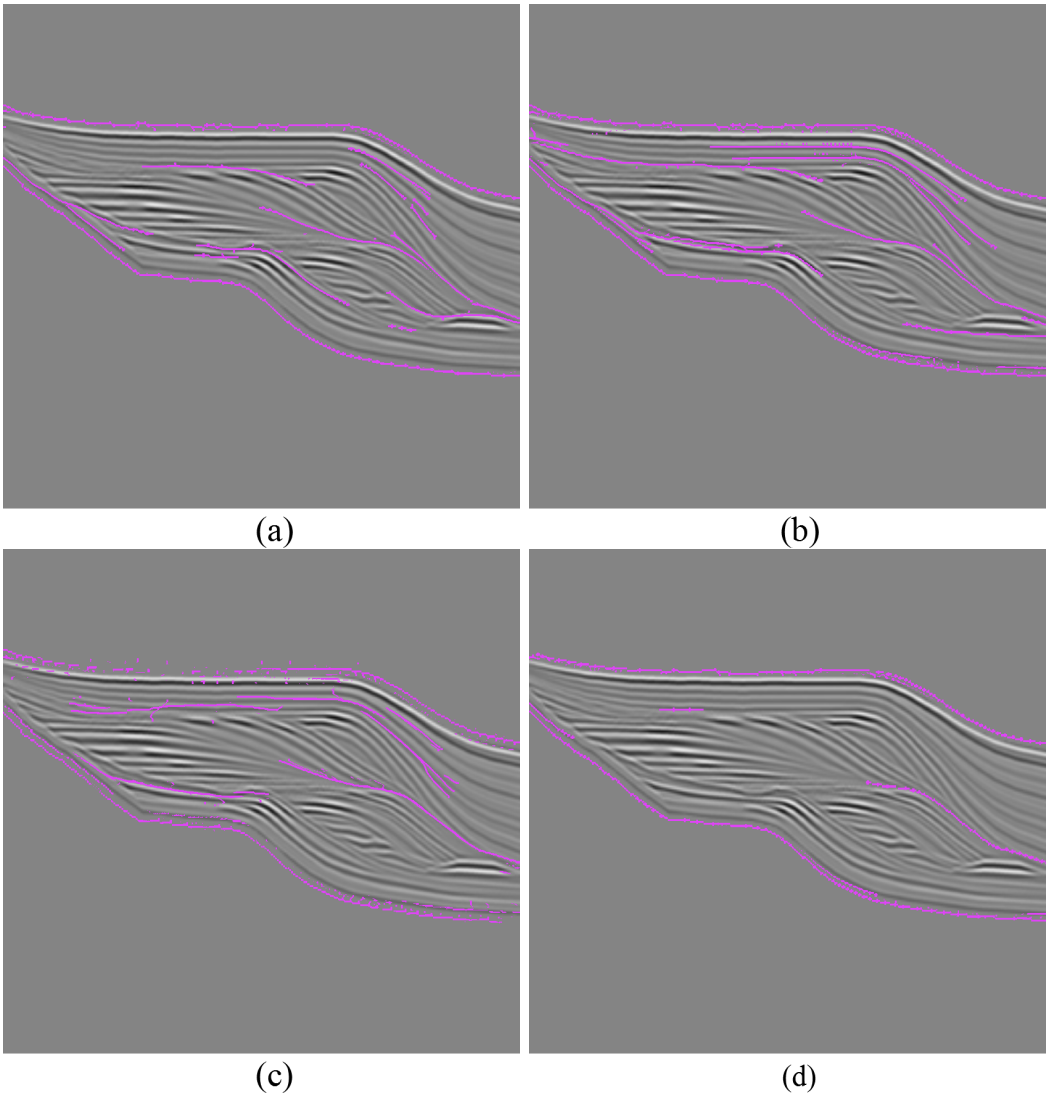


Fig. 5.12: a) The result from using 50 trajectory steps. b) The result from using 150 trajectory steps. c) The trajectories are calculated with high precision and a seeding radius of 5. d) The trajectories are calculated with high precision and a seeding radius of 0.5

In c) and d) are the length of the trajectories 100 voxels, 20 voxels shorter than in b), but calculated with a step size that give a higher precision. The difference of c) and d) is the distance between the seeded particles. In c) the particles are seeded 10 voxels apart in the x and y directions. In d) this distance is 1 voxel. The higher precision settings in d) resulted in no false positives, but many false negatives. c) contains both false negatives and false positives, but had an overall better outcome than d).

