UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

SPRING MESH HEURISTIC AND BARYCENTRIC DEFORMATION ALGORITHM FOR GEOLOGICAL SOLID OBJECT UNFOLDING

TESIS PARA OPTAR AL GRADO DE MAGÍSTER EN CIENCIAS, MENCION COMPUTACIÓN

CAMILO JOSE ROJAS MILLA

PROFESOR GUÍA:
NANCY HITSCHFELD KAHLER

MIEMBROS DE LA COMISIÓN:
FELIPE NAVARRO VARGAS
JOSE PIQUER GARDNER
JOSE SAAVEDRA RONDO
GUILLERMO CABRERA GUERRERO

Trabajo parcialmente financiado por CONICYT/PIA proyecto AFB180004 y Conicyt-Fondef proyecto ID16I10186

SANTIAGO DE CHILE
2020

Resumen

En la evaluación económica de recursos minerales, herramientas geoestadísticas y algoritmos geométricos son aplicados a modelos geológicos y datos cuantitativos para estimar el volumen y valor del mineral en un yacimiento. La presencia de fallas y pliegues en el modelo geológico, tiene un impacto negativo en la estimación de minerales, ya que agregan errores a los modelos estadísticos. Se utilizan técnicas de desenrollado (unfolding), basadas en algoritmos geométricos y métricas de calidad, para mitigar los errores de los pliegues y obtener estimaciones más precisas. Habitualmente, en la industria minera y restauración geológica, se utiliza un enfoque de aplanamiento de superficie basado en una superficie referencial para obtener el volumen desenrollado. El presente trabajo de investigación desarrolla un nuevo algoritmo de desenrollado (unfolding), basado en el aplanamiento de una malla de resortes de referencia y transformaciones baricéntricas. El algoritmo diseñado permite obtener un objeto sólido sin pliegues que es comparado en calidad y rendimiento con el resultado del software especializado U-Fo ” [1]. Los resultados obtenidos muestran que el nuevo algoritmo baricéntrico supera en métricas de calidad a “ U-Fo ” en dos casos simulados, mientras que tiene un rendimiento ligeramente peor en el caso real utilizado.

[1] Software creado por el laboratorio ALGES para desenrrollar modelos geológicos

Abstract

In mineral "Resource Assessment" (RA), geo-statistics tools and geometrical algorithms are applied to geological models and quantitative data for ore [2] estimation. The presence of faults and folded surfaces have a negative impact on mineral estimation as they add error to statistical models. Unfolding techniques using geometrical algorithms and quality measurements of the unfolded object are essential to obtain more accurate estimations. Historically, in the mining industry and geological restoration, a surface flattening approach based on a referential surface is being used to obtain an unfolded geometry. This Thesis presents a new unfolding algorithm based in an energy conservation spring mesh surface flattening and barycentric transformations to perform solid object unfolding. The algorithm is also compared in quality and performance against "U-Fo" [3] software. Quality results show that the new barycentric algorithm outperforms "U-Fo" in two simulation cases while performing slightly worse in one field case, without substantially increased execution time.

[2] An ore is an occurrence of rock or sediment that contains sufficient minerals with economically important elements.

[3] Software created by ALGES laboratory for unfolding geology models.

"*Thanks to my family, friends and Belen"*

Camilo Jose Rojas Milla

Acknowledgements

Agradezco a mi familia (padres y hermanos) por todo el apoyo brindado durante el tiempo que tomó realizar esta investigación.

Agradezco, también, a toda la comisión de Tesis, particularmente a la Dra Nancy Hitschfeld, por su excelente labor guiando este trabajo de investigación; asimismo a Felipe Navarro por su apoyo constante y conocimiento experto sobre geología y minería. Adicionalmente, gracias a los profesores Jose Miguel Piquer, Jose Manuel Saavedra y Guillermo Cabrera Guerrero cuyas acertadas correcciones mejoraron significativamente este trabajo.

Muchas gracias al laboratorio ALGES y a Conicyt-Fondef, por el financiamiento y apoyo técnico a todo el proceso de Magíster. En particular muchas gracias a Marcia Ojeda por la ayuda brindada en la simulacion de los objetos geologicos de prueba.

Gracias a todos mis amigos por ayudarme a soportar el estrés en espacios dentro y fuera de lo académico.

Por último y no por eso menos importante, infinitas gracias a mi compañera incondicional Belén González, por todo el amor que me brinda, por enfrentar este proceso juntos y por su eterna paciencia.

Contents

1 Introduction **1**
- 1.1 Problem Statement . . . 1
- 1.2 Investigation Questions . . . 2
- 1.3 Hypothesis . . . 2
- 1.4 Objectives . . . 3
  - 1.4.1 General Objective . . . 3
  - 1.4.2 Specific Objectives . . . 3
- 1.5 Contributions . . . 3

2 Conceptual Framework **4**
- 2.1 Fundamental Concepts . . . 4
  - 2.1.1 Resource Assessment Estimation . . . 4
  - 2.1.2 Geometric Algorithms . . . 5
    - Meshes . . . 5
    - Delaunay Triangulation . . . 6
    - Convex Hull . . . 6
  - 2.1.3 Statistic Estimators . . . 7
  - 2.1.4 Heuristic Algorithms . . . 7
- 2.2 State of Art . . . 7
  - 2.2.1 Geological Restoration . . . 7
  - 2.2.2 Tetrahedral Mesh . . . 8
  - 2.2.3 Surface Extraction . . . 9
  - 2.2.4 Surface Flattening . . . 10
  - 2.2.5 Barycentric Deformation . . . 12
  - 2.2.6 Quality Measures . . . 13
- 2.3 Software Tools . . . 14
  - 2.3.1 CGal . . . 14
  - 2.3.2 UFo . . . 15

3 Proposed Solution **16**
- 3.1 Novel Approach . . . 16
- 3.2 Algorithm Description . . . 17
  - 3.2.1 Pre-process . . . 17
  - 3.2.2 Surface Extraction . . . 19

3.2.3 Surface Flattening 20
3.2.4 Mesh Construction 22
3.2.5 Barycentric Deformation 26
3.2.6 Algorithm Complexity 28
3.2.7 Quality Measures 29
3.3 Implementation Details 33

4 Results **36**
4.1 Performance Comparison 39
4.2 Quality Comparison 41
4.2.1 Quality Results for simulation case 1 42
4.2.2 Quality Results for simulation case 2 47
4.2.3 Quality Results for field case 1 52
4.2.4 Quality Analysis 55

5 Conclusions **56**
5.1 Conclusions 56
5.2 Future Work 57

Bibliography **59**

Appendix **63**

A Source Code **63**
A.1 Python Source Code 63
A.2 C++ Source Code for CGAL 78

List of Figures

2.1 Resource assessment pipeline for ore estimation, in which remove folds and faults are depicted in steps 2 and 3 [8]. In order to test unfolding algorithms. New steps can be added in parallel to use simulated models instead of field cases. 5
2.2 Delaunay Triangulation over 2D random points in a square. 6
2.3 At left a point cloud dataset of 6510 samples. At right the tetrahedral mesh calculated with "Delaunay Triangulation" with a result of 38376 tetrahedrons. . 9
2.4 Alpha-shape representation for 2D dataset. Image downloaded from CGal user manual. 10
2.5 Spring mesh energy system. P_i represents vertices in projected surface into a plane 12
2.6 At left a cube of length one divided into tetrahedron. At right a deformed cube with a vertex moved to coordinate $(1.5, 1.5, 0)$. The points inside the cube are translated with barycentric transformation using initial and final tetrahedral coordinates. 13
2.7 NMD index depicted over a parallelepiped 14

3.1 Algorithm input and output diagram flow. Both input and output are matrices of shape $(Nx3)$. 17
3.2 Pre-process diagram 18
3.3 Figure at top shows raw dataset as given from input file, and second figure is the dataset scaled between $[0 \dots 1]$ range. 18
3.4 Point cloud rotated by its principal component vectors. In yellow the original orientation is depicted, while the rotated dataset is shown in purple color. The principal component vectors x_p and y_p are rendered at the centroid. 19
3.5 At left X matrix representing solid geological body. At right *Concave Hull* extracted from X data. 20
3.6 Top and bottom surfaces 20
3.7 Flattened top and bottom surfaces 21
3.8 At left surface unfolding energy algorithm evolution trough 12 iterations; Left axis shows "Mean Square Error" between folded surface and unfolded and right axis shows changes in total area. At right error between folded surface and unfolded one is plotted in each triangle. 22
3.9 Projection of a vertex from bottom surface into top surface. The point projected in top surface is then inserted Delaunay into the triangulation 23
3.10 Prism created by joining top and bottom triangle. 24

3.11 Prism divided into three tetrahedrons. In purple T_1, green T_2 and yellow T_3 . . 25
3.12 Tetrahedral mesh wire-frame view 25
3.13 Tetrahedral mesh with opacity 26
3.14 Unfolded point cloud colored with neighbor distance error against folded point cloud. 30
3.15 X and Y point clouds, colored with their neighbor quality measure ($NMD(\cdot)$) . 31
3.16 Histogram and Cumulative error distribution from a point cloud unfolding process. From a total of $21,600$ points, more than the 90% have an error less than 0.1 32

4.1 Computation flow for error quality index (Equation 2.4) in **simulated** and **field data**. Defining W as the original block model distribution, X as the folded model and Y as the unfolded model, we obtained two error distributions: $\epsilon(W, Y)$ and $\epsilon(X, Y)$ for simulated data and just one error distribution: $\epsilon(X, Y)$ for field data. . 37
4.2 Normalized point cloud datasets, two simulated (A and B) and one field case (C). In each figure the color represents the value of NMD (Equation 2.2) index per point. 38
4.3 Performance comparison of "surface flattening" and "3D unfolding" steps for Barycentric and U-Fo unfolding algorithms. The height in the bar plot represents the mean time in each step, and the standard deviation is visualized as an error line above the mean value. 41
4.4 Point clouds depicting NMD($\cdot$) index in each step of unfolding flow for simulation case 1. Index is calculated using Equation 2.2. We define W as the original block model distribution, X as the folded model and Y as the unfolded model. . 43
4.5 $\epsilon(X, Y)$ and $\epsilon(W, Y)$ distribution comparison between barycentric and U-Fo unfolding algorithms, using violin plots for simulation case 1. Error distributions are calculated using Equation 2.4. We define W as the original block model distribution, X as the folded model and Y as the unfolded model, we obtained two error distributions: $\epsilon(W, Y)$ and $\epsilon(X, Y)$ for simulated. 44
4.6 $\epsilon(X, Y)$ and $\epsilon(W, Y)$ distribution comparison between barycentric and U-Fo unfolding algorithms, using histogram plots for simulation case 1. Error distributions are calculated using Equation 2.4. We define W as the original block model distribution, X as the folded model and Y as the unfolded model, we obtained two error distributions: $\epsilon(W, Y)$ and $\epsilon(X, Y)$ for simulated data. 45
4.7 $\epsilon(X, Y)$ and $\epsilon(W, Y)$ distribution comparison between barycentric and U-Fo unfolding algorithms, using histogram and cumulative plots for simulation case 1. Error distributions are calculated using Equation 2.4. We define W as the original block model distribution, X as the folded model and Y as the unfolded model, we obtained two error distributions: $\epsilon(W, Y)$ and $\epsilon(X, Y)$ for simulated data. 46

4.8 Point clouds depicting NMD(·) index in each step of unfolding flow for simulation case 2. Index is calculated using Equation 2.2. We define W as the original block model distribution, X as the folded model and Y as the unfolded model. . 48
4.9 $\epsilon(X, Y)$ and $\epsilon(W, Y)$ distributions comparison, using violin plots for simulation case 2. Error distributions are calculated using Equation 2.4. We define W as the original block model distribution, X as the folded model and Y as the unfolded model, we obtained two error distributions: $\epsilon(W, Y)$ and $\epsilon(X, Y)$ for simulated data. 49
4.10 $\epsilon(X, Y)$ and $\epsilon(W, Y)$ distributions comparison, using histogram plots for simulation case 2. Error distributions are calculated using Equation 2.4. We define W as the original block model distribution, X as the folded model and Y as the unfolded model, we obtained two error distributions: $\epsilon(W, Y)$ and $\epsilon(X, Y)$ for simulated data. 50
4.11 $\epsilon(X, Y)$ and $\epsilon(W, Y)$ distributions comparison using cumulative plots for simulation case 2. Error distributions are calculated using Equation 2.4. We define W as the original block model distribution, X as the folded model and Y as the unfolded model, we obtained two error distributions: $\epsilon(W, Y)$ and $\epsilon(X, Y)$ for simulated data. 51
4.12 Point clouds depicting NMD(·) index in each step of unfolding flow for field case 1. Index is calculated using Equation 2.2. We define W as the original block model distribution, X as the folded model and Y as the unfolded model. . 53
4.13 $\epsilon(X, Y)$ distribution comparison between barycentric and UFo unfolding algorithms, using violin, histogram and cumulative plots for field case 1. Error distribution is calculated using Equation 2.4. We define W as the original block model distribution, X as the folded model and Y as the unfolded model, we obtained one error distribution: $\epsilon(X, Y)$ for field data. 54

List of Tables

4.1 Number of points per dataset. 37
4.2 Execution times for barycentric unfolding algorithm. Columns represents the algorithm steps, and for each, we calculate the mean time and standard deviation of 10 executions. 40
4.3 Execution times for UFo unfolding algorithm.Columns represents the algorithm steps, and for each, we calculate the mean time and standard deviation of 10 executions. The steps that can be compare with barycentric algotihm are “3d unfolding” and “surface flattening”. 40
4.4 Mean error comparison for each geological model. We define W as the original model distribution, X as the unfolded model and Y as the folded model. The Error index distribution ($\epsilon(\cdot,\cdot)$) is calculated using Equation 2.4 and the function $\mu(\cdot)$ symbolize the statistical mean. 55

Chapter 1

Introduction

In the mining industry, resource assessment is directly related to the Return of Investments (ROI) of mining projects. Nevertheless, resource assessment estimation is an uncertain process, associated with complex geological features like faults and folds, hardening the extraction of useful information from drill holes [8][26]. In recent years, geological restoration techniques based on a priory geology information have been developed (Maerten L. & Maerten F. 2006 [16], Bennis et al. 2013 [2], Horna et al. 2010 [11], Tang et al. 2016 [25]). Another approach to the restoration process is to use only the geometric information available in the solid body obtained from the mining block model and to validate the result with statistical techniques. This approach does not rely on any other data than the geometric topology of the ore body.

Restoration using geometrical information has been used as a novel approach by the "Advanced Laboratory for Geostatistical Supercomputing"(ALGES) in the U-Fo project [20]. In their approach, a referential surface of the ore body is flattened using a spring energy conservation model [19]. Then, all the points representing the ore body are rearranged based on the flattened surface. The result is a restored body where the statistical distribution of some features, in the initial surface, are preserved in the flattened surface. Preserving geometric statistics in the restored body assures better spatial conservation, which directly relates to mineral estimation [26].

This research presents a spring mesh energy conservation model for 3D solid body. The proposed algorithm solves the unfolding problem preserving the 3D geometric topology between the initial body and the restored one.

1.1 Problem Statement

Geological restoration manages to get an unfolded and unfaulted solid body from the geology model constructed in mine planning and resource assessment workflow. The current state of the art restoration techniques estimates physical properties of the ore body to improve their restoration model [11] [25]. Those approaches obtain good results but are only useful when

the dataset is rich enough to make assumptions and estimations. For automation purposes, a restoration algorithm using only geometric properties of the dataset would be beneficial as a recommendation guide to geological restoration experts.

We propose to obtain an unfolded (developable) 3D solid body from a point cloud dataset, representing a folded object. The algorithm achieved should preserve some geometry features information between the input and output body. Geometric statistics such as: volumes between tetrahedrons, area between triangulation faces and length between connection edges can be used to test quality in unfold distributions.

As related work, the ALGES group presented in their software U-Fo [20] a methodology to unfold a 3D solid object represented by a mining block model. In their software, they used an energy conservation model to unfold a known 2D surface and attach the centroids of all blocks to their respective cell in the unfolded surface. We propose to enlarge this "2D developable surface algorithm" to a 3D extension that should unfold the entire body, maintaining its most important geometric features.

1.2 Investigation Questions

- Is the new geometric unfolding algorithm better suited to geological unfolding compared with its 2D unfolding version like the one used in U-Fo? How are they compared in terms of conserving geometrical statistics such as volumes between tetrahedrons, area between triangulation faces and length between connection edges?
- What is the processing and memory cost of the new algorithm? How well does it perform in comparison to the one used previously?

1.3 Hypothesis

A geometrical unfolding algorithm based on mesh energy conservation and barycentric transformations will preserve edge distances distribution of triangulated mesh between simulated and unfolded objects.

1.4 Objectives

1.4.1 General Objective

Develop a new geometrical unfolding algorithm based on mesh energy conservation and barycentric transformations preserving edge distances distribution of triangulated mesh between simulated and unfolded objects.

1.4.2 Specific Objectives

1. Design, implement, and program a spring mesh energy conservation and barycentric deformation algorithm to perform unfolding on solid objects.
2. Create and apply quality estimators based on geometric features (like the distance between vertices) to compare unfolding models. In particular, the unfolding algorithm developed will be compared against U-Fo unfolding algorithm.
3. Measure complexity and CPU performance of the new algorithm.

1.5 Contributions

A summary of the contributions of this work would be:

1. Related to objective 1, we identify the following contributions: (1) Develop a new geometric based unfolding algorithm; (2) Surface extraction algorithm from point cloud dataset based in 'Alpha-shape' method; (3) Improve performance in energy release implementation.
2. Associated with objective 2 we: (4) Design an estimator and a methodology to measure quality in unfolding algorithms; (5) Create data and distribution visualizations for point cloud datasets and geometric meshes.
3. Concerning objective 3: (5) Measure complexity of the new unfolding algorithm; (6) Compare mean performance against U-Fo in the comparable unfolding steps.

Chapter 2

Conceptual Framework

In this chapter, we will summarize fundamental and state of the art concepts needed to design, implement and test the new unfolding algorithm.

2.1 Fundamental Concepts

2.1.1 Resource Assessment Estimation

Mineral resource estimation is defined as a technique to determine and define ore tonnage and grade of a geological deposit. A resource estimation process usually involves the construction of a geological block model, which is a discrete representation of the ore body to be estimated [22].

ALGES investigators define their resource estimation pipeline as the union of the following steps [8]:

1. Geological Modeling: In this step, a discrete representation of the ore body is created. Usually, cubic blocks are used to create this model. A point cloud representation can be created from the block model taking the centroid of each cube.
2. Unfolding: The objective of this step is to remove folds from the block model, maintaining some geometrical properties.
3. Restore Displacement: Some discontinuities known as faults are removed from the model in this step.
4. The Kriging Estimation, also know as "Gaussian Regression", is used to interpolate values and estimate the grade of mineral.
5. Finally a Backward Transformation is applied to maintain dimensions and scales against initial measures (drill holes).

The algorithm used in steps 2 and 3 directly relates to the amount of mineral estimated [26]. As a result, preserve spatial continuity in the unfolding step will impact the accuracy of the estimation.

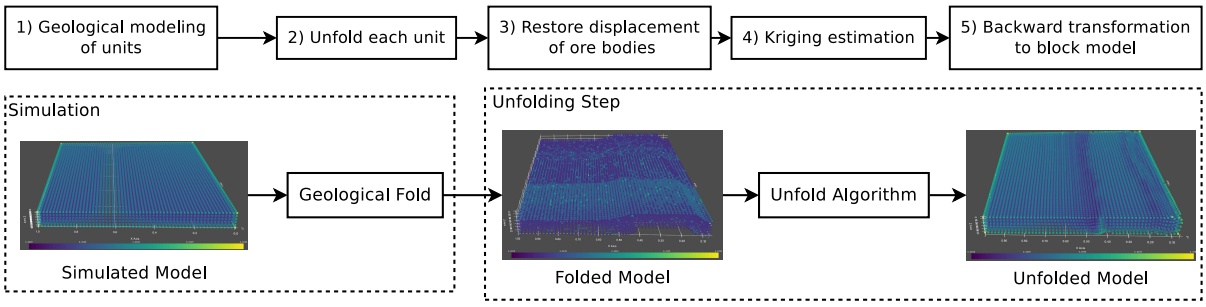


FIGURE 2.1: Resource assessment pipeline for ore estimation, in which remove folds and faults are depicted in steps 2 and 3 [8]. In order to test unfolding algorithms. New steps can be added in parallel to use simulated models instead of field cases.

2.1.2 Geometric Algorithms

Computational geometry emerged from the field of algorithm design and analysis. Over time geometric algorithms (algorithms used to solve geometric problems) have grown to be a field and a community with its own journals and researchers[5]. Some concepts from geometrical computation and algorithms used in this Thesis will be described in the following sections.

Meshes

A mesh is a type of conforming triangulation over a closed bounded domain (Ω) in R^2 or R^3. A triangulation T is a mesh of Ω if:

1. $\Omega = \bigcup K$, with $K \epsilon T$ being the elements in triangulation.
2. The interior if every element K in T is non empty.
3. The intersection for the interior of two elements is empty.

Most computational schemes using a mesh as a spatial support assume a conforming mesh in which the triangulation T also comply:

1. The intersection of two elements in T is either: the empty set, a vertex, an edge or a face (when $d = 3$)

Mesh elements are the basic components of a mesh. An element is defined by its geometric nature (triangle, quadrilateral, etc) and a list of vertices.[6]

Delaunay Triangulation

A Delaunay Triangulation (DT) for a set of points **P** in a plane, is a triangulation such that no point is inside the circumcircle of any triangle within the triangulation. Although the classical definition of DT is made in a 2D basis, it is possible to extend this concept to 3D points and tetrahedral meshes. Nevertheless, the 3D version of DT loses some properties compared to 2D version.

Convex Hull

A Convex Hull of a set of points can be defined as the smallest convex set containing the points. Convex Hulls can be defined for points in d-dimensions, but in particular, for 3-dimensional space, they have various applications such as value interpolation or finite element calculation. In 2D and 3D dimensions, the Delaunay Triangulation is also a Convex Hull over the data [5].

From the Convex Hull, exist some algorithms, like alpha-shapes, that extract a Concave Hull, structure that is useful for surface extraction over a point cloud dataset.