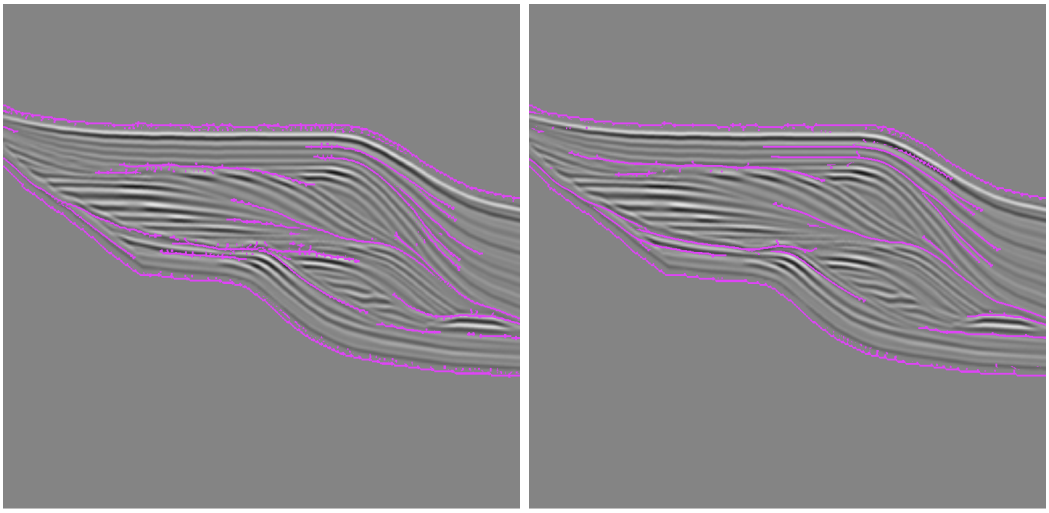


Fig. 5.13: a) The result from using inaccurate trajectory calculations with a seeding radius of 0.1. b) The best result found by trial and error.

In Figure 5.13 a) we used more inaccurate parameter settings for the trajectory calculations. We used few (10) trajectory steps and a large step size of 5. For the seeding radius we used an accurate setting of 0.1. These settings gave the least false negatives, but also many false positives. A trial and error approach revealed that the best parameter settings for this data set were in the midrange of the presented settings. The result can be seen in Figure 5.13 b). A setting of 175 for the number of trajectory steps, a steps size of 0.5 and a seeding radius of 1 was used.

5.3 3D test

To test our pipeline on 3D data, we stacked copies of the test images into volumes. Our pipeline processes all slices or a subset of slices at each processing stage. In the previous sections we have only looked at output slices. Figure 5.14 shows a volume that has been processed by our pipeline and visualized by raycasting in VolumeShop. A slice of the original data is blended with the visualized selection volume. We have used a seismic cross section image and stacked it into a volume of size $512 \times 512 \times 64$. This volume was processed by our unconformity detection pipeline, where the output

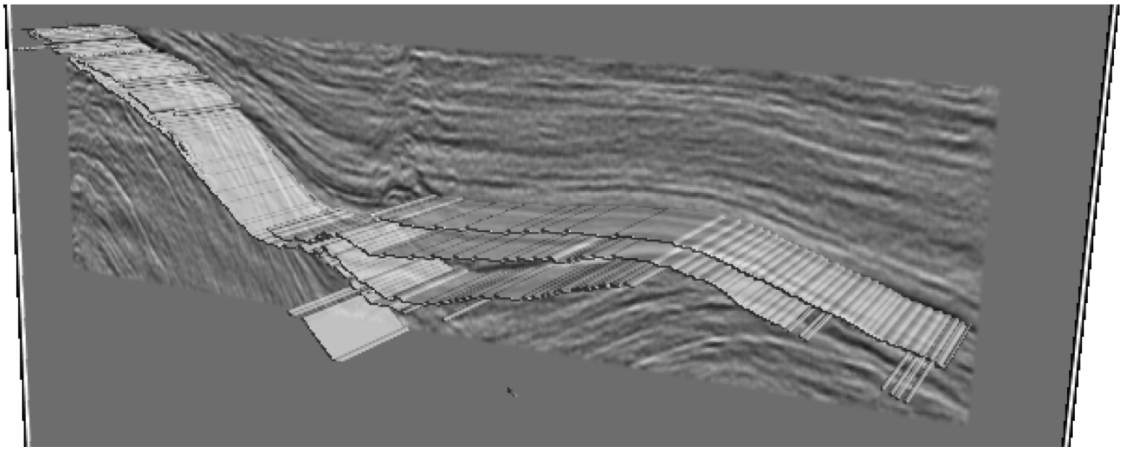


Fig. 5.14: A selection volume output from our pipeline visualized in VolumeShop. A slice of the input data is blended with the output volume.

is a selection volume containing the detected unconformities. The input volume does not have any variation in z direction, so neither do the detected surfaces. We do not have a qualified interpretation of this seismic cross section, and can therefore not comment on whether the detected unconformities are rightfully detected or not.

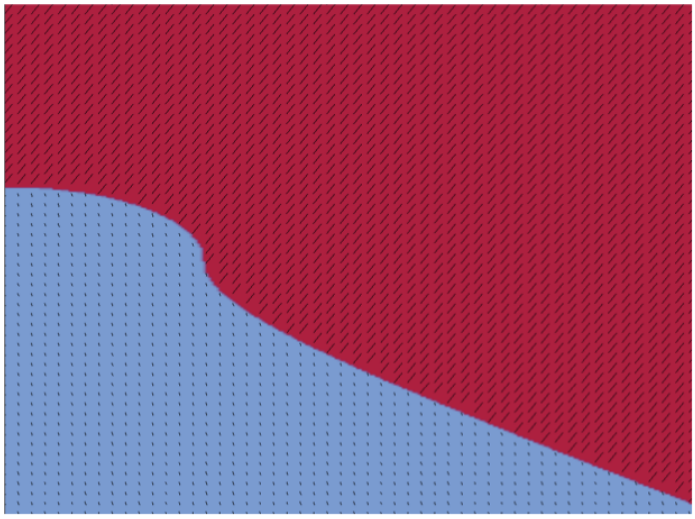


Fig. 5.15: A slice of our 3D vector field test set.

The final test set presented in this thesis, is a procedurally created 3D vector field that varies in z direction. This test volume contains two sections of different vectors. Figure 5.15 show a slice of this volume with added black lines to indicate the underlying vectors. The vectors in the blue area have a greater z-component than the vectors in the red area. The size of this data set is 256^3. It was made to test the

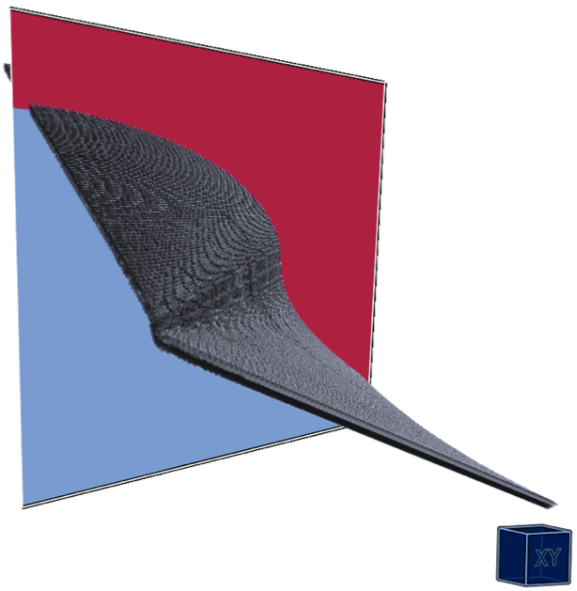


Fig. 5.16: A detected surface visualized in VolumeShop.