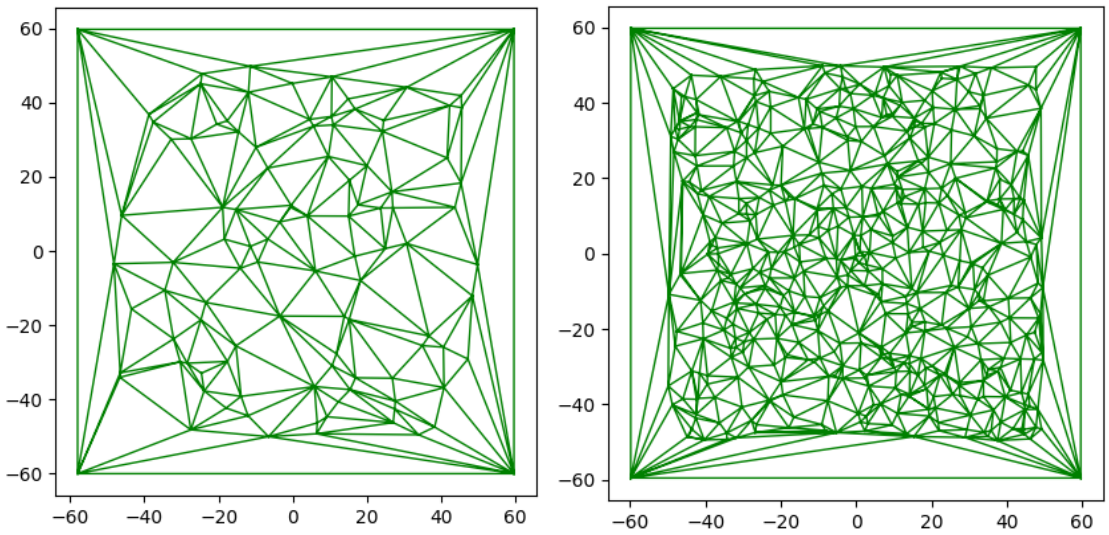


FIGURE 2.2: Delaunay Triangulation over 2D random points in a square.

2.1.3 Statistic Estimators

In statistic, a estimator is a procedure to calculate a quantity based in observed data. Some estimators, for example the "Mean Squared Error" (MSE), are used to measure the quality of errors between two statistic distributions [29]. Having two distributions, Y_i and $\hat{Y}_i$, we can calculate the MSE as:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

A lower MSE is a symptom of lower overall error between the distributions and can be used to determine which estimation seems better.

2.1.4 Heuristic Algorithms

In Computer Science a heuristic can be defined as an approximate solution for hard optimization problems. Usually, heuristic algorithms rely on methods used in statistics or calculus as instances of utilizing repeated instructions to produce a preferred candidate. For example, some well-known procedures that are considered heuristics are: The "Monte Carlo" method or "Particle Swarm" optimization [17].

Some of the algorithms used in this Thesis, such as: The"Spring Mesh Energy Optimization" for surface flattening; The "Alpha Shape algorithm" for surface extraction; Or the "Barycentric Transformation" for volume deformation, are considered heuristics.

2.2 State of Art

2.2.1 Geological Restoration

Geological or section restoration is a technique to progressively deform a geological section. It can be used to provide information on the early stages of geological development. A section suitable to be un-deformed [1] to a reasonable geometry is known as a balanced region. Two basic deformations that have to be restored are faults and folded regions.

Two main 2D deformation processes have been adapted to the 3D surface flattening problem: (1) flexural slip for compressional domains and competent rocks and (2) shear along vertical or inclined vectors extension. Those mechanisms have been applied by different authors to unfold 2D surfaces. Gratier et al. (1991) [10] using a flexural slip performed unfolding minimizing

[1] In Geometry and Mathematics a "Developable Surface" is defined in a formal way [14]

changes in area and length of a triangulated surface. Williams et al. (1997) [28] modified this approach using the sum of rigid translation, rotation and constant area as a quality measure. Rouby (2000) [23] used Gratier approach to perform restoration of complexly folded and faulted surfaces. All the techniques mentioned use geometrical or statistical (MSE) measures of quality to assure proper surface unfolding.

For complex folded and faulted surfaces restoration, Rouby et al. (2000), added extra information to the process by letting the user choose some guide vectors for homogeneous a heterogeneous inclined shear. Other methods of Geological restoration like Horna et al, 2010 [11]; and Tang et al., 2017 [25] have proposed procedures based on the minimization of elastic tensors and integration of dynamic equations related to the geological deformation force applied.

2.2.2 Tetrahedral Mesh

To maintain geometric properties in the dataset and generate a 3D connection graph from a cloud point, a tetrahedral mesh can be used. Furthermore, Delaunay triangulation can be used to perform a triangle mesh assuring some geometric properties such as minimum between neighbors. Delaunay triangulation in a 3D dataset losses some interesting properties in comparison with the 2D version, although to create the nearest neighbor graph or the Voronoi diagram it is good enough [13].

A tetrahedral mesh obtained by Delaunay triangulation defines a convex hull in the data. The external faces of the simplices conforming the complex hull form a triangle surface that can be used as a base surface for the solid body embedded in the data. The surface obtained in this way can be improved to better match the curvature of the external surface if some information of the point normals is known or can be estimated.

The Delaunay tetrahedral mesh is unique in cases where no more than four points lie on the same surface of a sphere. Due to precision uncertainty, the algorithm used for Delaunay triangulation should be robust when working with field data.

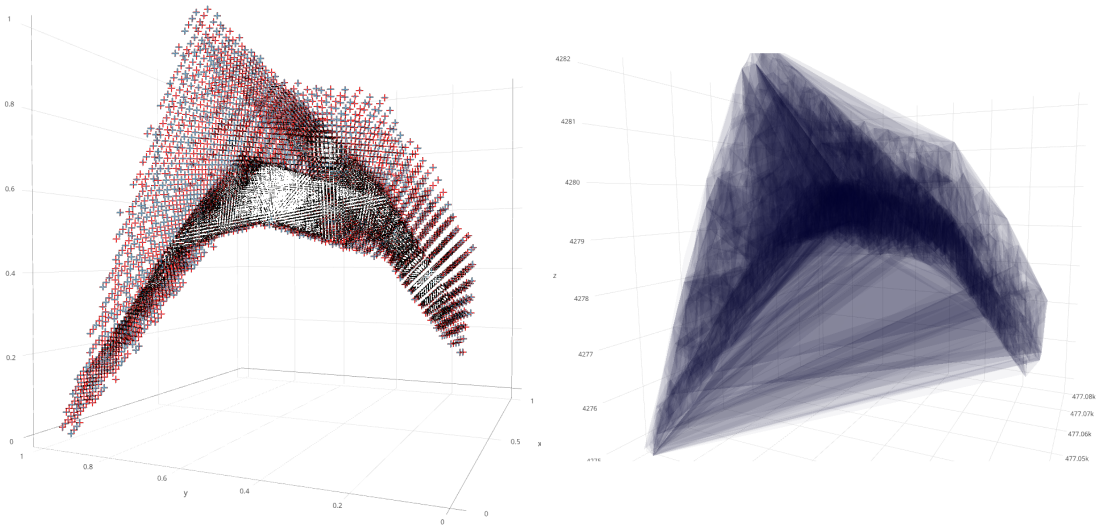


FIGURE 2.3: At left a point cloud dataset of 6510 samples. At right the tetrahedral mesh calculated with "Delaunay Triangulation" with a result of 38376 tetrahedrons.

2.2.3 Surface Extraction

It is possible to extract an external surface from a point cloud dataset representing a solid object. Nevertheless, most of the algorithms do not provide theoretical guarantees and are susceptible to noisy data or outliers. Four algorithms for surface reconstruction from a point cloud dataset will be discussed: "Poisson surface reconstruction", "scale-space surface reconstruction", "advancing front surface reconstruction" and "3D alpha shapes".

1. The **Poisson surface reconstruction** (PSR) [1] takes as input a point cloud with oriented normals and outputs a triangulation surface matching external points using a Poisson distribution. The algorithm assumes no outliers and little noise in the dataset. Poisson surface reconstruction solves an approximate indicator function of the inferred solid, whose gradient matches the input normals. To use this algorithm, a previous phase of normals estimation should be performed.

2. **Scale-space surface reconstruction** (SSR) [15] takes as input an unordered set of points, sampled on the surface of the solid object, and outputs a surface mesh interpolating the set. The algorithm gradually increases the scale of neighbor points to interpolate a surface function that is smoother as the scale increases. Scale-space does not require oriented normals as PSR does, however it requires surface sample points as an input and can fail in the output if interior points are given.

3. **Advancing Front Surface** (AFS) [3] is a Delaunay based algorithm that outputs triangulation surface from an unstructured set of points. The algorithm incrementally selects

suitable triangles to add to the front. At each step, the most plausible triangle is selected, such that all the triangles selected generate an orientable manifold triangulated surface.

4. **3D Alpha shapes** [4] assumes that a set of 3D points in space form a "shape" that can be reconstructed from the initial set. One can think that an alpha shape is like a huge mass of ice-cream and the points in the set are chocolate pieces, the algorithm tries to reconstruct the ice-cream shape from the chocolate using a configurable parameter (α) which give us distances between points.

Selecting a surface reconstruction algorithm depends mostly on the dataset topology and information given from each point. In the case of PSR, oriented normals are needed as an input and are not estimated inside the algorithm; SSR is a perfect choice if the point cloud is sampled from a surface like LIDAR [2] sampled point cloud; finally, Delaunay based algorithms such as AFS are useful if no other information than the geometry is known.

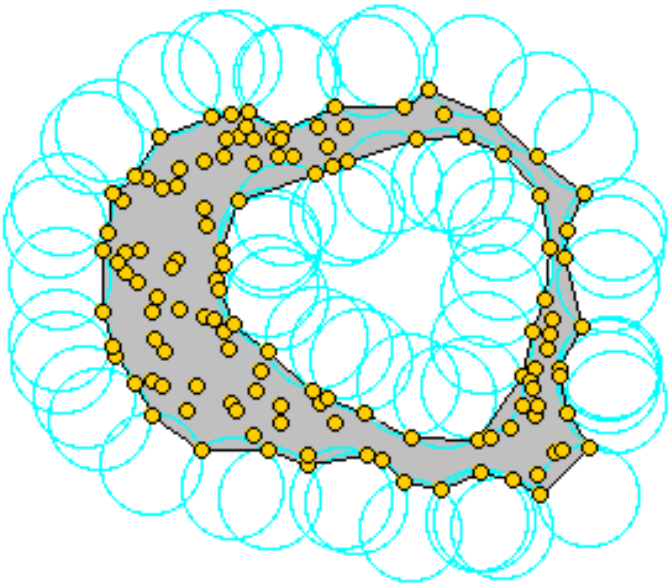


FIGURE 2.4: Alpha-shape representation for 2D dataset. Image downloaded from CGal user manual.

2.2.4 Surface Flattening

Surface flattening is a technique of dimensional reduction in which a 3D surface is flattened to a 2D plane keeping some geometric information in the output plane, but losing the third dimension that defines the input surface. Wang et al. (2002) [27], presented an algorithm, which uses a

[2] Stands for "Light Detection and Ranging", is a method to sample points in space using laser technology

spring mesh energy model to perform surface flattening, preserving the area of triangulated mesh and length of edges. In another approach to the problem, Zigelman et al. (2002) [30] perform multidimensional scaling (MDS) with a set of mathematical techniques that tries to encapsulate the geometric structure, to solve surface flattening and the inverse problem of texture mapping.

Wang et al.[27] method will be used in this Thesis for surface flattening. Wang algorithm model surface mesh as a spring mesh system with the following energy and force equations per vertex:

$$E(P_i) = \sum_{j=1}^{n} \frac{1}{2} C(|P_i P_j| - d_j)^2$$

$$\vec{f}(P_i) = \sum_{j=1}^{n} C(|P_i P_j| - d_j)\vec{n}_{p_i p_j}$$

where C is the spring constant, $|P_i P_j|$ the distances between vertex P_i and P_j in the output plane, d_j is the distance between Q_i and Q_j (vertices in input 3D surface), $\vec{n}_{p_i p_j}$ is the unit vector pointing to P_i to P_j and n is the number of nodes connected to P_i.

Adding all vertices energy and assuming a conservative system ($M\ddot{q} + Kq = 0$) allows us to solve iteratively this system using physics dynamic equations

$$\ddot{q}_i(t) = \frac{f_i(t)}{m_i}$$

$$\dot{q}_i(t + \Delta t) = \dot{q}_i(t) + \Delta t \ddot{q}_i(t)$$

$$q_i(t + \Delta t) = q_i(t) + \Delta t \dot{q}_i(t) + \frac{\Delta t^2}{2} \ddot{q}_i(t)$$

Finally, we can measure accuracy with the final relative area difference $E_s = \frac{\sum |A - A'|}{\sum A}$, where A is the actual area of one patch on the surface before development and A' is the area of its corresponding patch after development.

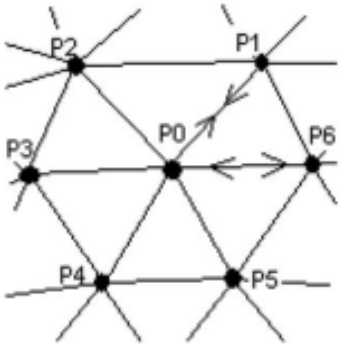


FIGURE 2.5: Spring mesh energy system. P_i represents vertices in projected surface into a plane

2.2.5 Barycentric Deformation

Let $\mathbf{x}_1, \ldots, \mathbf{x}_n$ be the vertices of a simplex in an affine space "A". If, for some point $\mathbf{p}$ in "A";

$$(a_1 + \cdots + a_n)\mathbf{p} = a_1\,\mathbf{x}_1 + \cdots + a_n\,\mathbf{x}_n \tag{2.1}$$

Then we say that the coefficients $(a_1, \ldots, a_n)$ are **barycentric coordinates** of $\mathbf{p}$ with respect to $\mathbf{x}_1, \ldots, \mathbf{x}_n$. If the coordinates are also bounded with the restriction $\Sigma a_i = 1$ then barycentric coordinates are also a type of homogeneous coordinates [7].

Barycentric coordinates are used to interpolate values inside of a geometry body. For example, Huang et al [12] use a control mesh and barycentric coordinates to deform any volume by changing the control mesh. Barycentric coordinates $(a_1 \ldots a_n)$ that are defined with respect to a polytope instead of a simplex are called **Generalized Barycentric Coordinates**[18]

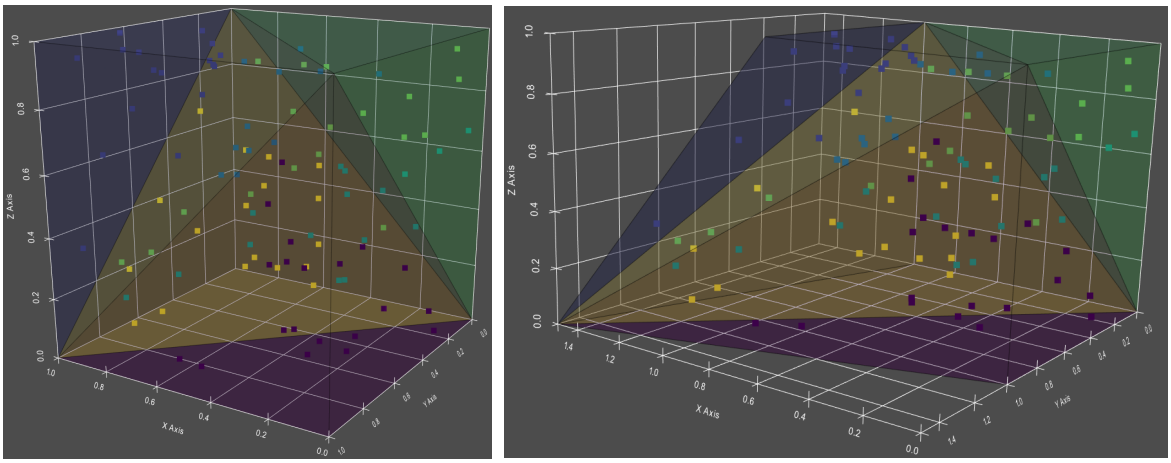


FIGURE 2.6: At left a cube of length one divided into tetrahedron. At right a deformed cube with a vertex moved to coordinate $(1.5, 1.5, 0)$. The points inside the cube are translated with barycentric transformation using initial and final tetrahedral coordinates.

2.2.6 Quality Measures

In Ojeda's Thesis, [21] a metric to measure the neighborhood of a point is proposed as a quality estimator for unfolding algorithms. The metric used in this Thesis is a variant of Ojeda's estimator and is defined as follows:

Having a point X_i, and his N nearest neighbors $\{p_1^{x_i}, \ldots, p_N^{x_i}\}$ then the **Neighbor Mean Distance** of the point will be:[3]

$$\mathrm{NMD}(X_i) = \frac{\sum_{n=1}^{N} \|X_i - p_n^{X_i}\|}{N} \tag{2.2}$$

We expect that the overall neighbor distance of the dataset should be preserved after the unfolding process, meaning that the relative location of unfolded points, in reference to his neighbors, must be maintained after the deformation. An error distribution between two datasets X and Y can be defined like:

$$\epsilon(X, Y) = [\epsilon_i(X, Y)] \tag{2.3}$$

$$\epsilon_i(X, Y) = \frac{|\mathrm{NMD}(X_i) - \mathrm{NMD}(Y_i)|}{\mathrm{NMD}(X_i)} \tag{2.4}$$

[3]The mod $\|\cdot\|$ operator is defined as the euclidean module of a vector

And the total quality error will be the sum of ϵ_i:

$$\sum \epsilon(X, Y) = \sum_i \epsilon_i(X, Y) \tag{2.5}$$

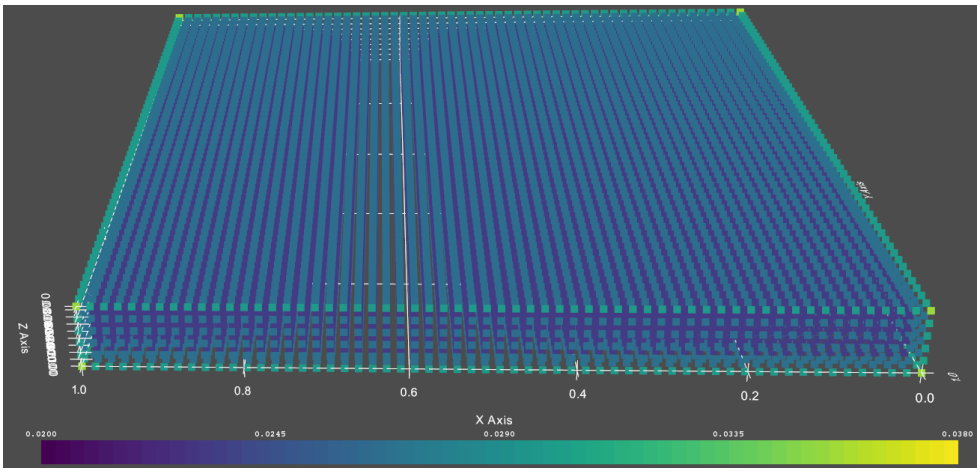


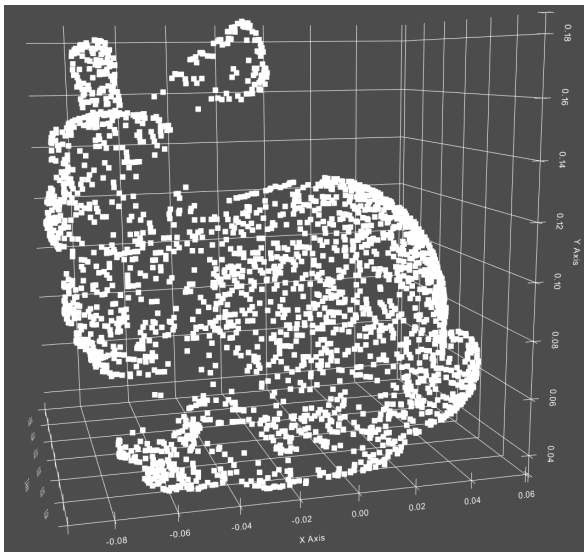
FIGURE 2.7: NMD index depicted over a parallelepiped

2.3 Software Tools

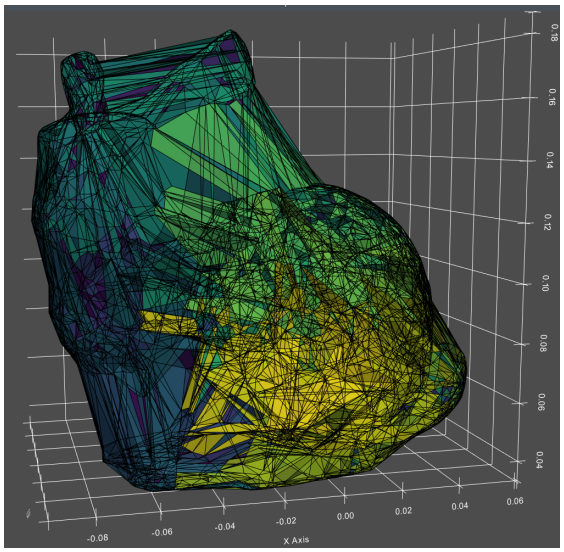
2.3.1 CGal

Is a library in C++ developed via a collaborative effort of various scientists. CGal is used in various areas needing geometric computation, such as: geographic information systems, computer-aided design, molecular biology, medical imaging, computer graphics and robotics.

The library offers state of the art algorithms in the following fields: triangulations, Voronoi diagrams, Boolean operations on polygons and polyhedra, point set processing, arrangements of curves, surface and volume mesh generation, geometry processing, alpha shapes, convex hull algorithms, shape reconstruction, AABB and KD trees



(A) CGal test point cloud.



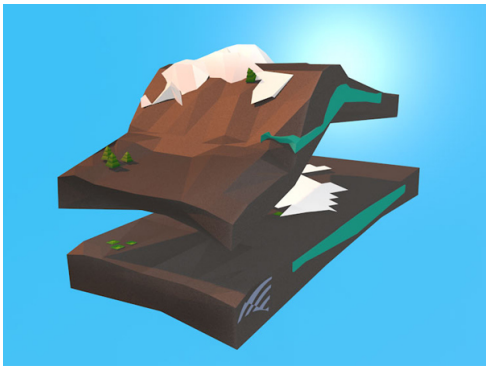
(B) CGal test mesh using alpha-shapes.