implementation of our unconformity algorithm that work on the data in 3D. However, the 2D implementation of the unconformity detection algorithm gave the same result as the 3D implementation on this test set. The surface was successfully detected by our 3D unconformity detection method (see Figure 5.16). We have visualized the resulting selection volume in VolumeShop blended with one of the input slices.

5.4 Performance

The tests of this chapter were done on a machine with an $NVIDIA\ GeForce\ GTX$ 295 graphics card with 896 MB texture memory and an $Intel^R\ Core^{TM}\ 2\ Duo$ processor with 2GB ram. The below table gives an overview of the running times of processing the data sets by our unconformity detection pipeline. The data is processed as described in Section 5.2. We have included running times for the optimized and the unoptimized method for data sets of two different sizes.

Slice Size	256×256	512×512
Trajectory Steps (Step Size 0.1)	1000	1000
Seconds pr Slice $\left(\frac{sec}{s}\right)$	1.59	2.94
Optimized Method $\left(\frac{sec}{s}\right)$ Both Flow Directions	0.28	0.58

Chapter 6

Summary and Conclusions

This thesis has presented two results. One general pipeline for processing seismic data with all calculations done on the GPU, and algorithms on top of the pipline for detecting unconformities in seismic data. The unconformity detection implementation is a specialized form of the general pipeline. Both results are described in Chapter 3. The major algorithmic focus has been to develop a new unconformity detection method that address the data on a global scale. A global approach will facilitate unconformity detection where local methods fail, namely in neighborhoods of parallel horizons. It can lighten the workload of interpreting seismic data and aid the search for stratigraphic hydrocarbon traps.

We have implemented the general pipeline and used it as a base for our implementation of the unconformity detection method. The implementation details can be found in Chapter 4. The pipelines are implemented as a set of plugins for the VolumeShop framework. Two 3D textures located on the GPU are utilized for reading and writing, and the data is moved back an forth between these textures while filtered. Most of the data processing is done by our general filter-plugin. The specific filters are implemented as shaders within this plugin. This program architecture allows tweaking of filter parameters in real-time.

Our unconformity detection method first extracts a flow field representation of the data. This flow field is smoothed according to its structures. We have implemented image processing methods for the extraction and smoothing. Gradients are calculated by a central difference or a Sobel mask, and rotated to tangent the horizons. We have found it necessary to smooth the vector field by a mean filter before the structure

oriented smoothing is done. The reason is that we get zero gradients in some voxels and the structure oriented smoothing filter will not smooth across these zero vectors. Particles are seeded in the flow field, and their trajectories calculated. To find a scalar value that characterizes the amount of stretching around a trajectory, FTLE is calculated from the final positions of close trajectories. Finally, the height ridges of the FTLE field is extracted. These height ridges can be considered as borders of LCS, and they coincide with unconformities in seismic data that has been processed by our flow extraction procedure.

We have made two different implementations of the unconformity detection algorithm. One where the trajectory calculations and FTLE calculations are done in the same shader, and a more optimized implementation that divides these operations into two different shaders. The optimization is based on calculating the trajectories only once for every voxel, opposed to calculating a set of trajectories for every voxel. The disadvantage with the faster method is that the processing is done in only one flow direction at the time. Hence, the user will have to inspect two different volumes to see all the detected unconformities. Although, one can easily add an extra pass for combining these two volumes. The slower method outputs the detected unconformities from both flow directions in the same volume.

The implementation of our unconformity technique has shown a promising outcome in several tests. We successfully detected the unconformity throughout a parallel area when the algorithm was run globally. In a different synthetic test set, the unconformity was not detected in the parallel area. The reason was that the data set contained horizons that were parallel in one area and converging in the next. The unconformity was only detected in the non-parallel area of this data set. In yet another test set, containing bell-shaped horizons, the unconformity was rightfully detected. Here the seeded particles moved through an area of converging horizons and into an area of diverging horizons.