2.3.2 UFo

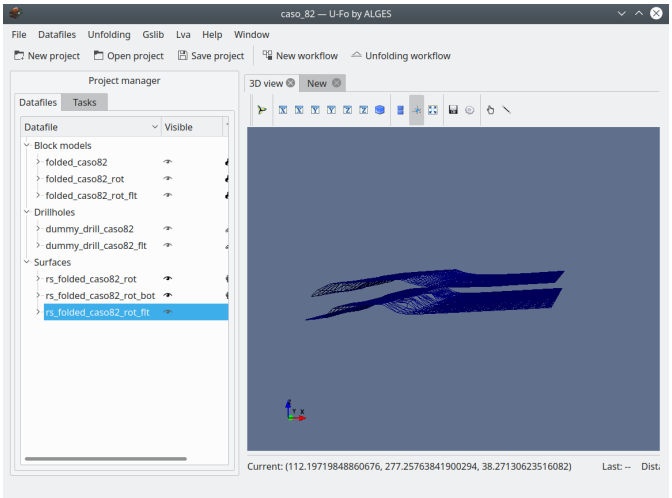
Software created by ALGES group that aims to solve the problem of geological resource assessment over a deposit affected by a complex geometry.

U-Fo software provides the following features: unfolding of geological bodies, anchorage of geological bodies, basic exploratory data analysis, Kriging and inverse of distance, 3D visualization and manipulation of models and drill-hole data.

U-Fo will be used to compare quality and performance between the new unfolding algorithm.



(A) U-Fo logo, downloaded from http://www.alges.cl/.



(B) U-Fo Screen.

Chapter 3

Proposed Solution

3.1 Novel Approach

A new methodology to unfold solid bodies was designed and implemented. The developed algorithm takes as an input a set of points $X = [X_i]$ of size N representing the discrete model of a geological volume in a 3D coordinate space and outputs another set of points Y of the same size but with different coordinates. We can describe the proposed algorithm as a function $F(X) = Y$ being X and Y matrices of $(Nx3)$ which in each row have a (x_i, y_i, z_i) coordinate vector. The function described follows various transformation steps until converges to an unfold point cloud. These steps are summarized as follows:

Algorithm Steps Summarized

1. Pre-process: This step is performed before the formal algorithm. A X_o matrix is constructed from a *"comma-separated values" (CSV)* file and scaled into $[0 \dots 1]$ coordinate values. Then, the values are rotated into their principal component directions to obtain the X input matrix.
2. Surface Extraction: A concave hull is extracted from the dataset using an *alpha shapes algorithm* [4]. Then, the surface is divided into a top and bottom surfaces.
3. Surface Flattening: The top and bottom surfaces are flattened into a plane using the energy model algorithm proposed by Wang et al. [27].
4. Mesh Construction: Based on the original and flattened surfaces a uniform control tetrahedral mesh is constructed joining the top and bottom surfaces.
5. Barycentric Deformation: Using the tetrahedral mesh constructed in the previous step, a *barycentric transformation* [12] is calculated and applied to the points inside each tetrahedron using as a referent the original and final meshes. After this step, the Y matrix is obtained.
6. Quality Measures: Comparing initial and final point clouds, we can compute quality measures to evaluate the magnitude of the flattened model.

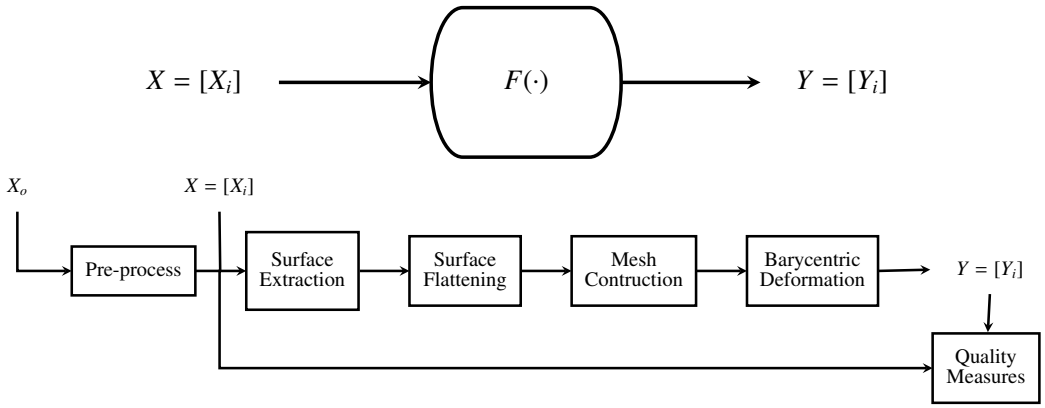


FIGURE 3.1: Algorithm input and output diagram flow. Both input and output are matrices of shape $(Nx3)$.

3.2 Algorithm Description

Detailed descriptions of each step, and variables involved, are provided in the following sections.

3.2.1 Pre-process

The algorithm expects as an input a *point cloud* representing the volume of a geological body. The point cloud is parsed from a csv file with at least three columns named: **centroid_x**, **centroid_y** and **centroid_z**. Each row of the file should have a point with 3 coordinates (x, y, z). We then construct a X_o matrix from the raw input to process a $X = [X_i]$ matrix from these values.

The X_o original matrix is then scaled to $[0 \dots 1]$ values (to minimize precision problems [9]) and rotated along its principal component vectors. This operation is performed to align the $\hat{z}$ axis with the natural *up* vector of the geological block model. With this step, some assumptions can be made in the following steps about the up and down orientations of the data.

We can call $\hat{x_P}$ and $\hat{y_P}$ the two unit vectors obtained applying principal component analysis in the scaled data X_s. After this, a new coordinate system $(\hat{x_p}, \hat{y_p}, \hat{z_p})$ can be obtained with $\hat{z_p} = \hat{x_p} \times \hat{y_p}$. To align the $\hat{z_p}$ vector to $\hat{z}$ a rotation θ_{rot} must be made along the n_z direction, which represent the normal of the plane that contains $\hat{z_p}$ and $\hat{z}$ vectors.

$$\theta_{rot} = arcos(\frac{\hat{z_P} \cdot \hat{z}}{|\hat{z_P}| \cdot |\hat{z}|})$$
$$n_z = \hat{z_p} \times \hat{z}$$

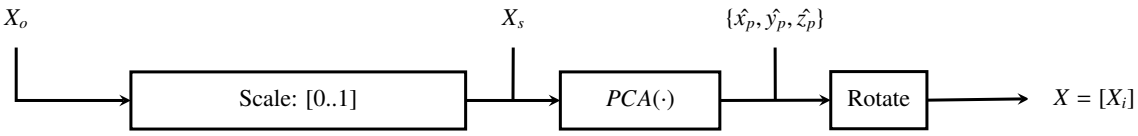
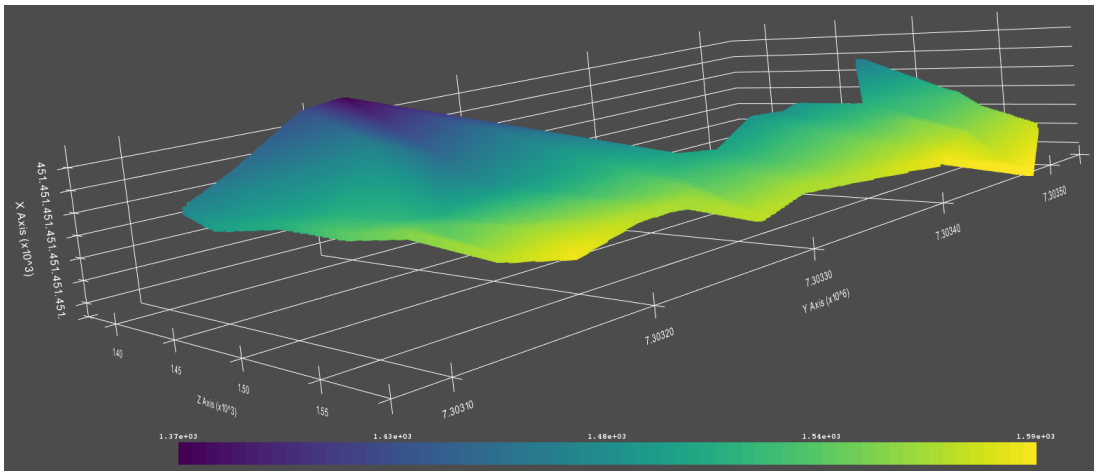
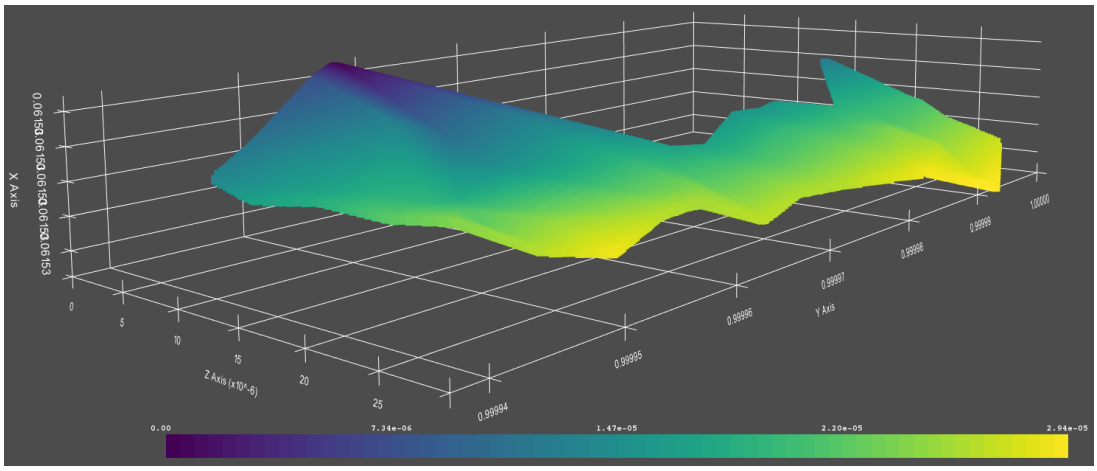


FIGURE 3.2: Pre-process diagram



(A) Raw input point cloud.



(B) Scaled input point cloud.

FIGURE 3.3: Figure at top shows raw dataset as given from input file, and second figure is the dataset scaled between $[0 \ldots 1]$ range.

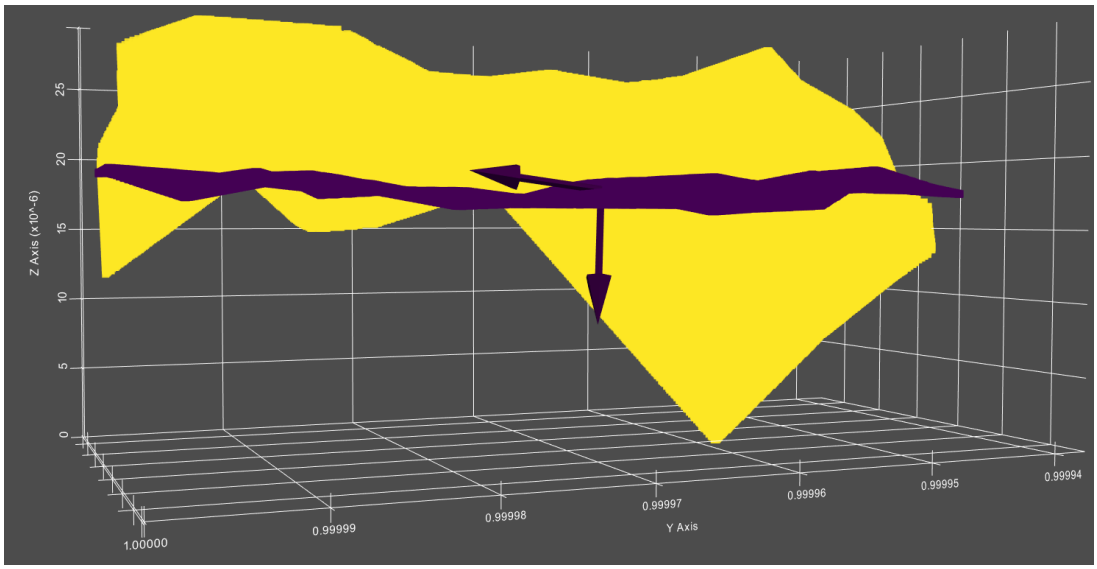
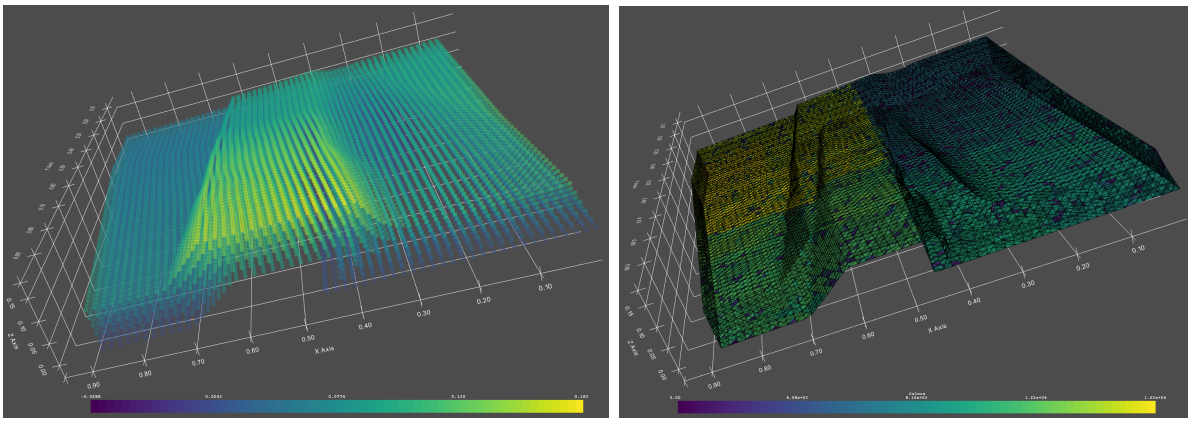


FIGURE 3.4: Point cloud rotated by its principal component vectors. In yellow the original orientation is depicted, while the rotated dataset is shown in purple color. The principal component vectors x_p and y_p are rendered at the centroid.

3.2.2 Surface Extraction

After the pre-processing, we extract a concave hull triangulation using an alpha-shapes[4] algorithm with an α parameter calculated from $X = [X_i]$ matrix. We use C-Gal library with "*Alpha_shape_3*" class and "*find_optimal_alpha*" function among others to calculate the hull. In particular the "*find_optimal_alpha*" function finds an α-value that at least satisfied that all data points are either on the boundary or in the interior of the regularized version of the α-shape [4].

The "Concave Hull" algorithm returns a triangulation with a subset of points on the surface of the geological solid body. The surface obtained might not be a developable surface, for this reason, an additional filter using the orientation of normal triangles is applied to obtain two separate developable surfaces. We will call these two surfaces top and down triangulation and unfold both surfaces separately.



(A) Cloud point $X = [X_i]$.

(B) Concave hull of data.

FIGURE 3.5: At left X matrix representing solid geological body. At right *Concave Hull* extracted from X data.

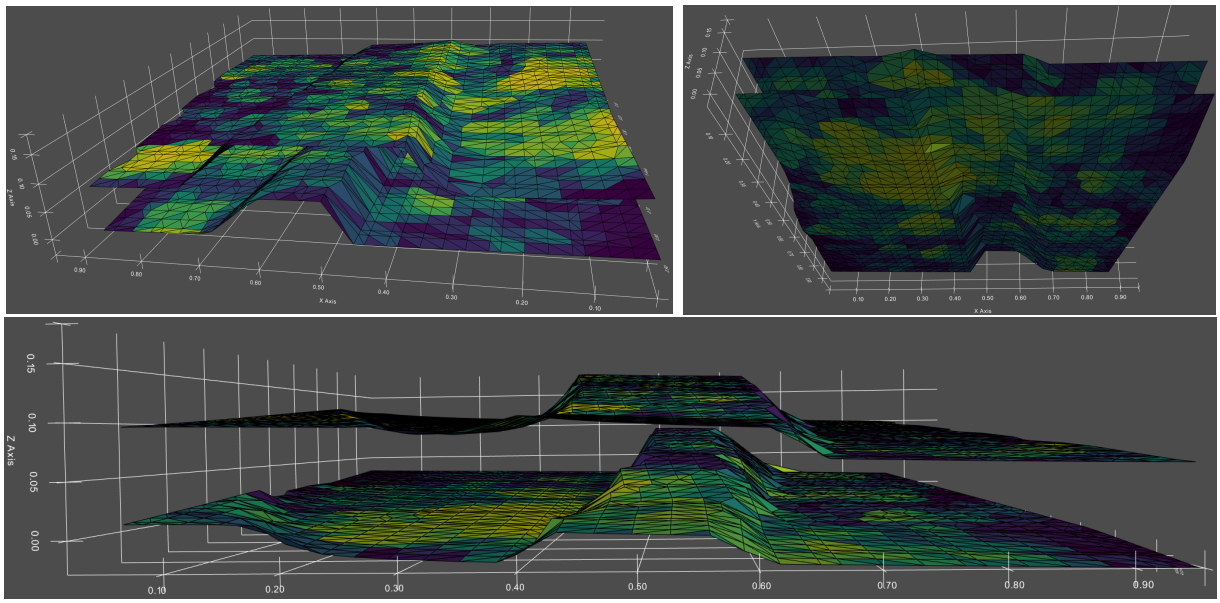


FIGURE 3.6: Top and bottom surfaces

3.2.3 Surface Flattening

Both top and bottom surfaces are unfolded separately using Wang energy optimization algorithm for developable surfaces described in Section 2.2.4. The algorithm tries to minimize the differences between the initial and final triangles area.

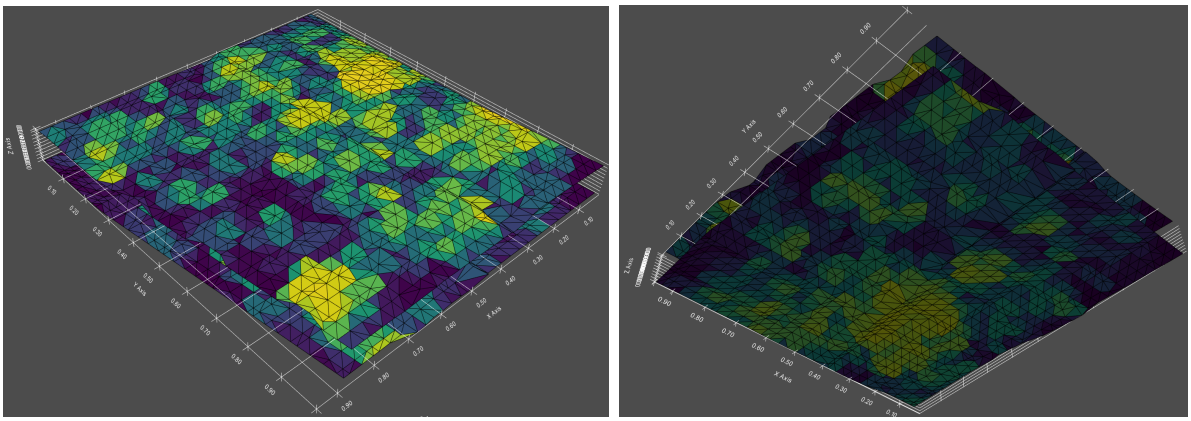


FIGURE 3.7: Flattened top and bottom surfaces

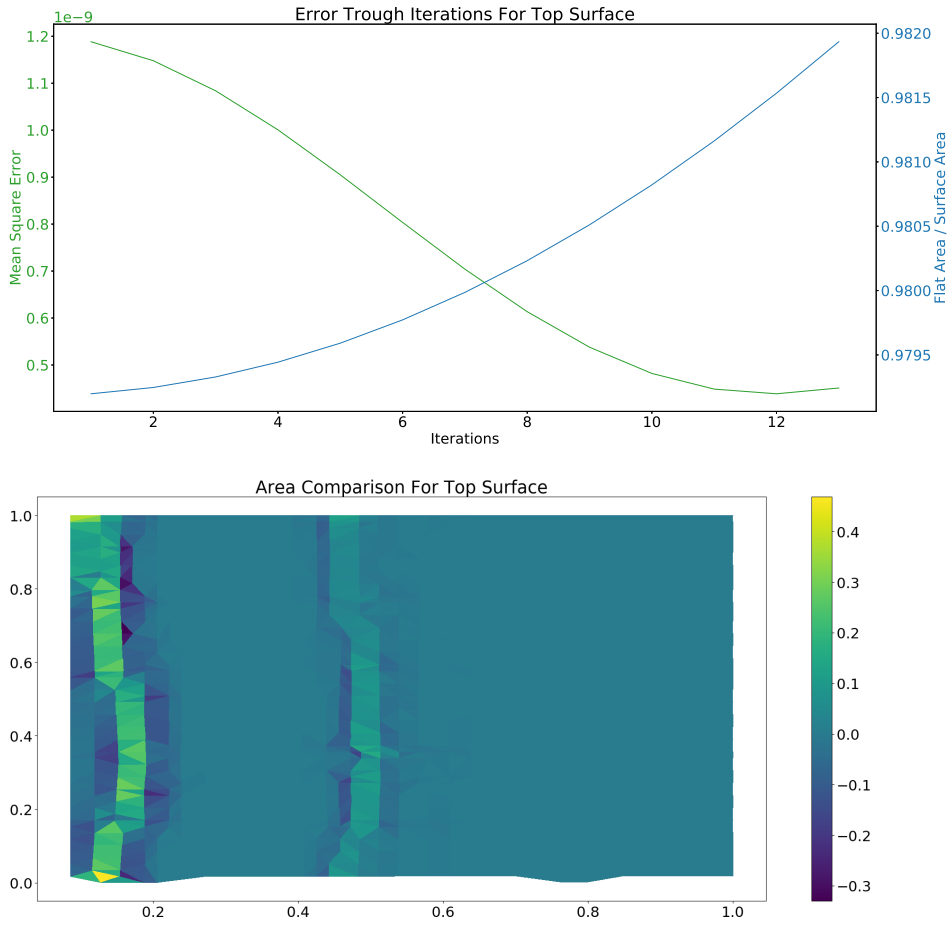


FIGURE 3.8: At left surface unfolding energy algorithm evolution trough 12 iterations; Left axis shows "Mean Square Error" between folded surface and unfolded and right axis shows changes in total area. At right error between folded surface and unfolded one is plotted in each triangle.

3.2.4 Mesh Construction

After the Surface Flattening step, we obtain unfolded versions of top and bottom folded surfaces. We wish to transfer the deformation of both surfaces to the point cloud inside the concave hull. One approach for this is to construct a tetrahedral mesh between the top and bottom surfaces and then interpolate new points based on the deformation of such tetrahedral. The quality of the interpolation obtained will depend on the uniformity of the mesh constructed. To obtain a uniform mesh we follow the next procedure: **Join top and bottom** surfaces into one; **Create prisms** between new surfaces; **Divide** each prism into three tetrahedrons.

Join top and bottom: We wish to obtain new versions of both top and bottom surfaces with added vertices and triangles. After this phase, both surfaces will have the same number of vertexes and the same 2D triangulation but the slope will be maintained. In Algorithm 1 the procedure to join top with bottom surfaces is explained, the same method can be used to join the bottom with the top surface.

Algorithm 1 Pseudo code for join top with bottom surface

$T_{top} := (V_{top}, E_{top}) \rightarrow$ Top triangulation formed of Vertices and Edges
$T_{bot} := (V_{bot}, E_{bot}) \rightarrow$ Bottom triangulation formed of Vertices and Edges
T'_{top} = copy(T_{top})
for V in V_{bot} **do**
  V_{prj} = projection of V in T'_{top}
  if V_{prj} is not in T'_{top} **then**
    T'_{top}.insertDel2D(V_{prj})
  end if
end for

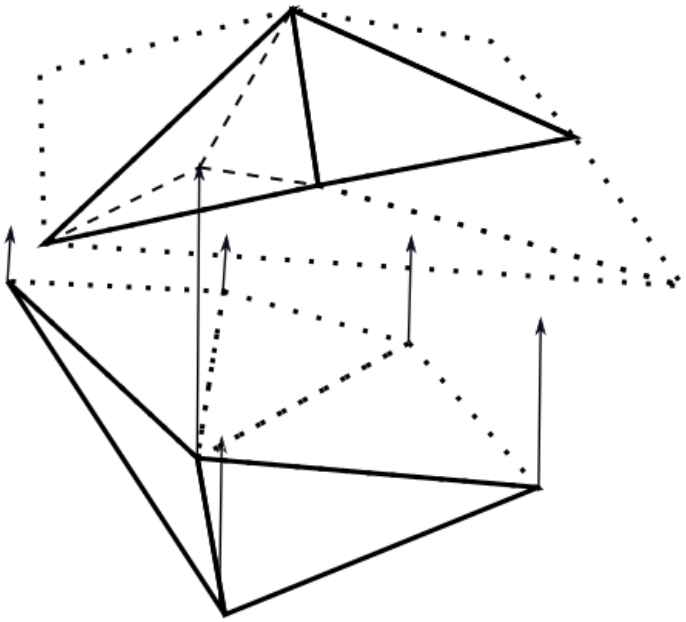


FIGURE 3.9: Projection of a vertex from bottom surface into top surface. The point projected in top surface is then inserted Delaunay into the triangulation

Create Prism: After join top and bottom surfaces, the new surfaces created have the same 2D triangulation and we can construct a set of prisms joining the aligned triangles. Let us call a

pair of aligned triangles T_t and T_b and their vertices $T_t = (V_1, V_2, V_3)$ and $T_b = (V_4, V_5, V_6)$, assuming that both triangles have the same orientation, a prism can be created such that $P = (V_1, V_2, V_3, V_4, V_5, V_6)$.

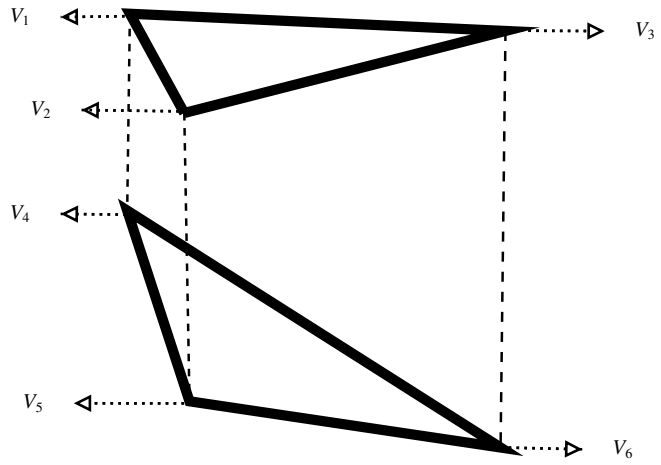


FIGURE 3.10: Prism created by joining top and bottom triangle.

Divide Prisms: Each prism obtained in previous step can be divided into three tetrahedrons. Assuming a prism formed by six vertices $P = (V_1, V_2, V_3, V_4, V_5, V_6)$ we can divide it in: $T_1 = (V_1, V_2, V_3, V_5)$, $T_2 = (V_4, V_5, V_6, V_3)$ and $T_3 = (V_4, V_5, V_1, V_3)$. Applying this division to each prism in the mesh results in a uniform tetrahedral mesh that includes all points between top and bottom surfaces.

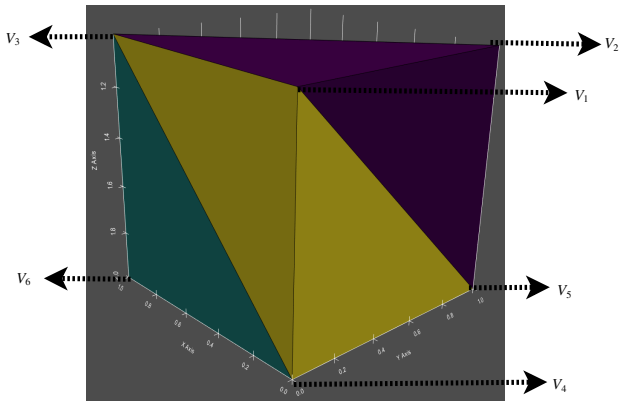


FIGURE 3.11: Prism divided into three tetrahedrons. In purple T_1, green T_2 and yellow T_3

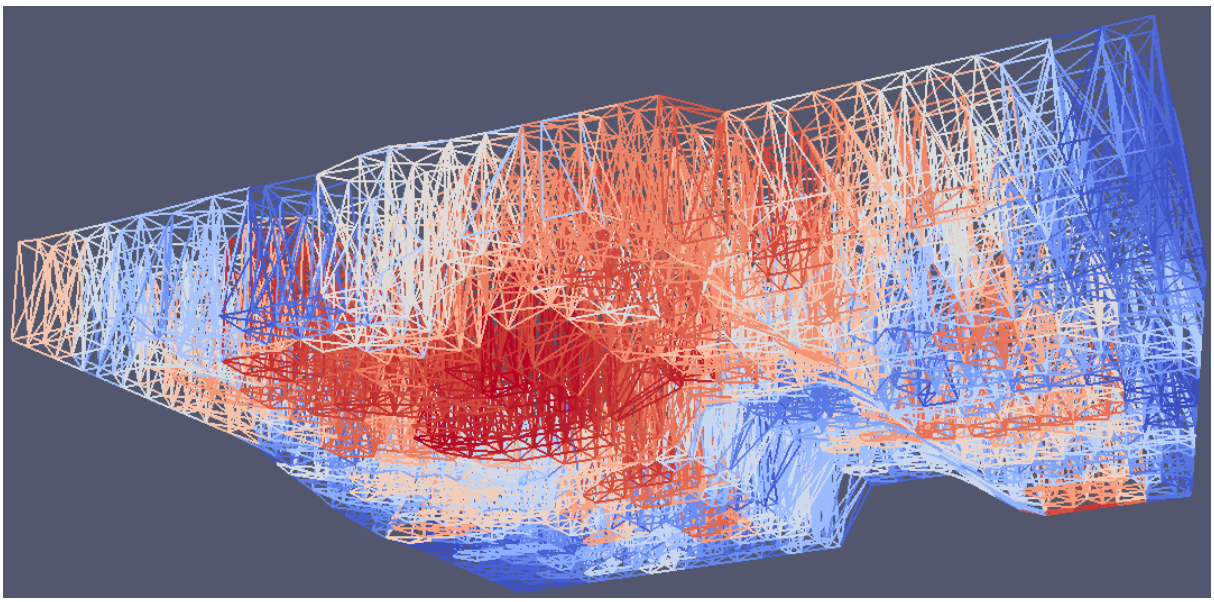


FIGURE 3.12: Tetrahedral mesh wire-frame view

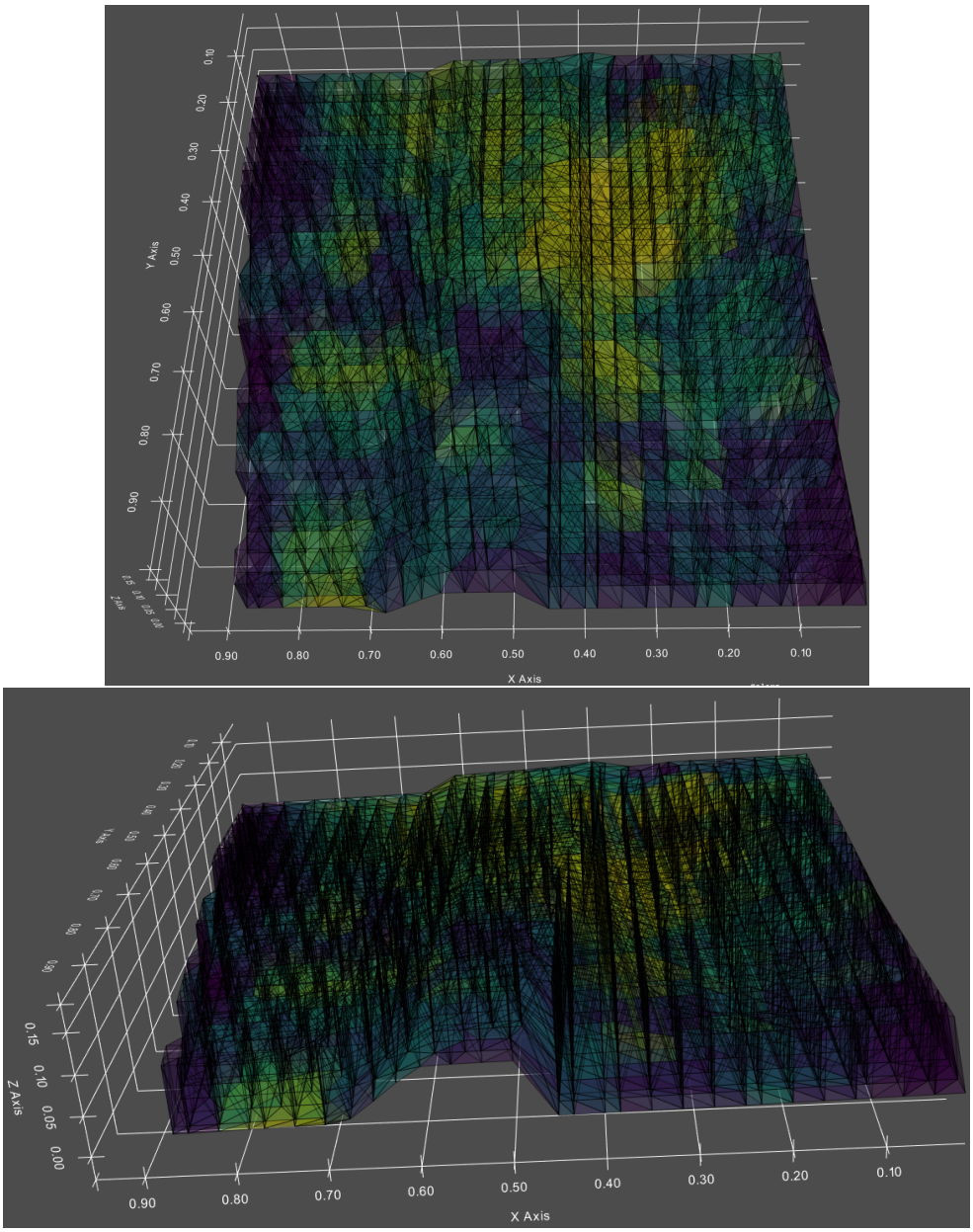


FIGURE 3.13: Tetrahedral mesh with opacity

3.2.5 Barycentric Deformation

Using the mesh constructed in section 3.2.4 and flattened surfaces obtained in section 3.2.3 we can use a barycentric transformation to translate points inside the mesh into a new location.

Assuming a tetrahedron with vertices $\{V_1 \dots V_4\}$ and $V_i = (x_i, y_i, z_i)$, we call the parameters $(c_1, \cdots, c_4)$ barycentric coordinates of some point p such that $(c_1 + \ldots + c_4) \cdot p = c_1 V_1 + \ldots + c_4 V_4$. Such coordinates are called homogeneous coordinates if $\sum c_i = 1$. The homogeneous coordinates can be calculated using the next set of equations:

$$\begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \mathbf{M}^{-1}(p - V_4) \text{ with } \rightarrow \mathbf{M} = \begin{pmatrix} x_1 - x_4 & x_2 - x_4 & x_3 - x_4 \\ y_1 - y_4 & y_2 - y_4 & y_3 - y_4 \\ z_1 - z_4 & z_2 - z_4 & z_3 - z_4 \end{pmatrix}$$

$$c_4 = 1 - c_1 - c_2 - c_3$$

Considering two tetrahedrons $T = \{V_1, \dots, V_4\}$ and $T' = \{V'_1, \dots, V'_4\}$ a transformation function $\mathbf{D}(\cdot)$ it is called a deformation of tetrahedron T into T' if

$$T' = \mathbf{D}(T) \rightarrow \begin{pmatrix} V'_1 \\ \vdots \\ V'_4 \end{pmatrix} = \mathbf{D} \begin{pmatrix} V_1 \\ \vdots \\ V_4 \end{pmatrix}$$

and we can interpolate any point p inside the tetrahedron T to a new location p' inside tetrahedron T' using barycentric coordinates

$$p' = \frac{c_1 V'_1 + c_2 V'_2 + c_3 V'_3 + c_4 V'_4}{\sum c_i}$$

applying this transformation to all points $X = [X_i]$ will unfold the solid body in reference of the flattened surfaces. The algorithm 2 explain the steps to obtain a flattened $Y = [Y_i]$ point cloud.

Algorithm 2 Algorithm for unfold point cloud X into Y

$X = [X_i]$
$Y = []$
for i, X_i in X **do**
  $[V_1, \ldots, V_4], [V'_1, \ldots, V'_4] = \text{find\_tetra\_including}(X_i)$
  $[c_1, \ldots, c_4] = \text{calc\_bary\_coords}(X_i, T)$
  $Y_i = c_1 V'_1 + c_2 V'_2 + c_3 V'_3 + c_4 V'_4$
  $Y.\text{append}(Y_i)$
end for
return Y

3.2.6 Algorithm Complexity

Each step of the algorithm complexity is analyzed in this section and the total overall complexity is estimated assuming an input matrix X of shape $N \times 3$

Algorithm Steps Complexity

1. The Pre-process phase is divided in **Scale**, **PCA calculation** and **Rotate** steps. All these steps are linear in the rows dimension, which leave us with a complexity $O(N)$.

2. Surface Extraction phase is divided in α**-Shape** and **Orientation Filter** steps. The α-shape algorithm complexity is related to a 3D Delaunay triangulation which is $O(N^2)$. After this step a sub-set of external points of length N_e and a triangulation of approximately ${N_e}^2$ triangles it is obtained. The orientation filter is applied linearly in the triangulation, so has complexity $O({N_e}^2)$.

3. Surface Flattening is performed using Wang et al. algorithm explained in Section 2.2.4. This algorithm solves dynamic equations per point (N_e) in the spring mesh. For each point, a subset of adjacent points is used to calculate force and energy equations. The algorithm can be considered $O(N_e)$ once the spring mesh of the referential surface is created.

4. Mesh Construction phase is divided in **Join Top and Bottom** and **Divide Prisms** steps. For the joining step, a search of $O(log(N_e))$ is performed approximately N_e times and a point is added Delaunay each time ($O({N_e}^2)$). 'Divide prisms' is linear in N_e, which leave us with a phase complexity of:

$$O(log(N_e)) \cdot N_e + O({N_e}^2) + O(N_e) \approx O({N_e}^2)$$

after this phase a tetrahedral mesh of length $3 \cdot {N_e}^2$ is obtained.

5. Finally Barycentric Deformation phase contemplates a search in tetrahedral space ($O(log({N_e}^2))$) for N points, which leave us with a complexity of $O(N \cdot N_e)$.

Adding the complexity for all phases gives us the overall complexity for the unfolding algorithm:

$$O(N) + O(N^2) + O({N_e}^2) + O(N \cdot N_e) = O(N^2)$$

In comparison, U-Fo algorithm uses a raytracing algorithm for block model flattening. This algorithm traverses each coordinate in the point cloud (N) and associates it with a corresponding triangle in the referential surface (N_e). The raytracing algorithm is estimated to be $O(N \cdot N_e) = O(N^2)$. Since both unfolding algorithms are $O(N^2)$, they are comparable in performance related to the length of input samples.

3.2.7 Quality Measures

Once an unfolded point cloud has been calculated, we can define a measure to compare mean distances between folded and unfolded datasets. We use Equation 2.2 to quantify the mean

distance between a point X_i and his N nearest neighbors $\{p_1^{x_i}, \ldots, p_N^{x_i}\}$. Then, we construct an error index with the "Neighbor Mean Distance" of both folded and unfolded distributions.

We expect that the overall neighbor distance of the dataset should be preserved after the unfolding process, meaning that the relative location of unfolded points, in reference to his neighbors, must be maintained after the deformation. We can use the NMD quality measure to define a percentage error between points in the dataset using Equation 2.4. Figure 3.14 visualize the overall error of the unfolding process.

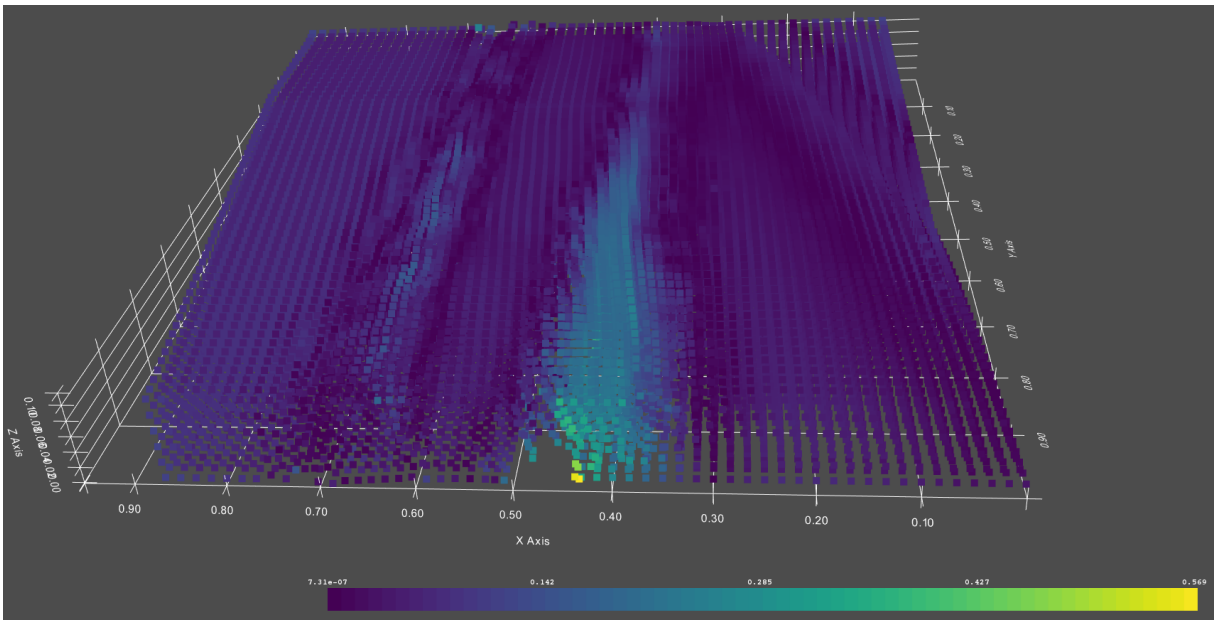
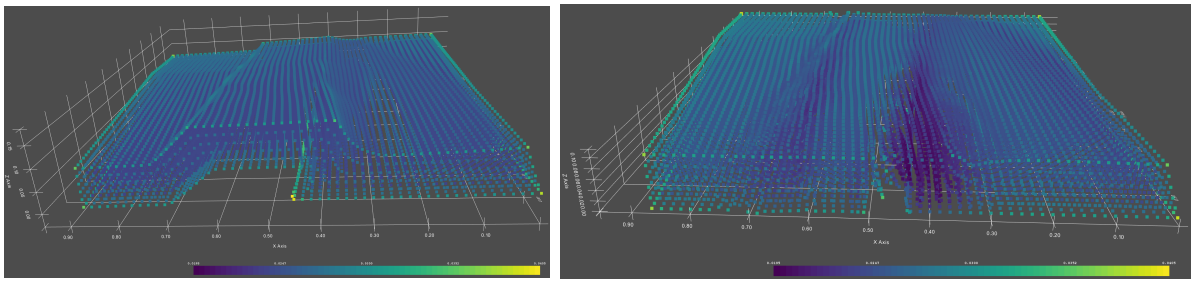


FIGURE 3.14: Unfolded point cloud colored with neighbor distance error against folded point cloud.



(A) Original cloud point data $X = [X_i]$.

(B) Unfolded cloud point data $Y = [Y_i]$.