When we tested our algorithm on a volume created from a seismic cross section image, the unconformities were detected for the most part. We also got some false positives and some false negatives. The false positives were seen in an area where the horizons were not parallel although they belonged to the same sedimentary unit. The false negatives occurred were closely seeded particles would not separate at any time.

Our algorithm depends on user-defined settings for these parameters: number of trajectory steps, step size, and the seeding radius of a particle set. In Section 5.2.1 we showed that different parameter settings gave quite different results. The number of steps together with the step size decides to what degree the data is address locally or globally. The step size together with the seeding radius defines the accuracy of the calculations. In one test set we saw that too much accuracy gave false negatives while not enough accuracy resulted in false positives. At this stage, we can not say anything definite about good or bad parameter settings. We can only conclude that what is good parameter settings depends on the data itself, and is best found by trial and error. Which is well supported in our pipeline that enables interactive parameter adjustment. Many parameters is not special to our algorithm, many other methods also require the setting of several parameters [201].

6.1 Future Work

Large Data

For processing actual seismic data volumes, a few improvements to the pipeline are needed. Seismic data sets are often very large in size, and it would not be possible to load such volumes in their entirety onto the GPU. A method for processing large data sets in sequence, such as stream-processing, is necessary.

True 3D Feature Extraction

We extract a 2D flow field in our pipeline, and by that we lose the information inherent in the $z-$components of the horizon gradients. For our test sets, this has not been an issue, but with real seismic data, a gradient reduction from 3D to 2D may remove relevant information. The algorithm could be improved by processing the data twice in orthogonal directions. Hence, the information that is lost in one processing direction is picked up in the other.

Alternatively, we could extend the 2D trajectories into 3D surfaces and use a surface difference measure for finding the degree of separation.

Better Segmentation

A more advanced height ridge extraction algorithm than the one we have implemented could improve our unconformity detection method. For example could a watershed algorithm be used. A better algorithm for this processing step could detect more height ridges in the FTLE field, and thereby detect more unconformities.

Optimization

Optimization have not been our focus in this thesis, however, different optimization measures could be implemented to reduce processing times. The filtered AMR ridge extraction method [SP07] mentioned in Section2.3.1 is only one example that would result in a speed-up of our unconformity detection pipeline.

Acknowledgments

This work has been carried out within the Geoillustrator research project (# 200512), which is funded by Statoil and the PETROMAKS programme of the Research Council of Norway.

I would like to thank my supervisor Daniel Patel for his inspiration, constant support and never-failing patience. Thanks to Armin Pobitzer for clarifying the concept of FTLE, and to Helwig Hauser and the rest of the visualization group at the University of Bergen for everything they have taught me.

Bibliography

[200] OpenWalnut Community 2009. BSV-Leipzig and CNCF-CBS. GLSL code for eigenanalysis by Mario Hlawitschka 2007, GNULGPL. http://www.openwalnut.org Accessed: May 2012. 46

[201] Petrel E&P Software Platform 2012. Software package for oil and gas exploration and production developed by Schlumberger Limited. http://www.slb.com/petrel.aspx Accessed: May 2012. 11, 12, 24, 67

[AT04] F. Admasu and K. Toennies. Automatic method for correlating horizons across faults in 3d seismic data. In *Computer Vision and Pattern Recognition, 2004 IEEE*, Washington, DC, USA, July 2004. 25

[Bar00] A. E. Barnes. Weighted average seismic attributes. *Geophysics*, 65(1):275–285, January 2000. 15

[BC99] M. Brown and R. G. Clapp. Seismic pattern recognition via predictive signal/noise separation. *Stanford Exploration Project, Report 102*, pages 177–187, 1999. 13

[BF95] M. Bahorich and S. Farmer. The coherence cube. *The Leading Edge*, 14(October):1053–1058, 1995. 13