FIGURE 3.15: X and Y point clouds, colored with their neighbor quality measure ($NMD(\cdot)$)

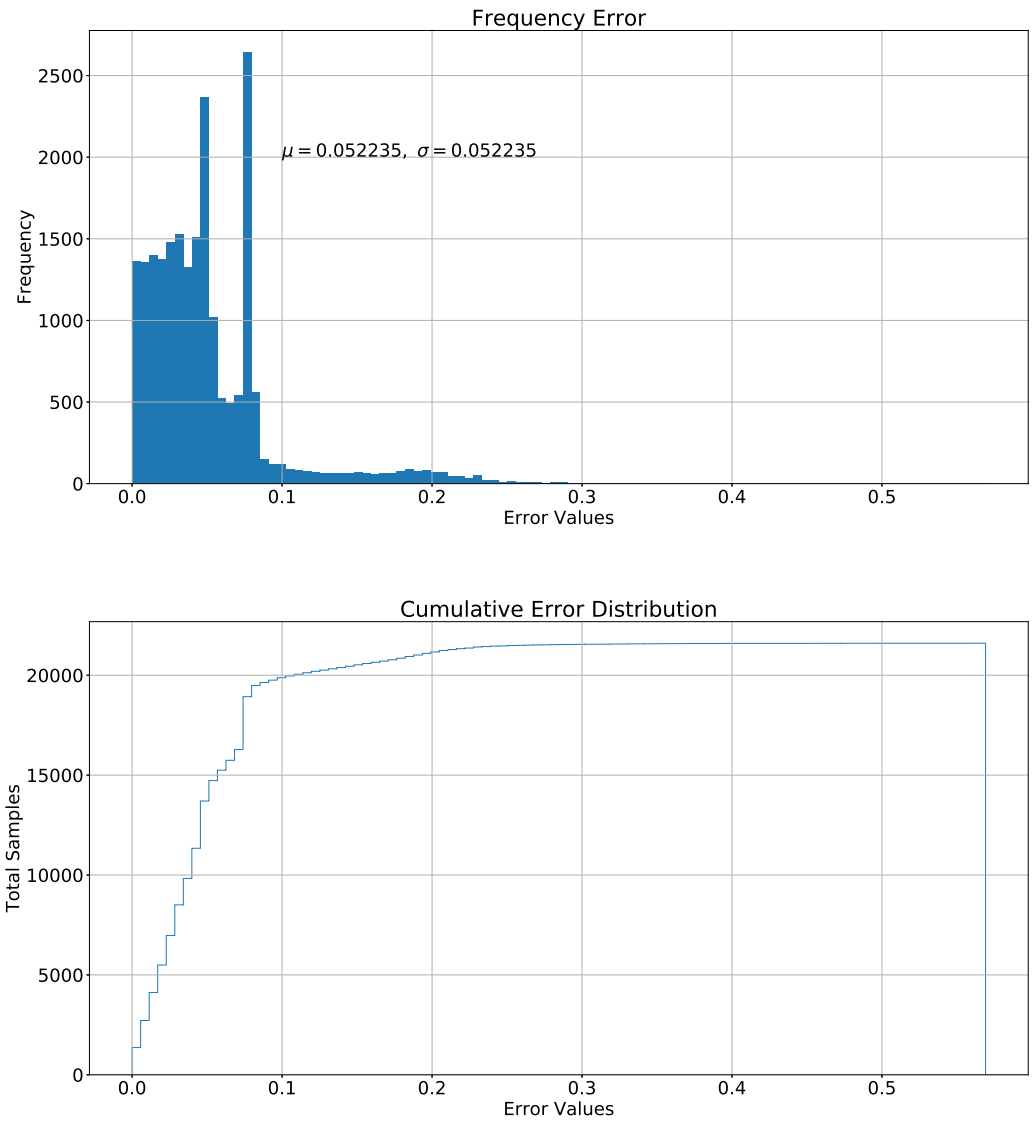


FIGURE 3.16: Histogram and Cumulative error distribution from a point cloud unfolding process. From a total of 21, 600 points, more than the 90% have an error less than 0.1

3.3 Implementation Details

The algorithm described was implemented using python 3.6 with the exception of the surface extraction step, where CGAL was used. The main folder for the project is called "UnfoldPointCloud" and contains python files with the programmed functions. The main files are listed in listing 3.1, while auxiliary folders are listed in listing 3.2. A brief description of each file will be provided in this section, for a complete list of functions, the source code is provided in Appendix A.

LISTING 3.1: python files

```
UnfoldPointCloud
|-- data_utils.py
|-- energy_model.py
|-- geometry_algorithms.py
|-- geometry_utils.py
|-- plot_utils.py
|-- unfold_pc.py
```

LISTING 3.2: auxiliary folders

```
UnfoldPointCloud
|-- concave_hull
|-- datasets
|-- docs
|-- results
|-- sandbox
|-- ufo_energy_release
```

General Issues During Implementation and Limitations

During the implementation of the algorithm, some issues were discovered and solved in the final design. These issues forced in some cases a change of perspective or a natural limitation of the

solution. In this section, some of these issues will be mentioned and discussed per step in the algorithm pipeline.

Algorithm Steps

1. Pre-process: Ore datasets usually have two preferred coordinates (Ex: $\vec{X}$ and $\vec{Y}$) where their dimension is bigger than the last coordinate (Ex: $\vec{Z}$). In preferred directions, datasets can have magnitudes of kilometers while having a few meters in the last direction. This inequality produces various precision problems in geometry algorithms that can be minimized scaling the dataset in $[0 \dots 1]$ range[9]. In addition to scaling, epsilon techniques were implemented for precision problems, nevertheless, in limit cases, some precision issues could raise.

2. Surface Extraction: For surface extraction, algorithms like 'Poisson Surface Reconstruction' or 'Advance Front' were reviewed, choosing 'Alpha Shape' in the end. The 'Alpha Shape' algorithm uses a free α parameter that is calculated and optimized from data. Although for some datasets the optimization of α might not converge and produce non-developable surfaces. In addition, an orientation filter is used after 'Alpha Shape' to separate top and bottom surfaces, the target angle in this filter can also produce errors in steep slopes.

3. Surface Flattening: Wang [27] spring mesh algorithm has known issues with overlapped triangles during energy release. This problem can be mitigated by imposing restrictions on the release system. Nevertheless, In the datasets under testing, this error did not arise,

4. Mesh Construction: Precision problems can also be encountered during this phase. Epsilon techniques were used to mitigate them during the merge of top and bottom surface meshes. Other algorithms for tetrahedral meshing were tested during the implementation

(like 3D Delaunay or Tetgen library[24]) but they were discarded because quality issues in the resulted mesh arise.

5. Barycentric Transformation: Barycentric coordinates calculation is performed per point in a tetrahedron; this can cause errors when the tetrahedron is near to degenerate to a triangle. In these cases is necessary to either move the point inside to nearest well form tetrahedron or calculate barycentric coordinates for a triangle.

Chapter 4

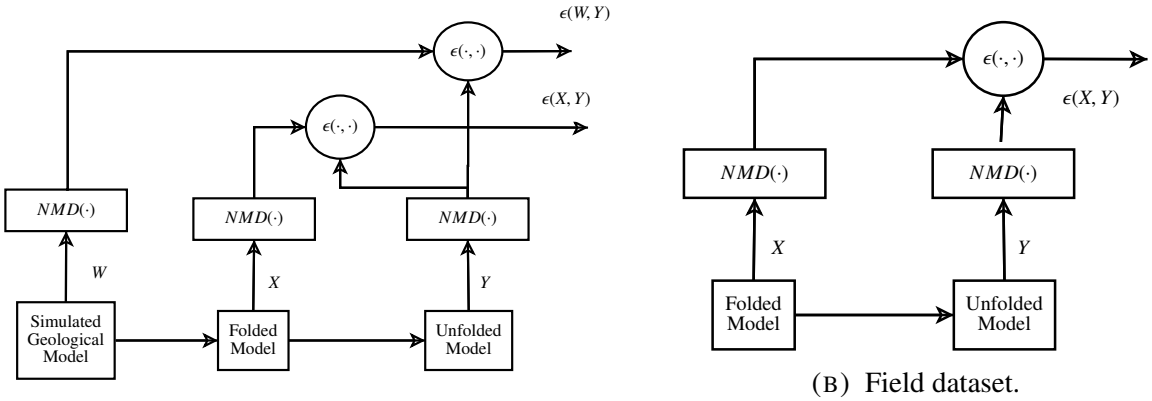
Results

The barycentric algorithm described in chapter 3 was executed in three datasets. In this chapter, we provide the results in performance (execution time) and unfolding quality (using the NMD index of equation 2.2). We used two kinds of geological volume models for these experiments: **computer simulated** and **field data**. In Figure 4.2 and Table 4.1 we summarize the shape and total number of points for each dataset.

Geological block models are expensive and complex to design, consequently, mining companies usually do not openly publish them for the public. For this reason, other restoration investigations, such as Horna et al, 2010 [11]; and Tang et al., 2017 [25], use a couple of datasets for comparison. Therefore, we assume that three models are good enough to make conclusions about the quality and performance of the new algorithm.

To compare performance we can run both unfolding algorithms and measure the execution time for each dataset. In contrast, for quality measurement, we can only directly compare the simulated models, as we know in advance the original geological model (W distribution) that we want to restore. Defining W as the original block model distribution, X as the folded model, and Y as the unfolded model, we can obtain two error distributions: $\epsilon(W, Y)$ and $\epsilon(X, Y)$ using

Equation 2.4. Figure 4.1 shows the flow diagram of error calculation for simulated and field datasets.



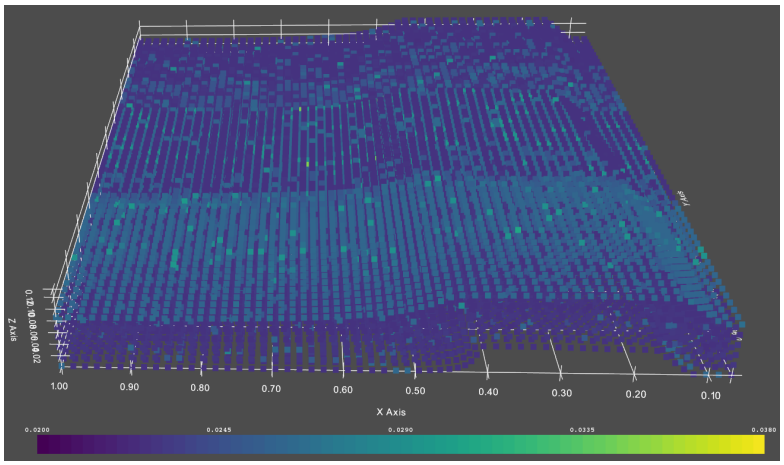
(A) Computer simulated dataset.

(B) Field dataset.

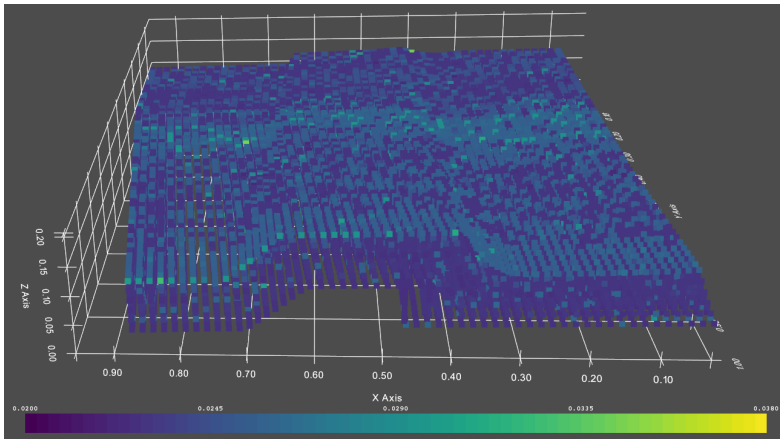
FIGURE 4.1: Computation flow for error quality index (Equation 2.4) in **simulated** and **field data**. Defining W as the original block model distribution, X as the folded model and Y as the unfolded model, we obtained two error distributions: $\epsilon(W,Y)$ and $\epsilon(X,Y)$ for simulated data and just one error distribution: $\epsilon(X,Y)$ for field data.

Datasets	
geological models	input points
sim_case_1	21600
sim_case_2	21600
field_case_1	6510

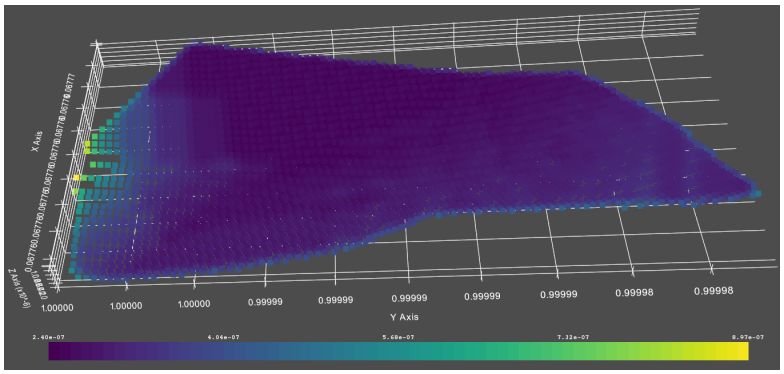
TABLE 4.1: Number of points per dataset.



(A) sim_case_1, 21600 points, 957 points in surface



(B) sim_case_2, 21600 points, 946 points in surface



(C) field_case_1, 6510 points, 346 points in surface

FIGURE 4.2: Normalized point cloud datasets, two simulated (A and B) and one field case (C). In each figure the color represents the value of NMD (Equation 2.2) index per point.

4.1 Performance Comparison

To compare performance between barycentric and UFo unfolding algorithms we have to break them into smaller parts and measure time in each part separately. Both algorithm pipelines have differences and we can only compare some parts directly. For example, UFo does not extract surfaces from the 3D block models and unfolds based on only one reference surface, top or bottom, while the barycentric approach uses both surfaces which add more time to **surface flattening** step.

Tables 4.2 and 4.3 summarize the time needed for each step using three datasets (two simulated and one field case). The results were obtained using a machine with Intel(R) Core(TM) i7-8550U, 16 Gigas of RAM and Ubuntu 18.04 as operative system with a repetition $n = 10$ for each dataset.

We can compare the surface flattening and 3D unfolding parts between algorithms (unfolding part in barycentric algorithm correspond to mesh_construction plus bary_deformation steps). In Figure 4.3 the comparison of both steps is made, surface flattening is faster in the barycentric algorithm in spite of having more points in the surface, while UFo algorithm is faster in the unfolding part because it does not have to build a mesh from a point cloud which simplifies this step. Nevertheless, the performance of both algorithms is comparable in order of magnitude, adding barycentric unfolding as an additional overhead produces, in the worst-case scenario, a difference of approximately 15 seconds.

				Execution Times (seconds)					
geological models	points	points in surface		preprocess	surf_extraction	surf_flattening	3d_unfolding		total_time
							mesh_construction	bary_deformation	
sim_case_1	21600	1969	mean	0.0492	13.2157	12.8987	20.6015	2.251	49.0162
			stdev	0.016	0.3823	0.2367	0.2225	0.0556	0.4722
sim_case_2	21600	114	mean	0.0679	13.0883	13.4587	20.8853	2.236	49.7362
			stdev	0.0493	1.929	1.3272	0.125	0.0956	3.1921
field_case_1	6510	854	mean	0.1784	3.7048	6.5493	6.3229	0.6780	17.4334
			stdev	0.3091	0.2728	0.1587	0.1535	0.0369	0.6956

TABLE 4.2: Execution times for barycentric unfolding algorithm. Columns represents the algorithm steps, and for each, we calculate the mean time and standard deviation of 10 executions.

				Execution Times (seconds)	
geological models	points	points in surface		surf_flattening	3d_unfolding
sim_case_1	21600	957	mean	11.8733	10.7923
			stdev	3.7323	1.0765
sim_case_2	21600	946	mean	16.8282	12.4462
			stdev	1.5495	1.1507
field_case_1	6510	346	mean	7.3151	3.8131
			stdev	0.6375	0.6833

TABLE 4.3: Execution times for UFo unfolding algorithm.Columns represents the algorithm steps, and for each, we calculate the mean time and standard deviation of 10 executions. The steps that can be compare with barycentric algotihm are "3d unfolding" and "surface flattening".

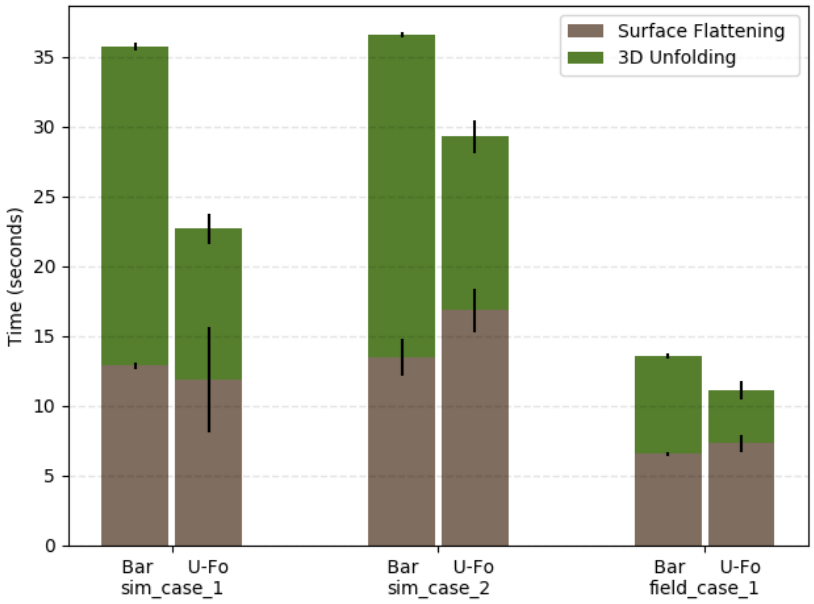


FIGURE 4.3: Performance comparison of "surface flattening" and "3D unfolding" steps for Barycentric and U-Fo unfolding algorithms. The height in the bar plot represents the mean time in each step, and the standard deviation is visualized as an error line above the mean value.

4.2 Quality Comparison

For quality comparison, we used 3 geological datasets. Two datasets are computer-simulated ones, and the other is a field geological block model. For the simulated data, we know a priori the original restoration and the folded model, while for the field data only the folded model is known. A pipeline like the one depicted in Figure 4.1 was executed to calculate $\epsilon(X, Y)$ (and $\epsilon(W, Y)$ for simulated data) [1] random variables with the Barycentric and UFo unfolding algorithms, and we compare the distribution obtained of these variables for both algorithms. A lower overall quality error($\epsilon(X, Y)$ and $\epsilon(W, Y)$) mean that the "**N**eighbor **M**ean **D**istances" of

[1] Errors between distribution were calculated using Equation 2.4

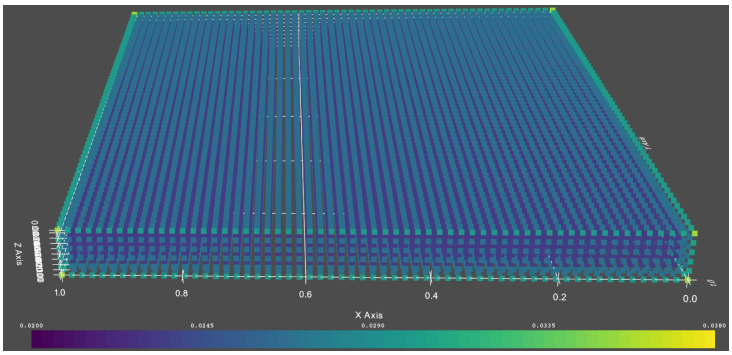
dataset points have little change. We present the results, visualizations and analysis of each dataset in the following sections.

4.2.1 Quality Results for simulation case 1

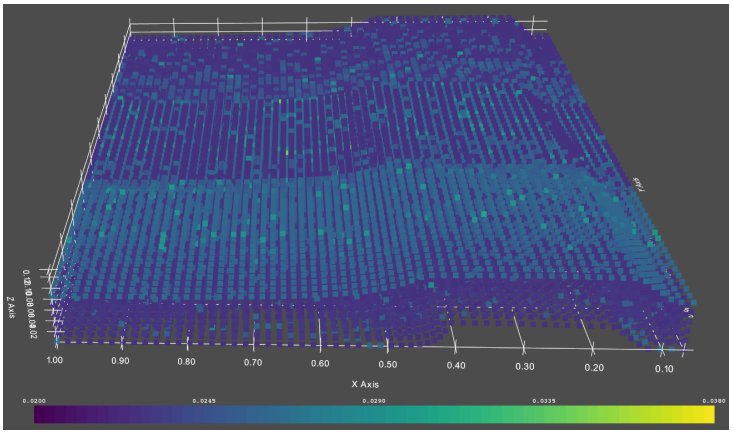
Simulation case 1 starts as a parallelepiped polyhedron that is folded using geological transformations. Figure 4.4 shows the point cloud models of simulation case 1 and the results after the unfolding process using barycentric and UFo unfolding algorithms. The color visualized shows the NMD(·) index (equation 2.2) for simulated, folded and unfolded model.

For simulated data we have the original model before geological transformations (Figure 4.4a). We expect that a good unfolding algorithm will have little changes between the NMD index before folding and after unfolding (i.e NMD(W) VS NMD(Y)). As fast inspection in Figure 4.4 we can noted just with color distribution that barycentric approach does a better job keeping the NMD distributions (Figures 4.4a and 4.4c) than UFo (Figures 4.4a and 4.4d).

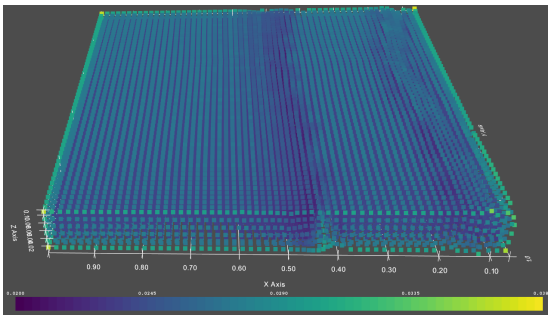
For a more in depth analysis of how unfolding algorithms change NMD index per point, we have calculated and visualized errors distributions (Equation 2.4) in Figures 4.5, 4.6 and 4.7. Comparing plots, is clear that barycentric algorithm have lower $\epsilon(W, Y)$ distribution with $\mu = 0.022$ and $\sigma = 0.015$ versus $\mu = 13.64$ and $\sigma = 4.53$ for UFo algorithm, if we compare $\epsilon(X, Y)$ instead, barycentric approach still got lower values, but differences are less extreme.



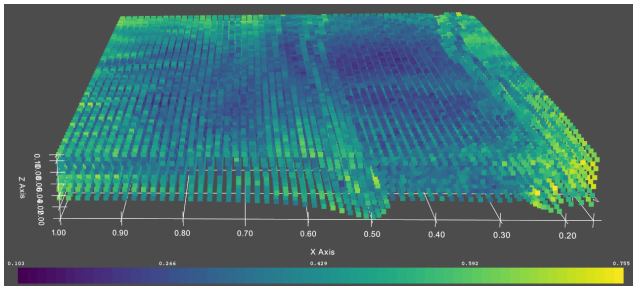
(A) Simulated model colored with NMD(W) per point. Color bar ranges between 0.02 and 0.038 values.