[BG05] S. Bruckner and M. E. Gröller. Volumeshop: An interactive system for direct volume illustration. *IEEE Visualization '05*, pages 671–678, 2005. 23, 32, 33

[BGS00] R. Bloem, H. N. Gabow, and F. Somenzi. An algorithm for strongly connected component analysis in n log n symbolic steps. In *Formal Methods in Computer-aided Design*, pages 56–73. Springer, 2000. 31

[BvVV99] P. Bakker, L. J. van Vliet, and P. W. Verbeek. Edge preserving orientation adaptive filtering. In *Conference on Computer Vision and Pattern Recognition, IEEE*, June 1999. 15

[Cat06] O. Catuneanu. *Principles of Sequence Stratigraphy*. Elsevier, 2006. 7

[CCvB+97] J. Caldwell, A. Chowdhury, P. van Bemmel, F. Engelmark, L. Sonneland, and N. S. Neidell. Exploring for stratigraphic traps. *Oilfield Review*, 9(4):48–61, Winter 1997. 6

[Cho02] S. Chopra. Coherence cube and beyond. *First Break*, 20(1):27–33, 2002. 14

[CM05] S. Chopra and K. J. Marfurt. Seismic attributes - a historical prespective. *Geophysics*, 70(5):3–28, September 2005. 8, 12

[CM07] S. Chopra and K. Marfurt. Volumetric curvature attributes add value to 3d seismic data interpretation. *The Leading Edge*, 26(7):856–867, July 2007. 24

[CM08] S. Chopra and K. J. Marfurt. Emerging and future trends in seismic attributes. *SEG, The Leading Edge*, 27(3):298–318, March 2008. 12

[CM11] S. Chopra and K. J. Marfurt. Volume co-rendering of seismic attributes - a great aid to seismic interpretation. In *2011 SEG Annual Meeting*, San Antonio, Texas, USA, September 2011. 12

[GGTH07] C. Garth, F. Gerhardt, X. Tricoche, and H. Hagen. Efficient computation and visualization of coherent structures in fluid flow applica-

tions. *Transactions on Visualization and Computer Graphics, IEEE*, 13(6):1464–1471, November 2007. 18, 19

[GLT+09] C. Garth, Guo-Shi Li, X. Tricoche, C. D. Hansen, and H. Hagen. Visualization of coherent structures in transient 2d flows. In *Topology-Based Methods in Visualization II*, pages 1–13. Springer, 2009. 18, 19

[Hal01] G. Haller. Distinguished material surfaces and coherent structures in three-dimensional fluid flows. *Physica D*, 149(4):248–277, March 2001. 17, 18

[Hal02] G. Haller. Lagrangian coherent structures from approximate velocity data. *Physics of Fluids*, 14(6):1851, May 2002. 17, 18, 47, 59

[HF02] C. Höcker and G. Fehmers. Fast structural interpretation with structure-oriented filtering. *SEG, The Leading Edge*, 21(3):238–243, July 2002. 11, 25

[HH89] J. Helman and L. Hesselink. Representation and display of vector field topology in fluid flow data sets. *Computer*, 22(8):27–36, August 1989. 17

[HS11] G. Haller and T. Sapsis. Lagrangian coherent structures and the smallest finite-time lyapunov exponent. *Chaos*, 21(2):023115, May 2011. 19

[JST81] A. Jameson, W. Schmidt, and E. Turkel. Numerical solution of the euler equations by finite volume methods using runge-kutta time stepping schemes. In *AIAA 14th Fluid Dynamics and Plasma Dynamics Conference*, Paolo Alto, California, USA, June 1981. 43

[Ker03] H. Kermit. *Niels Stensen, 1638-1686: The Scientist who was Beatified*. Gracewing Publishing, 2003. 3

[KLC09] H. Kang, S. Lee, and C. K. Chui. Flow-based image abstraction. *Transactions on Visualization and Computer Graphics, IEEE*, 15(1):62–76, 2009. 16, 25, 26, 40, 41