(B) Folded model colored with NMD(X) per point. Color bar ranges between 0.02 and 0.038 values.

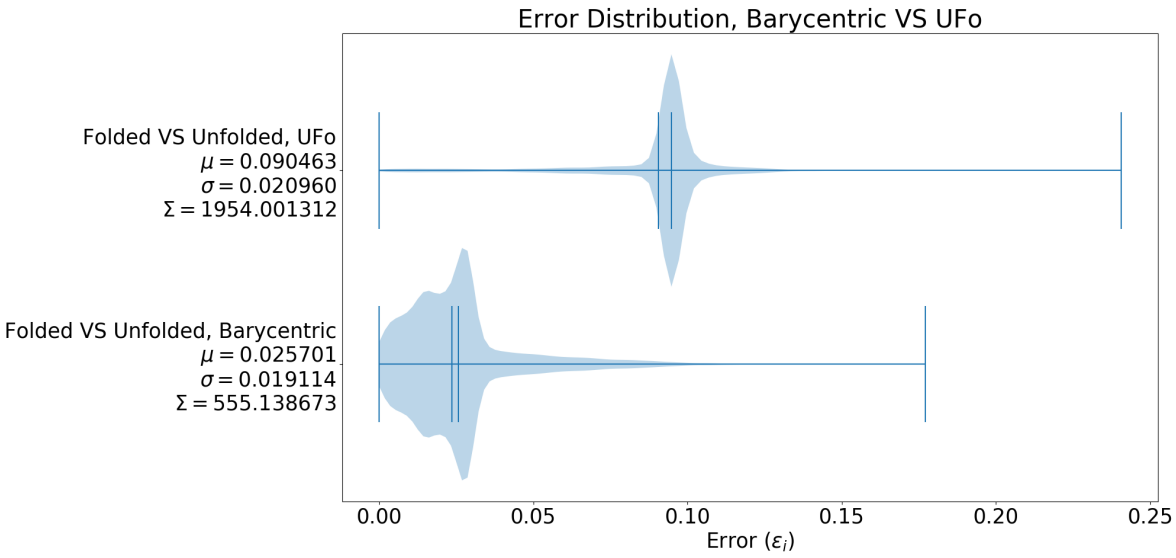


(C) Unfolded model, using barycentric algorithm, colored with NMD(Y) per point. Color bar ranges between 0.02 and 0.038 values.

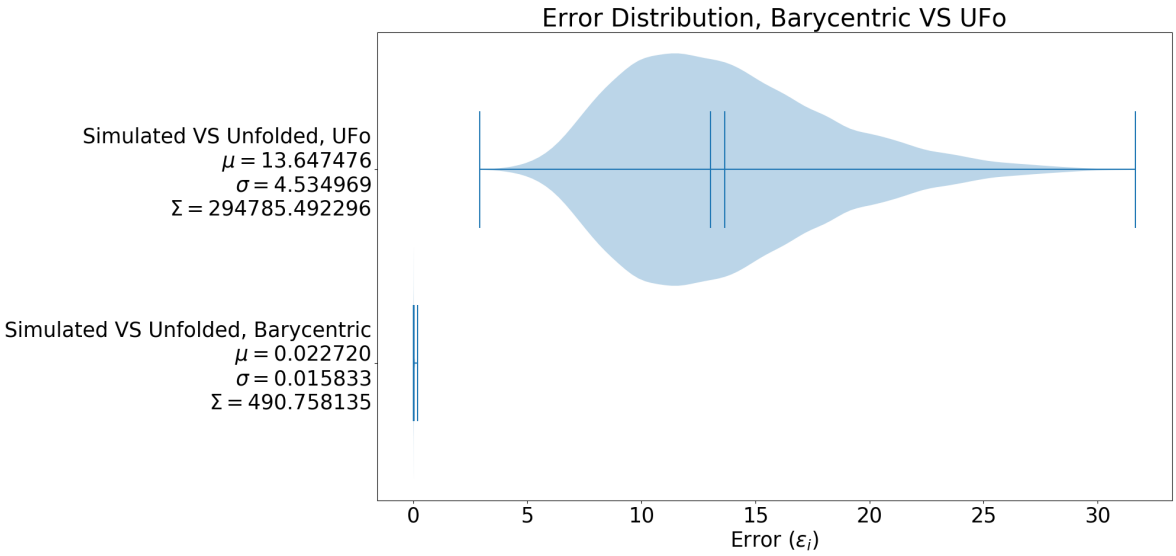


(D) Unfolded model, using UFo algorithm, colored with NMD(Y) per point. Color bar ranges between 0.103 and 0.755 values.

FIGURE 4.4: Point clouds depicting NMD(·) index in each step of unfolding flow for simulation case 1. Index is calculated using Equation 2.2. We define W as the original block model distribution, X as the folded model and Y as the unfolded model.

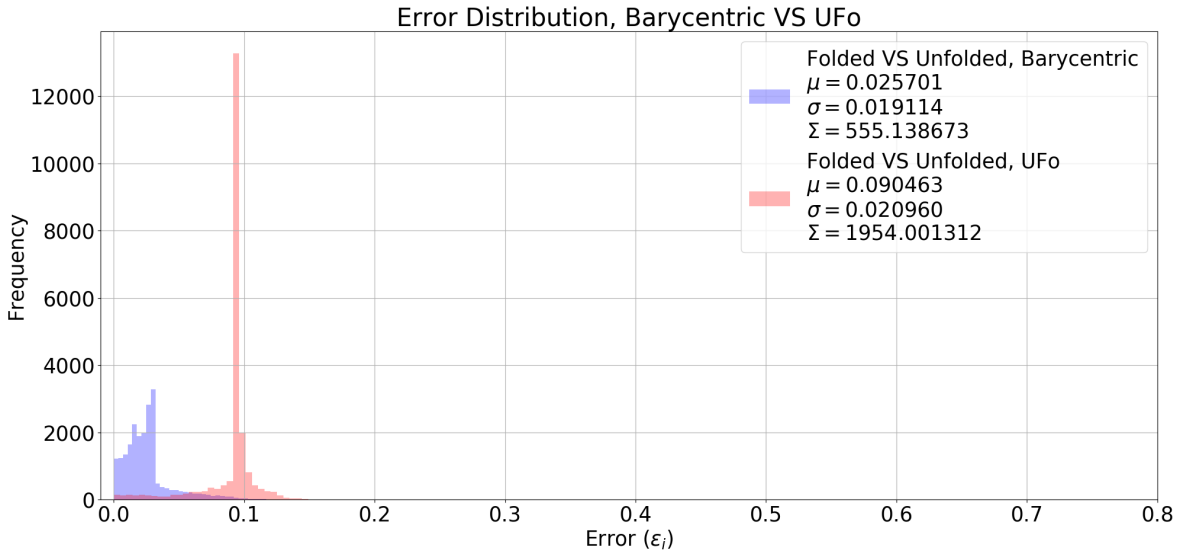


(A) $\epsilon(X, Y)$ violin plot for barycentric and UFo algorithms.

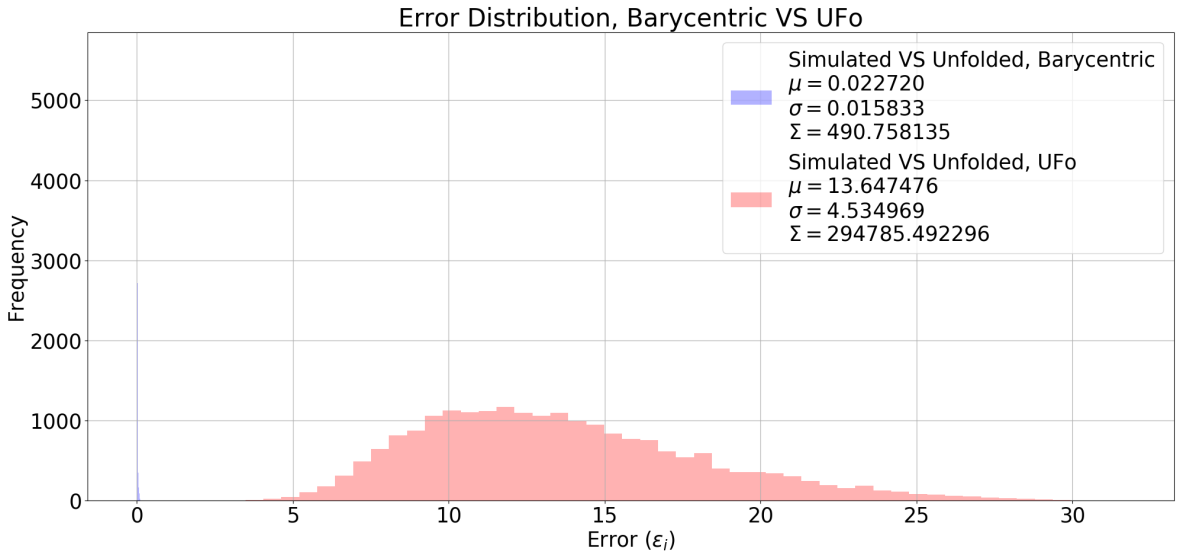


(B) $\epsilon(W, Y)$ violin plot for barycentric and UFo algorithms

FIGURE 4.5: $\epsilon(X, Y)$ and $\epsilon(W, Y)$ distribution comparison between barycentric and U-Fo unfolding algorithms, using violin plots for simulation case 1. Error distributions are calculated using Equation 2.4. We define W as the original block model distribution, X as the folded model and Y as the unfolded model, we obtained two error distributions: $\epsilon(W, Y)$ and $\epsilon(X, Y)$ for simulated.

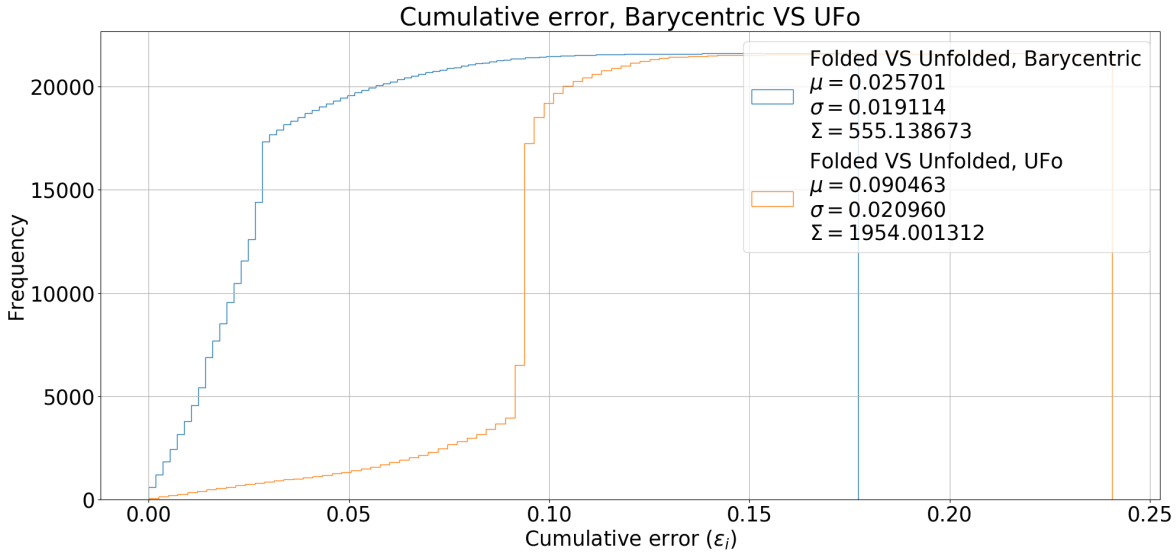


(A) $\epsilon(X, Y)$ histogram for barycentric and UFo algorithms

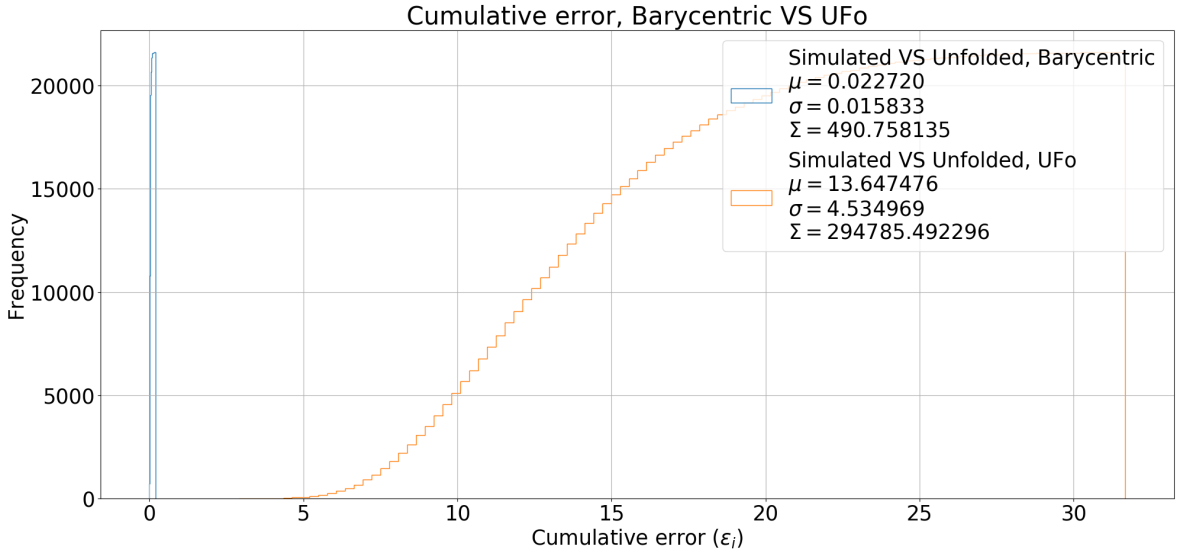


(B) $\epsilon(W, Y)$ histogram for barycentric and UFo algorithms

FIGURE 4.6: $\epsilon(X, Y)$ and $\epsilon(W, Y)$ distribution comparison between barycentric and U-Fo unfolding algorithms, using histogram plots for simulation case 1. Error distributions are calculated using Equation 2.4. We define W as the original block model distribution, X as the folded model and Y as the unfolded model, we obtained two error distributions: $\epsilon(W, Y)$ and $\epsilon(X, Y)$ for simulated data.



(A) $\epsilon(X, Y)$ cumulative plot for barycentric and U-Fo algorithms

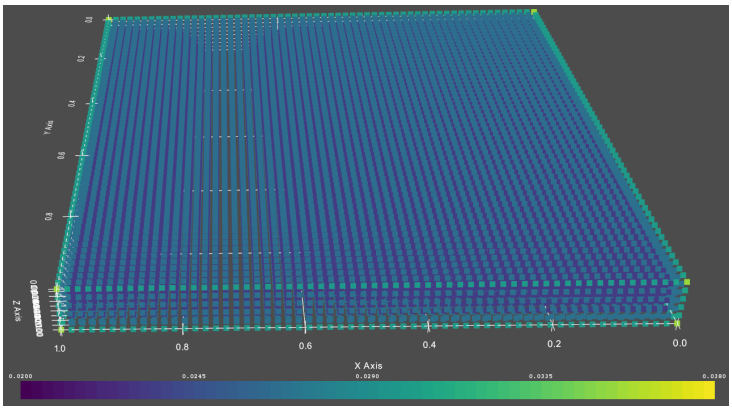


(B) $\epsilon(W, Y)$ cumulative plot for barycentric and U-Fo algorithms.

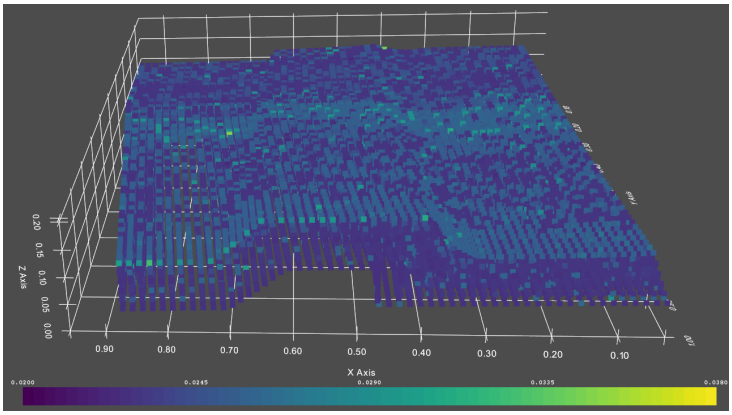
FIGURE 4.7: $\epsilon(X, Y)$ and $\epsilon(W, Y)$ distribution comparison between barycentric and U-Fo unfolding algorithms, using histogram and cumulative plots for simulation case 1. Error distributions are calculated using Equation 2.4. We define W as the original block model distribution, X as the folded model and Y as the unfolded model, we obtained two error distributions: $\epsilon(W, Y)$ and $\epsilon(X, Y)$ for simulated data.

4.2.2 Quality Results for simulation case 2

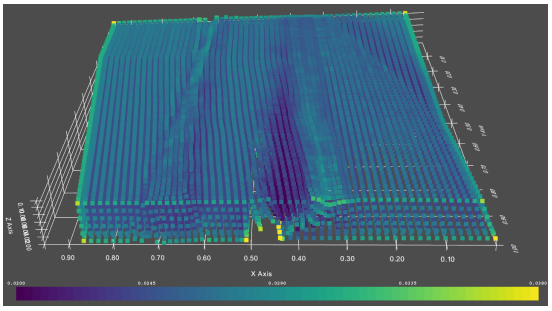
Simulation case 2 has the same start polyhedron than case 1, but it differ in the geological transformation used to obtain the folded model. Once again barycentric algorithm is a better approach when we compare NMD color distributions (Figure 4.8) and error distributions (Figures 4.9, 4.10 and 4.11). The results are similar to simulation case 1 but with higher variations between algorithms, for example, the distribution of $\epsilon(W, Y)$, we obtained $\mu = 0.032$, $\sigma = 0.029$ for barycentric algorithm and $\mu = 13.68$, $\sigma = 3.78$ for UFo.



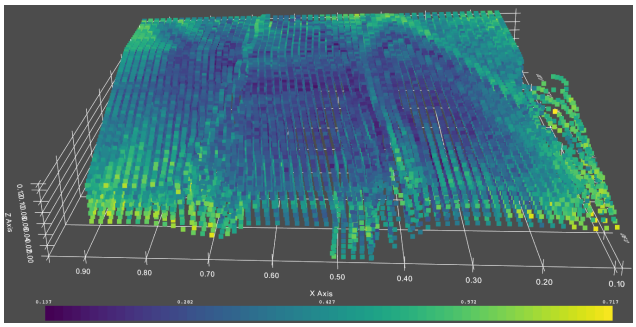
(A) Simulated model colored with NMD(W) per point. Color bar ranges between 0.02 and 0.038 values.



(B) Folded model colored with NMD(X) per point. Color bar ranges between 0.02 and 0.038 values.

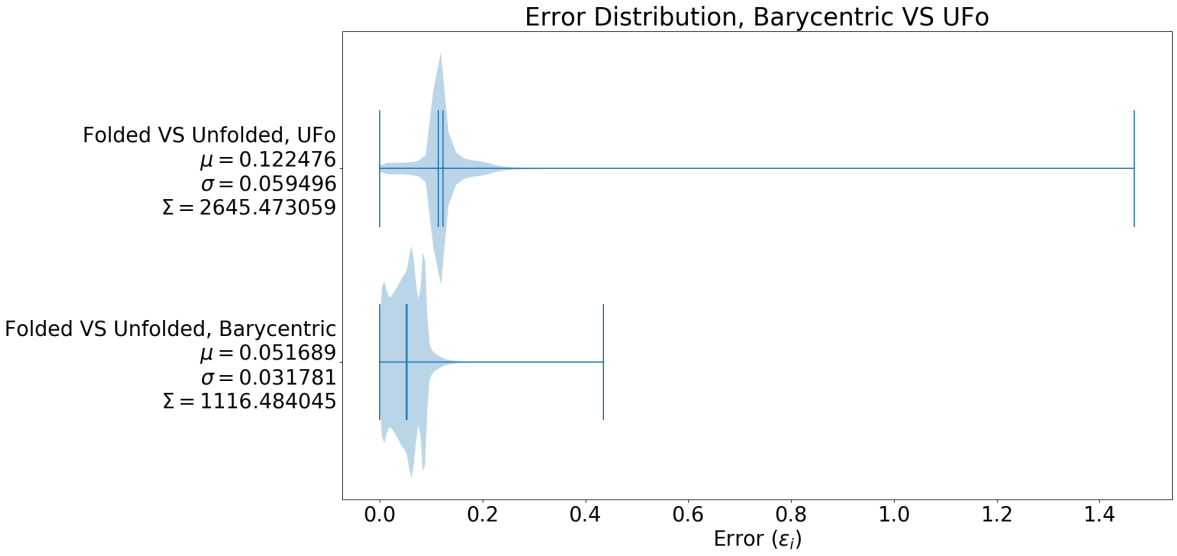


(C) Unfolded model, using barycentric algorithm, colored with NMD(Y) per point. Color bar ranges between 0.02 and 0.038 values.