[KW85] M. Kass and A. P. Witkin. Analyzing oriented patterns. *In Proc. Ninth IJCAI*, pages 944–952, 1985. 15

[LM10] D. Lipinski and K. Mohseni. A ridge tracking algorithm and error estimate for efficient computation of lagrangian coherent structures. *Chaos*, 20(1):017504, January 2010. 18

[LMADA02] Y. Luo, M. Marhoon, S. Al Dossary, and M. Alfaraj. Edge-preserving smoothing and applications. *The Leading Edge*, 21(2):136–141, February 2002. 24

[LPG+07] O. Lavialle, S. Pop, C. Germain, M. Donias, S. Guillon, N. Keskes, and Y. Berthourmieu. Seismic fault preserving diffusion. *Journal of Applied Geophysics*, 61(2):132–141, February 2007. 24

[Mar06] K. J. Marfurt. Robust estimates of 3d reflector dip and azimuth. *Geophysics*, 71(4):P29–P40, 2006. 11, 15

[MKF98] K. J. Marfurt, R. L. Kirlin, and S. L. Farmer. 3-d seismic attributes using a semblance-based coherency algorithm. *Geophysics*, 63(4):1150–1165, July 1998. 15

[MM00] W. Y. Ma and B. S. Manjunath. Edgeflow: A technique for boundary detection and image segmentation. *Transactions on Image Processing, IEEE*, 9(8):1375–88, January 2000. 16

[MTHC03] I. Mikić, M. Trivedi, E. Hunter, and P. Cosman. Human body model acquisition and tracking using voxel data. *International Journal of Computer Vision*, 53(3):199–223, 2003. 31

[Nic09] G. Nichols. *Sedimentology and Stratigraphy*. John Wiley and Sons, 2009. 7

[Pat09] D. Patel. Knowledge-assisted visualization of seismic data. *Computers & Graphics*, 33(5):585–596, October 2009. 12

[Pat10] D. Patel. Seismic volume visualization for horizon extraction. In *Pacific Visualization Symposium (PacificVis), 2010 IEEE*, Taipei, Taiwan, March 2010. 12

[PD10] T. Peacock and J. Dabiri. Introduction to focus issue: Lagrangian coherent structures. *Chaos*, 20(1):017501, January 2010. 18

[PPF+10] A. Pobitzer, R. Peikert, R. Fuchs, B. Schindler, A. Kuhn, H. Theisel, K. Matković, and H. Hauser. On the way towards topology-based visualization of unsteady flow - the state of the art. In *Eurographics 2010*, Norrköping, Sweden, May 2010. 18

[RPS01] T. Randen, S. I. Pedersen, and L. Sonneland. Automatic extraction of fault surfaces from three-dimensional seismic data. In *SEG Annual Meeting, 2001*, San Antonio, Texas, USA, September 2001. 25

[RRSS98] T. Randen, B. Reymond, H. I. Sjulstad, and L. Sonneland. New seismic attributes for automated stratigraphic facies boundary detection. 17(3):628–631, 1998. 12

[RS91] A. R. Rao and B. G. Schunck. Computing oriented texture fields. *CVGIP: Graphical Models and Image Processing*, 53:157–185, 1991. 15

[SLM] S. C. Shadden, F. Lekien, and J. E. Marsden. Definition and properties of lagrangian coherent structures from finite-time lyapunov exponents in two-dimensional aperiodic flows. *Physica D*, 212(3). 18

[SP07] F. Sadlo and R. Peikert. Efficient visualization of lagrangian coherent structures by filtered amr ridge extraction. *Transactions on Visualization and Computer Graphics, IEEE*, 13(5):1456–1463, November 2007. 19, 68

[SP09] S. Sadlo and R. Peikert. Visualizing lagrangian coherent structures and comparison to vector field topology. In *Topology-Based Methods in Visualization II*, pages 15–29. Springer, 2009. 17