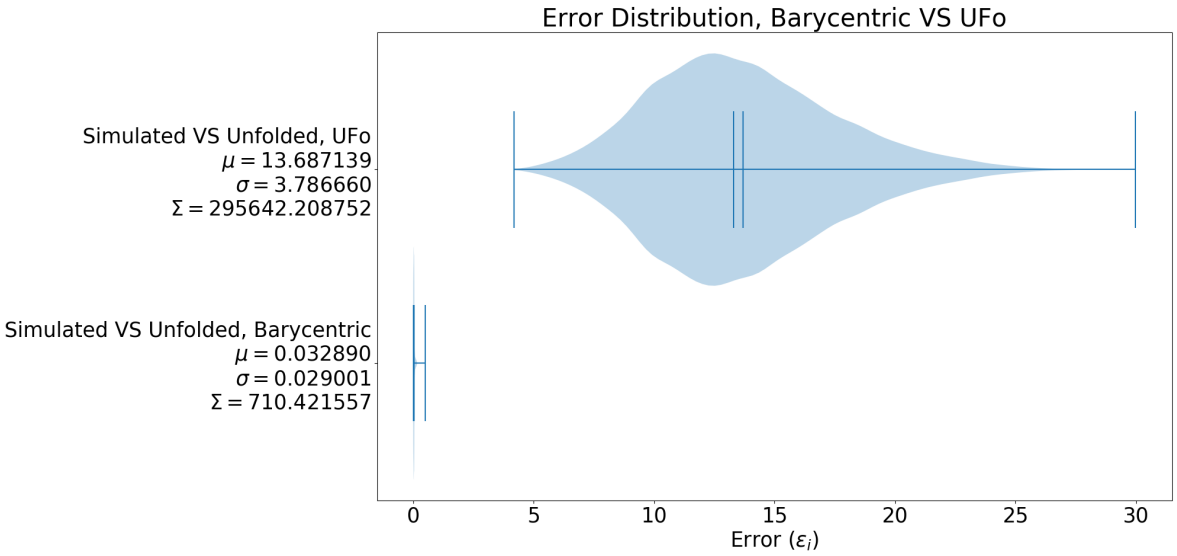


(D) Unfolded model, using UFo algorithm, colored with NMD(Y) per point. Color bar ranges between 0.137 and 0.717 values.

FIGURE 4.8: Point clouds depicting NMD(·) index in each step of unfolding flow for simulation case 2. Index is calculated using Equation 2.2. We define W as the original block model distribution, X as the folded model and Y as the unfolded model.

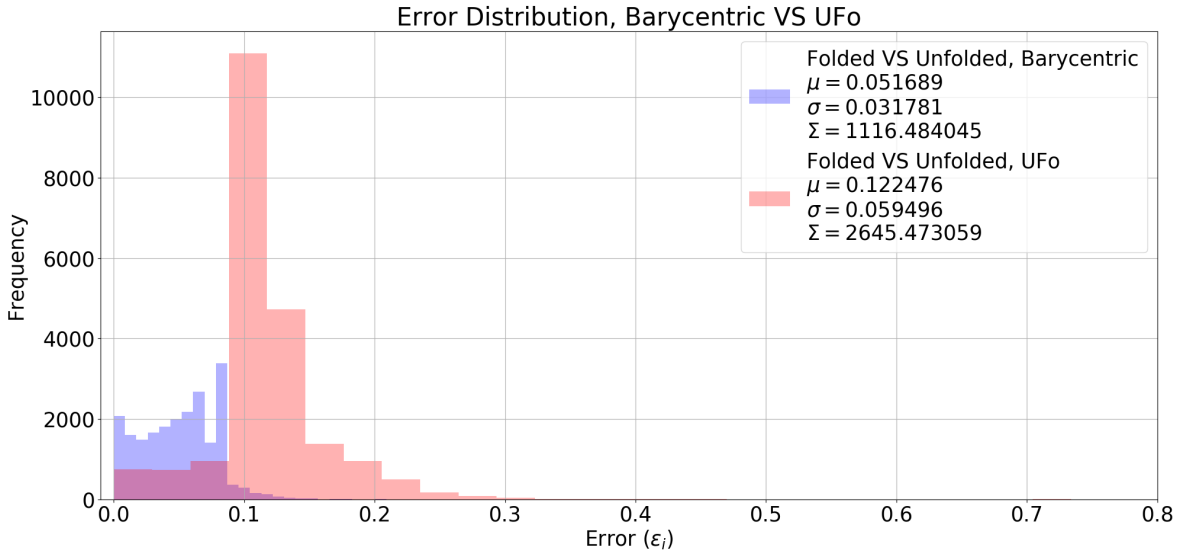


(A) $\epsilon(X, Y)$ violin plot for barycentric and UFo algorithms.

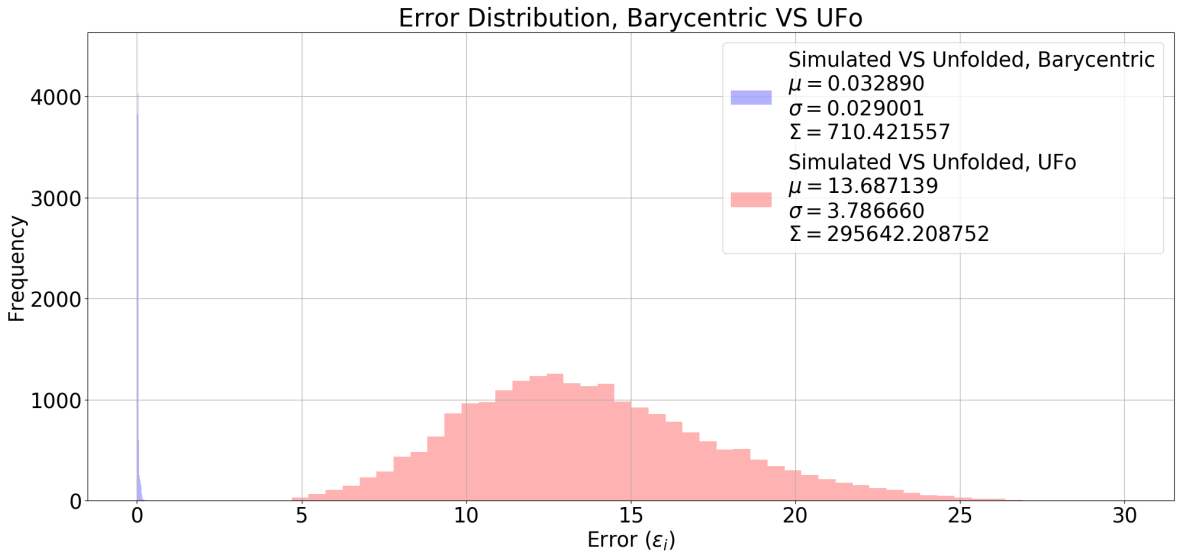


(B) $\epsilon(W, Y)$ violin plot for barycentric and UFo algorithms

FIGURE 4.9: $\epsilon(X, Y)$ and $\epsilon(W, Y)$ distributions comparison, using violin plots for simulation case 2. Error distributions are calculated using Equation 2.4. We define W as the original block model distribution, X as the folded model and Y as the unfolded model, we obtained two error distributions: $\epsilon(W, Y)$ and $\epsilon(X, Y)$ for simulated data.

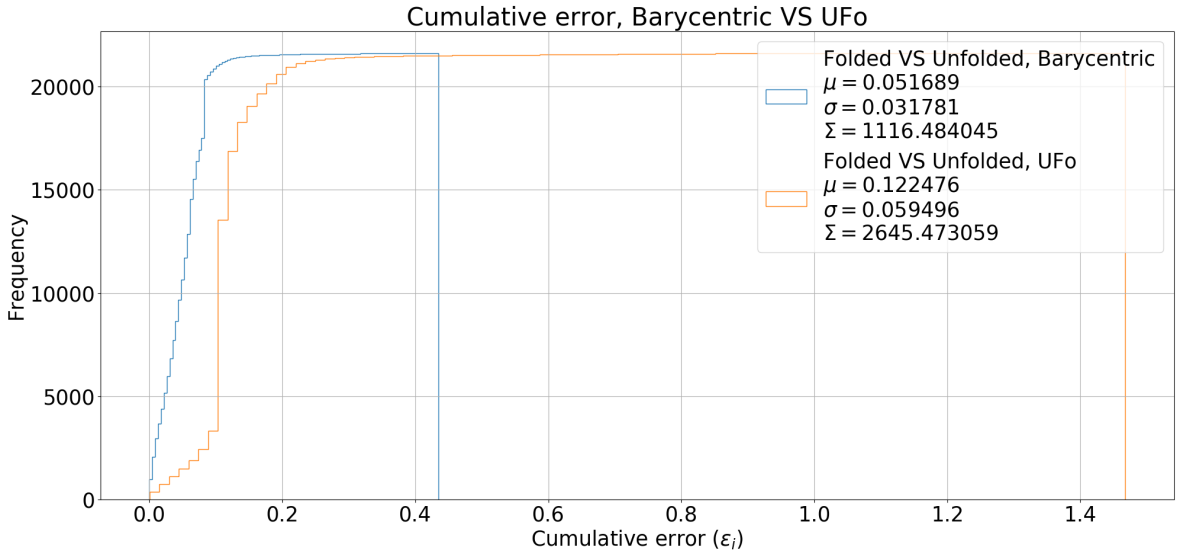


(A) $\epsilon(X, Y)$ histogram for barycentric and UFo algorithms

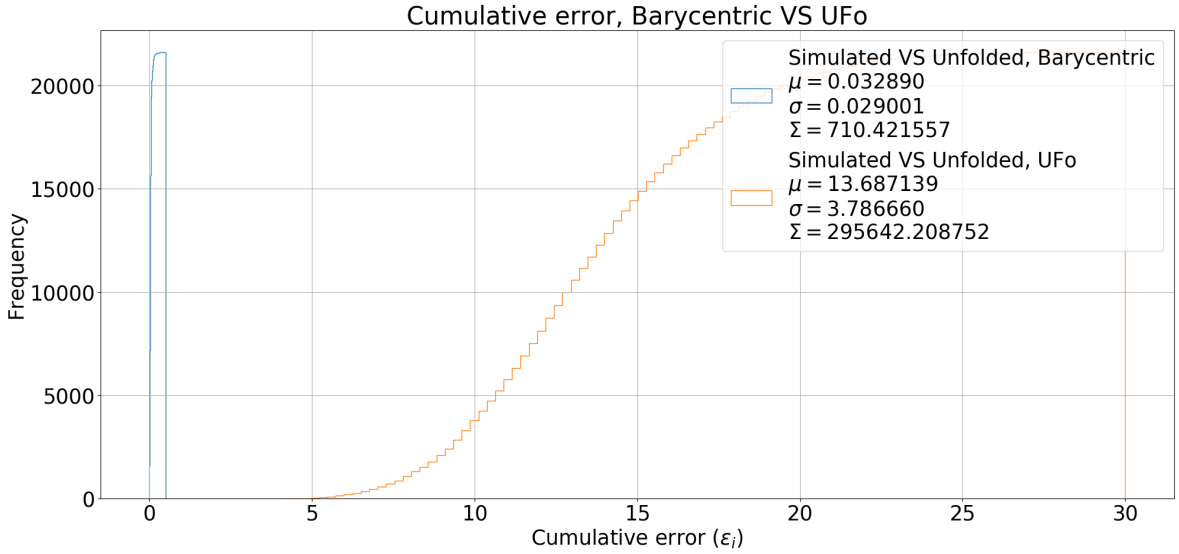


(B) $\epsilon(W, Y)$ histogram for barycentric and UFo algorithms

FIGURE 4.10: $\epsilon(X, Y)$ and $\epsilon(W, Y)$ distributions comparison, using histogram plots for simulation case 2. Error distributions are calculated using Equation 2.4. We define W as the original block model distribution, X as the folded model and Y as the unfolded model, we obtained two error distributions: $\epsilon(W, Y)$ and $\epsilon(X, Y)$ for simulated data.



(A) $\epsilon(X, Y)$ cumulative plot for barycentric and UFo algorithms



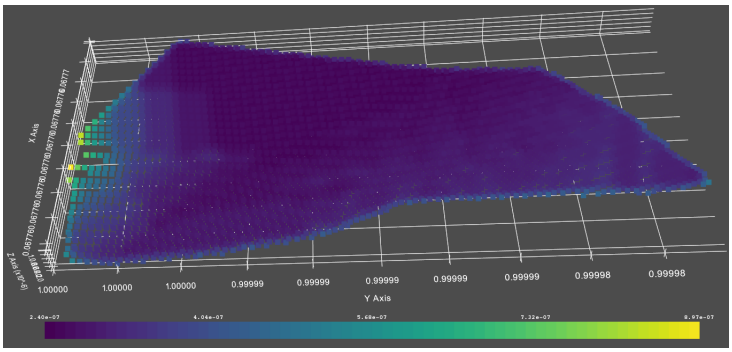
(B) $\epsilon(W, Y)$ cumulative plot for barycentric and UFo algorithms

FIGURE 4.11: $\epsilon(X, Y)$ and $\epsilon(W, Y)$ distributions comparison using cumulative plots for simulation case 2. Error distributions are calculated using Equation 2.4. We define W as the original block model distribution, X as the folded model and Y as the unfolded model, we obtained two error distributions: $\epsilon(W, Y)$ and $\epsilon(X, Y)$ for simulated data.

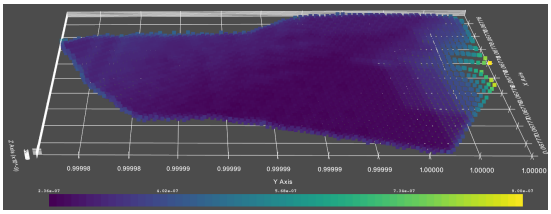
4.2.3 Quality Results for field case 1

Since field data is obtained from real geological measures, we do not have an original point cloud to compare with the unfolding model, and the comparison can only be made between folded and unfolded models.

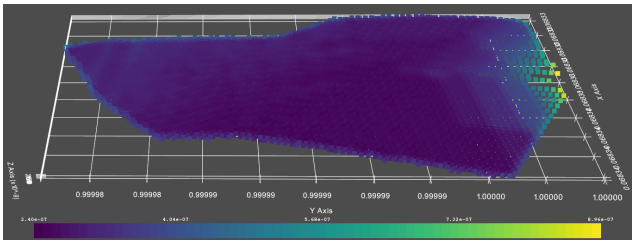
Figure 4.12 shows the distribution of NDM index in folded (NMD(X)) and unfolded (NMD(Y)) point clouds for both algorithms. In this case, we can not conclude whether the algorithm does better by just comparing the color distribution because both are similar and differences are small. By comparing error distribution in Figure 4.13 we got lower errors ($\epsilon(X, Y)$) using UFo approach with $\mu = 0.0021$, $\sigma = 0.0032$ versus $\mu = 0.0115$, $\sigma = 0.0095$ using barycentric algorithm. Nevertheless, for field data, we can not be certain of which algorithm is doing better because we can only compare NMD index in fold and unfold models, and by definition, if we do not unfold at all the folded model, $\epsilon(X, Y)$ distribution will be zero.



(A) Folded model colored with NMD(X) per point.Color bar ranges between 2.4E-07 and 8.97E-07 values.

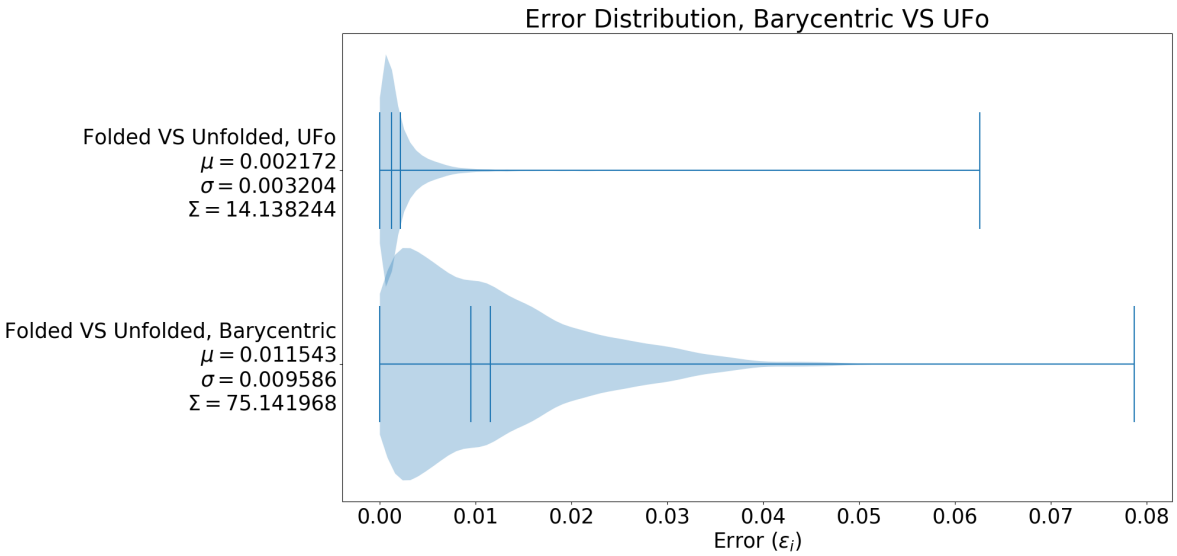


(B) Unfolded model, using barycentric algorithm, colored with NMD(Y) per point. Color bar ranges between 2.36E-07 and 9E-07 values.

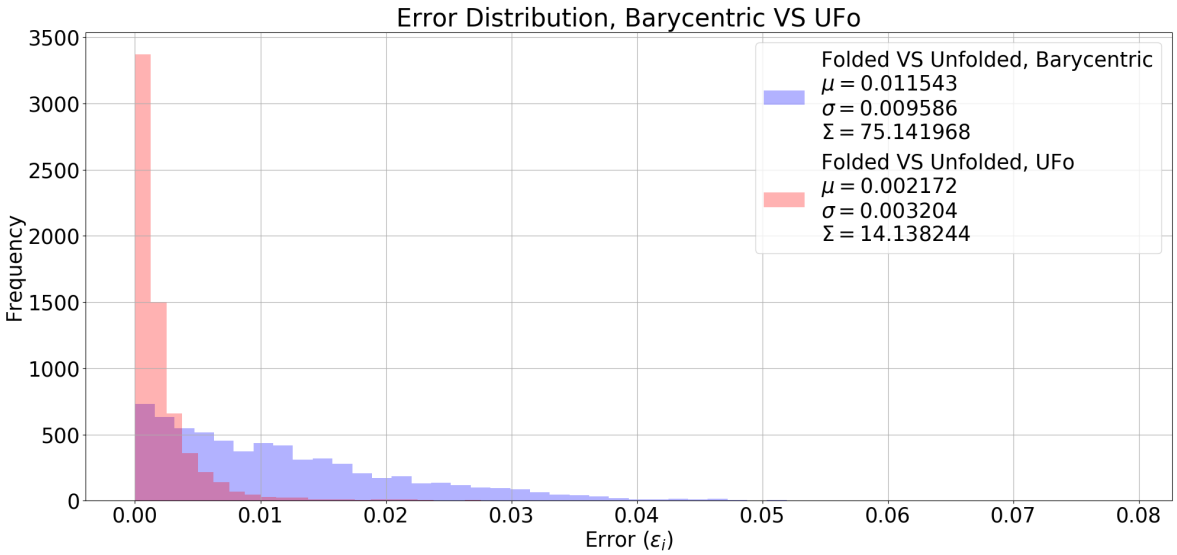


(C) Unfolded model, using UFo algorithm, colored with NMD(Y) per point. Color bar ranges between 2.4E-07 and 8.96E-07 values.

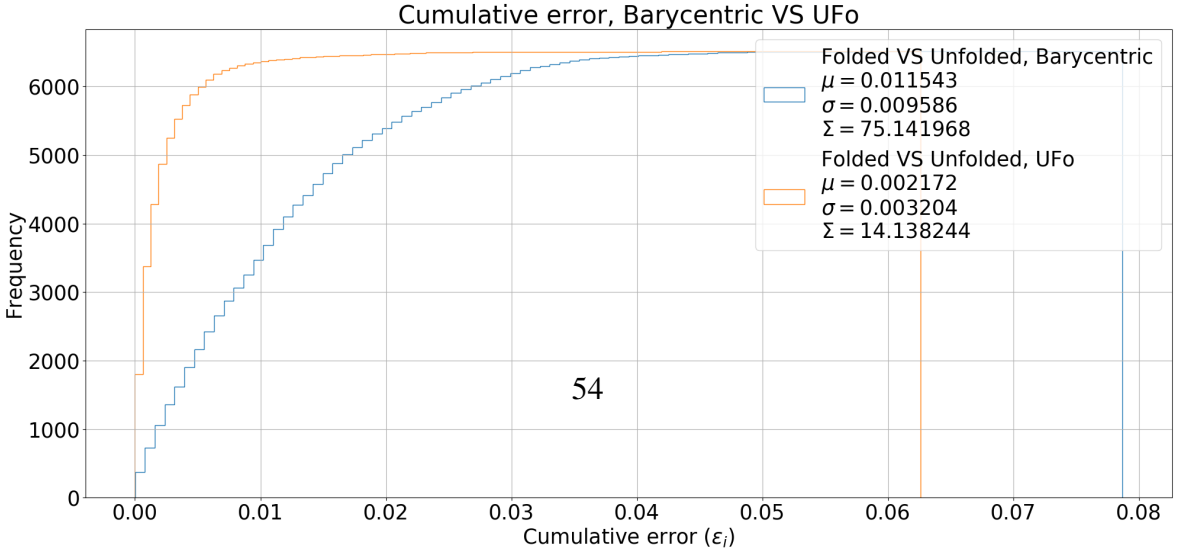
FIGURE 4.12: Point clouds depicting NMD(·) index in each step of unfolding flow for field case 1. Index is calculated using Equation 2.2. We define W as the original block model distribution, X as the folded model and Y as the unfolded model.



(A) $\epsilon(X, Y)$ violin plot for barycentric and UFo algorithms.



(B) $\epsilon(X, Y)$ histogram for barycentric and UFo algorithms.



4.2.4 Quality Analysis

We show a summarize of the mean error per geological model in Table 4.4. Results show that the barycentric unfolding algorithm has better quality than U-Fo in simulation cases. For the field case, U-Fo has a lower $\mu(\epsilon(X,Y))$ indicating that folded and unfolded distributions are more similar for U-Fo unfolding than for barycentric unfolding. Nevertheless, as explained in previous sections, for field cases we do not have an original restoration model, therefore, we do not have an objective distribution to follow.

A guess to explain why barycentric algorithm perform better in quality than U-Fo unfolding is that the mathematical construct of barycentric transformations (Equation 2.1) conserves reference distances inside a geometrical mesh, therefore, a quality index that measure geometrical properties will have better results in barycentric unfolding. If the previous guess is true, we can expect an improvement if we change barycentric transformations with a more sophisticated variant, like Wachpress coordinates [18].

Mean Error Comparison					
geological models	points	$\mu(\epsilon(W,Y))$		$\mu(\epsilon(X,Y))$	
		U-Fo	Barycentric	U-Fo	Barycentric
sim_case_1	21600	13.6475	0.0227	0.0905	0.0257
sim_case_2	21600	13.5871	0.0329	0.1225	0.0595
field_case_1	6510	N/A	N/A	0.0022	0.0115

TABLE 4.4: Mean error comparison for each geological model. We define W as the original model distribution, X as the unfolded model and Y as the folded model. The Error index distribution ($\epsilon(\cdot,\cdot)$) is calculated using Equation 2.4 and the function $\mu(\cdot)$ symbolize the statistical mean.