[SRP11] F. Sadlo, A. Rigazzi, and R. Peikert. Time-dependent visualization of lagrangian coherent structures by grid advection. In *Topology-Based Methods in Data Analysis and Visualization*, pages 151–165. Springer, 2011. 18

[Tan01] M. T. Taner. Seismic attributes. *Canadian Society of Exploration Geophysicists Recorder*, 26(9):48–56, 2001. 8

[vHGP10] T. van Hoek, S. Gesbert, and J. Pickens. Geometric attributes for seismic stratigraphy interpretation. *SEG, The Leading Edge*, 29(9):1056–1065, 2010. 14, 49, 59

Appendix

Algorithm 1 SOF for smoothing the vector field

Input: the current position
Output: the smoothed tangent, the initial gradient magnitude

$tangent \leftarrow flowSample(currentPosition)$
for $1 \rightarrow maskSizeX$ **do**
  for $1 \rightarrow maskSizeY$ **do**
    $tmpPosition \leftarrow nextPositonInMask$
    $tmpTangent \leftarrow flowSample(tmpPosition)$
    if $tangent \cdot tmpTangent \geq 0$ **then**
      $\phi \leftarrow 1$
    else
      $\phi \leftarrow -1$
    end if
    $w_m \leftarrow \frac{gradientMagnitude(currentPosition) - gradientMagnitude(tmpPosition) + 1}{2}$
    $w_d = |normalize(tangent) \cdot normalize(tmpTangent)|$
    $smoothedVec \leftarrow smoothedVec + \phi \times tmpTangent \times w_m \times w_d$
    $k \leftarrow k + 1$
  end for
end for
$currentVoxel.RG \leftarrow smoothedVec/k$
$currentVoxel.B \leftarrow tangent.B$ ▷ The initial gradient magnitude

Algorithm 2 Get End of Path

Input: the start position, a flow direction (either -1 or 1)
Output: the end position of a trajectory, number of steps in the trajectory

$s \leftarrow stepSize \times direction$
while $i < maxSteps \, and \, update = true$ **do**
  $tmpPosition \leftarrow RK4(currentPosition, s)$
  if $tmpPosition = currentPosition$ **then**
    $update \leftarrow false$
  else
    $currentPostion \leftarrow tmpPosition$
  end if
  $i \leftarrow i + 1$
end while
$currentVoxel.RG \leftarrow currentPosition$
$currentVoxel.B \leftarrow i$ ▷ The performed number of trajectory steps

Algorithm 3 Calculate FTLE

Input: the current position
Output: the maximum FTLE value

$endPositionsArray\,[8] \leftarrow InitPathEndArray(currentPosition)$
for $i = 1 \rightarrow 2$ **do** ▷ For both directions of the flow
  $\mathbf{J} \leftarrow calculate \, the \, Jacobian \, matrix \, from \, the \, endPositions$
  $\mathbf{C} \leftarrow \mathbf{J^T J}$
  $\lambda \leftarrow majorEigenvalue(\mathbf{C})$
  $FTLEtmp\,[i] \leftarrow \frac{1}{2n} \ln \lambda$
end for
$FTLE \leftarrow max(FTLEtmp\,[1], FTLEtmp\,[2])$
$currentVoxel.R = FTLE$

Algorithm 4 Height Ridges

Input: the current position
Output: the current voxel as selected or not selected

$heightRidge \leftarrow false$
$sample1 \leftarrow sample(currentPosition + dy)$
$sample2 \leftarrow sample(currentPosition - dy)$
if $FTLE(sample1)\ and\ FTLE(sample2) < FTLE(currentPosition)$ **then**
    $heightRidge \leftarrow true$
end if
if $heigthRidge = false$ **then**
    $sample1 \leftarrow sample(currentPosition + dx)$
    $sample2 \leftarrow sample(currentPosition - dx)$
    if $FTLE(sample1)\ and\ FTLE(sample2) < FTLE(currentPosition)$ **then**
        $heightRidge \leftarrow true$
    end if
end if
if $heightRidge = true$ **then**
    $currentVoxel = selected$
else
    $currentVoxel = notSelected$
end if