Chapter 5

Conclusions

5.1 Conclusions

In geological restoration, unfolding a mineral solid body is a crucial and not resolved task for ore estimation process. Currently working solutions involves expert knowledge and a priori information of physics and rock properties.

U-Fo was created, in this context, with the objective of being a semi-automated algorithm that aids geologist in the unfolding and unfualted pipeline, using geometry and geostatistics tools.

In this Thesis we designed, implemented and analyzed a new unfolding algorithm. The new algorithm improves UFo approach in: 1) **Automated extraction of top and bottom surfaces from point cloud**; 2) **Surface flattened of top and bottom using energy release optimization algorithm**; 3) **Barycentric deformation of solid body using two referential surfaces and tetrahedral mesh**. The algorithm complexity is $O(N^2)$ (where N is the number of input points) and we compared the unfolding results in performance and quality (using NMD index) with UFo program.

In performance, the barycentric approach seems to be slower than UFo approach mainly because of the use of 2N points when using the top and bottom surfaces and the tetrahedral mesh creation step, while in quality comparison barycentric algorithm had a lower error ($\epsilon(W, Y)$) in simulated dataset and similar results in the field block model. the contributions, of this Thesis, can be summarized as follows:

1. **Automatic surfaces extraction algorithm from point cloud based in alpha-shape algorithm**. In comparison, U-Fo unfolding expects a reference surface as an input, therefore, this implementation is an enhancement and might be helpful as a guide to geologist experts.
2. **Slightly improvement of performance in surface flattening algorithm**. As shown in section 4.1 and Figure 4.3, surface flattening step is faster (per point in surface) in barycentric unfolding than U-Fo.
3. **New barycentric based unfolding algorithm**. Which according to results presented in 4.2 does a better job, than U-Fo, preserving geometrical features in simulated geological models.
4. **Methodology to measure quality in unfolding algorithms**. Based on NMD index and error distribution comparison.
5. **Geometry visualization code based in pyVista**.Written to visualize point cloud datasets and results over the unfolding pipeline.

5.2 Future Work

As future work, several techniques/algorithms could be adapted in order to test if they improve the proposed unfolding solution:

- Use 3D generalized barycentric coordinates (like Wachpress coordinates [18]) may improve the deformation process and minimize pathological cases of points in surfaces.
- Design and test other mesh creation and refine algorithms, such as the ones used in popular mesh libraries like Tetgen [24].

- The design and implementation of parallel (GPU) versions of deformation and mesh creation algorithm should improve algorithm performance.
- Comparison of field dataset unfolding results with real ore extraction data to improve quality testing.

Bibliography

[1] Pierre Alliez, Laurent Saboret, and Gaël Guennebaud. “Poisson Surface Reconstruction”. In: *CGAL User and Reference Manual*. 4.12. CGAL Editorial Board, 2018. URL: https://doc.cgal.org/4.12/Manual/packages.html{\#}PkgPoissonSurface

[2] Chakib Bennis, Houman Borouchaki, Cyrielle Dumont, Olivier Lerat, Mathieu Poudret, and Jean François Rainaud. “3D line-support grid flattening for more accurate geostatistical reservoir population with petrophysical properties”. In: *Engineering with Computers*. Vol. 30. 3. 2014, pp. 403–421. DOI: 10.1007/s00366-012-0311-9.

[3] Tran Kai Frank Da and David Cohen-Steiner. “Advancing Front Surface Reconstruction”. In: *CGAL User and Reference Manual*. 4.12. CGAL Editorial Board, 2017. URL: http://doc.cgal.org/4.11/Manual/packages.html{\#}PkgAdvancingFrontSurfac

[4] Tran Kai Frank Da, Sébastien Loriot, and Mariette Yvinec. “3D Alpha Shapes”. In: *CGAL User and Reference Manual*. 4.14. CGAL Editorial Board, 2019. URL: https://doc.cgal.org/4.14/Manual/packages.html{\#}PkgAlphaShapes3.

[5] Mark De Berg, Otfried Cheong, Marc Van Kreveld, and Mark Overmars. *Computational geometry: Algorithms and applications*. 2008, pp. 1–386. ISBN: 9783540779735. DOI: 10.1007/978-3-540-77974-2.

[6] Pascal Jean Frey and Paul-Louis George. *Mesh Generation - Application to Finite Elements*. 2007. ISBN: 1903398002. DOI: 10.1046/j.1365-232x.2001.0190d.x.

URL: http://www.blackwell-synergy.com/links/doi/10.1046{\%}2Fj.1365-232X.2001.0190d.x.

[7] R. L. G., R. Deaux, and H. Eves. *Introduction to the Geometry of Complex Numbers*. Vol. 42. 341. 1958, p. 245. ISBN: 9780486466293. DOI: 10.2307/3610417.

[8] Mauricio Garrido, Felipe Navarro, Julián Ortiz, and Josemar Moreira. "Resource assessment with unfolding methodology : a case study". In: *Minin Conference*. 2016.

[9] David Goldberg. "What every computer scientist should know about floating-point arithmetic". In: *ACM Computing Surveys (CSUR)* 23.1 (1991), pp. 5–48. ISSN: 15577341. DOI: 10.1145/103162.103163.

[10] J P Gratier, B Guillier, A Delorme, and F Odonne. "Restoration and balanced cross section of a folded and faulted surface by computer program: principle and application". In: *Journal of structural geology* 13 (1991), pp. 111–115.

[11] S. Horna, C. Bennis, H. Borouchaki, C. Delage, and J.F. Rainaud. "Isometric Unfolding of Stratigraphic Grid Units for Accurate Property Populating - Mathematical Concepts". In: *Ecmor Xii*. September 2010. 2010, p. 12. DOI: B005.

[12] Jin Huang, Lu Chen, Xinguo Liu, and Hujun Bao. "Efficient mesh deformation using tetrahedron control mesh". In: *Computer Aided Geometric Design* 26.6 (2009), pp. 617–626. ISSN: 01678396. DOI: 10.1016/j.cagd.2008.12.002.

[13] J. Mark Keil and Carl A. Gutwin. "Classes of graphs which approximate the complete euclidean graph". In: *Discrete & Computational Geometry* 7.1 (1992), pp. 13–28. ISSN: 01795376. DOI: 10.1007/BF02187821.

[14] Wolfgang Kuhnel. *Differential geometry : curves - surfaces - manifolds*. 2002, p. 358. ISBN: 9780821826560.

[15] Thijs van Lankveld. "Scale-Space Surface Reconstruction ". In: *CGAL User and Reference Manual*. 4.12. CGAL Editorial Board, 2018.

[16] Laurent Maerten and Frantz Maerten. “Chronologic modeling of faulted and fractured reservoirs using geomechanically based restoration: Technique and industry applications”. In: *AAPG Bulletin* 90.8 (2006), pp. 1201–1226. ISSN: 01491423. DOI: 10.1306/02240605116.

[17] Rafael Martí, Jose A Lozano, and Alexander Mendiburu. *Handbook of Heuristics*. 2016. ISBN: 9783319071534. DOI: 10.1007/978-3-319-07153-4.

[18] M.S. Floater. “Generalized barycentric coordinates and applications”. In: *Acta Numerica* 24.April (2015), pp. 161–214. ISSN: 0962-4929. DOI: 10.1017/S09624929. arXiv: arXiv:1711.05337v1.

[19] Felipe Navarro, Daniel Baeza, Dafne Herreros, and Marcos Valencia. “Calculating ore resources on complex geology using a geometric restitution methodology: from modeling to the estimation”. In: *Conference of underground minning*. 2016.

[20] Felipe Navarro, Mauricio Garrido, Carlos González, and Daniel Baeza. “U-Fo : An innovative successful case of geological resource assessment on gold deposits under complex geometries”. In: *Minexcellence*. 2016.

[21] Marcia Ojeda. “Metodología de Validación de Técnicas de Restauración de Cuerpos Geológicos en 3D”. MA thesis. 2019.

[22] Mario E. Rossi and Clayton V. Deutsch. *Mineral resource estimation*. 2014, pp. 1–332. ISBN: 9781402057175. DOI: 10.1007/978-1-4020-5717-5.

[23] Delphine Rouby, Hongbin Xiao, and John Suppe. “3-D restoration of complexly folded and faulted surfaces using multiple unfolding mechanisms”. In: *AAPG Bulletin* 84.6 (2000), pp. 805–829. ISSN: 01491423. DOI: 10.1306/A9673400-1738-11D7-8645000102C1865D

[24] Hang Si. “TetGen, a Delaunay-Based Quality Tetrahedral Mesh Generator”. In: *ACM Trans. Math. Softw.* 41.2 (2015), 11:1–11:36. ISSN: 0098-3500. DOI: 10.1145/2629697. URL: http://doi.acm.org/10.1145/2629697.

[25] Peipei Tang, Chengjing Wang, and Xiaoxia Dai. “A majorized Newton-CG augmented Lagrangian-based finite element method for 3D restoration of geological models”. In: *Computers and Geosciences* 89 (2016), pp. 200–206. ISSN: 00983004. DOI: 10.1016/j.cageo.2016.01.013. URL: http://dx.doi.org/10.1016/j.cageo.2016.01.013.

[26] Andrés Vicencio, Marcia Ojeda, Daniel Baeza, and Felipe Navarro. “The impact of geometric variation on the resource estimation quality using an unfolding methodology”. In: *Geomin Mineplanning* (2017), pp. 1–10.

[27] Charlie C.L. Wang, Shana S.F. Smith, and Matthew M.F. Yuen. “Surface flattening based on energy model”. In: *CAD Computer Aided Design* 34.11 (2002), pp. 823–833. ISSN: 00104485. DOI: 10.1016/S0010-4485(01)00150-6.

[28] G. D. Williams, S. J. Kane, T. S. Buddin, and A. J. Richards. “Restoration and balance of complex folded and faulted rock volumes: Flexural flattening, jigsaw fitting and decompaction in three dimensions”. In: *Tectonophysics* 273.3-4 (1997), pp. 203–218. ISSN: 00401951. DOI: 10.1016/S0040-1951(96)00282-X.

[29] Eric R. Ziegel, E. L. Lehmann, and George Casella. *Theory of Point Estimation*. Vol. 41. 3. 1999, p. 274. ISBN: 0387985026. DOI: 10.2307/1270597. URL: http://www.amazon.com/dp/0387985026.

[30] Gil Zigelman, Ron Kimmel, Nahum Kiryati, and Senior Member. “Texture Mapping Using Surface Flattening via Multidimensional Scaling”. In: *Visualization and Computer Graphics, IEEE Transactions on* 8.2 (2002), pp. 198–207.

Appendix A

Source Code

A.1 Python Source Code

LISTING A.1: unfold_pc.py

```
from data_utils import *
from plot_utils import *

from geometry_algorithms import divide_volume_surface
from geometry_algorithms import unfold_surface
from geometry_algorithms import join_surfaces
from geometry_algorithms import create_tetra_mesh
from geometry_algorithms import search_points_inside_tetra_mesh
from geometry_algorithms import apply_barycentric_deformation
from geometry_algorithms import calculate_quality_measures
from geometry_algorithms import generate_concave_hull
from geometry_algorithms import rotate_point_cloud

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import argparse
import time
import matplotlib
import re

from pathlib import Path


if __name__ == '__main__':
    matplotlib.rcParams.update({'font.size': 22})
    parser = argparse.ArgumentParser()
    parser.add_argument('-p', '--point_cloud', default='datasets/folded_caso82.csv')
    parser.add_argument('-n', '--n_sim', default='1')
    parser.add_argument('-s', '--ref_surface', default=None)
    args = parser.parse_args()
    sim_results = []
    n_sim = int(args.n_sim)
    bm_file = args.point_cloud

    last_verts = None
    last_flt_verts = None

    for i in range(n_sim):
        start_t = time.time()
        verts_sc = read_block_model(bm_file)
        verts_sc = rotate_point_cloud(verts_sc)
        preprocess_t = time.time()

        # Plot with PCA axis
```

```
        # z_scalars = [p[2] for p in verts_sc]
        # arrow_pos = np.mean(verts_sc, 0)
        # arrow_pos[2] = np.max(z_scalars)
        # plot_scatter(verts_sc, scalars=z_scalars, opacity=0.5, arrows=(arrow_pos, np.array([0, 0, 1])*0.09))
        # plot_scatter(verts_sc, scalars=z_scalars, opacity=0.5)

        if args.ref_surface is None:

            chull_points, chull_faces = generate_concave_hull(verts_sc, bm_file)
            # plot_surface(chull_points, chull_faces)
            (top_points, top_faces), (bot_points, bot_faces) = divide_volume_surface(chull_points, chull_faces)
            concave_t = time.time()
            plot_two_surfaces(top_points, top_faces, bot_points, bot_faces)
            # plot_two_surfaces(top_points, top_faces, bot_points, bot_faces, all_points=verts_sc)
        else:
            top_points, top_faces = read_surface(args.ref_surface + '_top.csv')
            bot_points, bot_faces = read_surface(args.ref_surface + '_bot.csv')
            # Translate in z
            top_points[:, 2] = top_points[:, 2] + (verts_sc[:, 2].max() - top_points[:, 2].min())
            bot_points[:, 2] = bot_points[:, 2] + (verts_sc[:, 2].min() - bot_points[:, 2].max())

             # Translate centroid
            trans_vec =  verts_sc[:, 0:2].mean(axis=0) - top_points[:, 0:2].mean(axis=0)
            top_points[:, 0:2] = top_points[:, 0:2] + trans_vec

            trans_vec = verts_sc[:, 0:2].mean(axis=0) - bot_points[:, 0:2].mean(axis=0)
            bot_points[:, 0:2] = bot_points[:, 0:2] + trans_vec
            concave_t = time.time()

            # plot_two_surfaces(top_points, top_faces, bot_points, bot_faces, all_points=verts_sc)

        delta_t = 0.01
        top_flt, mse_top, areas_top = unfold_surface(top_points, top_faces, step=delta_t, iterations=-1, use_ufo=False)
        bot_flt, mse_bot, areas_bot = unfold_surface(bot_points, bot_faces, step=delta_t, iterations=-1, use_ufo=False)

        # plot_mse(mse_top, areas_top/calculate_total_area(top_points, top_faces), title='Error Trough Iterations For Top Surface')
        # diff_area = unfold_diff_areas(top_points[:, :2], top_faces, top_flt, top_faces)
        # plot_triangulation(top_points, top_faces, scalars=diff_area, title='Area Comparison For Top Surface')
        # plot_surface(top_flt, top_faces)

        # plot_mse(mse_bot, areas_bot/calculate_total_area(bot_points, bot_faces), title='Error Trough Iterations For Bottom Surface')
        # diff_area = unfold_diff_areas(bot_points[:, :2], bot_faces, bot_flt, bot_faces)
        # plot_triangulation(bot_points, bot_faces, scalars=diff_area, title='Area Comparison For Bottom Surface')
        # plot_surface(bot_flt, bot_faces)
        unfold_t = time.time()

        up_points, up_faces, up_flt = join_surfaces(top_points, top_faces, bot_points, bot_faces, top_flt)
        # plot_two_surfaces(up_points, up_faces, bot_points, bot_faces, all_points=verts_sc)

        tetra_points, tetra_simplices, tetra_points_flt = create_tetra_mesh(up_points, up_faces, bot_points, bot_faces, up_flt, bot_flt)
        plot_tetras(tetra_points, tetra_simplices)
        # plot_tetras(tetra_points, tetra_simplices, export_file='tetra_mesh.vtk')
        tetra_indices, inside_tetras = search_points_inside_tetra_mesh(verts_sc, tetra_points,tetra_simplices, up_points, up_faces)
        mesh_t = time.time()

        verts_sc_flt = apply_barycentric_deformation(verts_sc, tetra_points, tetra_simplices, tetra_points_flt, tetra_indices)
        bary_t = time.time()
        end_t = time.time()

        initial_index, final_index, diff_index = calculate_quality_measures(verts_sc, verts_sc_flt)

        all_index = np.array([initial_index, final_index])
        min_in, max_in = all_index.min(), all_index.max()
        # plot_scatter(verts_sc_flt, np.array(inside_tetras, dtype='int'))
        # plot_scatter(verts_sc_flt, initial_index)
        plot_scatter(verts_sc_flt, diff_index)
        # plot_hist(diff_index)
        # plot_cumulative(diff_index)
        times = [preprocess_t - start_t,
                 concave_t - preprocess_t,
                 unfold_t - concave_t,
                 mesh_t - unfold_t,
                 bary_t - mesh_t,
                 end_t - start_t]

        sim_results.append(times)
        last_verts = verts_sc
        last_flt_verts = verts_sc_flt

columns = ['preprocess_time',
           'sur_ext_time',
           'sur_flat_time',
           'mesh_cons_time',
           'bary_deformation_time',
           'total_time']
```

```
    sim_results = np.array(sim_results)
    name = re.search('/(.*).csv', bm_file)[1]
    sim_times_df = pd.DataFrame(sim_results, columns=columns)
    out_name = 'results/' + name +'_times.csv'

    csv_file = Path(out_name)
    if csv_file.is_file():
        with open(out_name, 'a') as f:
            sim_times_df.to_csv(f, header=False)
    else:
        sim_times_df.to_csv(out_name)

    columns = ['x', 'y', 'z']
    verts_name = 'results/' + name + '_folded.csv'
    verts_flt_name = 'results/' + name + '_unfolded.csv'
    verts_df = pd.DataFrame(last_verts, columns=columns)
    verts_flt_df = pd.DataFrame(last_flt_verts, columns=columns)
    verts_df.to_csv(verts_name)
    verts_flt_df.to_csv(verts_flt_name)
```

LISTING A.2: data_utils.py

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
from sklearn.neighbors import kneighbors_graph


def mse(vec1, vec2):
    return (np.square(vec1 - vec2)).mean()


def neighbor_measure(points):
    graph = kneighbors_graph(points, 26, mode='connectivity', include_self=False)
    neigh_list = []
    for p_index, row in enumerate(graph):
        nei_indexes = np.where(row.T.todense() == 1.0)[0]
#         print(points[nei_indexes].shape)
#         print(points[p_index].shape)
#         print((points[nei_indexes] - points[p_index]).shape)
        neigh_prom = np.linalg.norm(points[nei_indexes] - points[p_index], axis=1).sum()/26.0
#         print(neigh_prom)
        neigh_list.append(neigh_prom)
    return np.array(neigh_list)


def create_sub_structures(points, triangles):
    all_points = points.copy()
    all_triangles = triangles.copy()

    flat_triangles = np.unique(all_triangles.flatten()).tolist()

    sub_points = np.empty((len(flat_triangles), all_points.shape[1]))

    points_map = {}
    for i, val in enumerate(flat_triangles):
        sub_points[i] = all_points[flat_triangles[i]]
        points_map.update({flat_triangles[i]: i})

    sub_triangles = np.empty(all_triangles.shape)
    for i, tr in enumerate(all_triangles):
        sub_triangles[i] = np.array([points_map[val] for val in tr])

    return sub_points, sub_triangles.astype(int), points_map


def read_off(file_name):
    f = open(file_name, 'r')
    # First line is just a OFF string
    f.readline()
    # Second line has points and faces
    n_points, n_faces, dummy= [int(v) for v in f.readline().split("␣")]
    # print(n_points, n_faces)
    # Parse points in surface
    points = []
    for i in range(n_points):
        points.append([float(v) for v in f.readline().split("␣")])
    # Parse faces in surface
    faces = []
    for j in range(n_faces):
        faces.append([int(v) for v in f.readline().split("␣")][1:4])
    f.close()
    return np.array(points), np.array(faces)
```

```
def read_block_model(file_name, scale=True):
    bm_df = pd.read_csv(file_name)
    if bm_df.columns.contains('centroid_x'):
        bm_verts = bm_df[["centroid_x", "centroid_y", "centroid_z"]].values
    else:
        bm_verts = bm_df[["x", "y", "z"]].values

    if(scale):
        scaler = MinMaxScaler(feature_range=(0, 1))
        bm_verts_sc = scaler.fit_transform(bm_verts.reshape(bm_verts.size, 1)).reshape(bm_verts.shape)
    else:
        bm_verts_sc = bm_verts
    return bm_verts_sc


def read_surface(file_name, scale=True):
    surf_df = pd.read_csv(file_name)
    rs_verts = surf_df[["x", "y", "z"]].dropna().values
    rs_faces = surf_df[['v1', 'v2', 'v3']].dropna().values
    scaler = MinMaxScaler(feature_range=(0, 1))
    if scale:
        rs_verts = scaler.fit_transform(rs_verts.reshape(rs_verts.size, 1)).reshape(rs_verts.shape)

    return rs_verts, rs_faces


def write_off(verts, file_name):
    # export points
    f = open(file_name, "w")

    n_points = verts.shape[0]
    first_line = "%d\n" % (n_points)
    f.write(first_line)

    for i in range(n_points):
        line = "%.20f %.20f %.20f\n" % (verts[i][0], verts[i][1], verts[i][2])
        f.write(line)
    f.close()


def scale(data_vector):
    scaler = MinMaxScaler(feature_range=(0, 1))
    return scaler.fit_transform(data_vector.reshape((-1, 1)))
```

LISTING A.3: energy_model.py

```
import numpy as np
from data_utils import mse
from geometry_utils import *
# import warnings


def plot_mesh(verts, faces):
    x = verts[:, 0].tolist()
    y = verts[:, 1].tolist()
    if verts.shape[1] > 2:
        z = verts[:, 2].tolist()
    else:
        z = np.zeros(verts.shape[0]).tolist()

    xe = np.array([[verts[pt][0].tolist() for pt in tr] for tr in faces]).tolist()
    # x_test = np.array([[pt for pt in tr] for tr in faces]).flatten().tolist()

    ye = np.array([[verts[pt][1].tolist() for pt in tr] for tr in faces]).tolist()

    if verts.shape[1] > 2:
        ze = np.array([[verts[pt][2].tolist() for pt in tr] for tr in faces]).tolist()
    else:
        ze = np.array([[0.0 for pt in tr] for tr in faces]).tolist()
    return


def vertex_mass(v_index, graph_tr, faces, verts):
    sum_area = 0.0
    for f_ind in graph_tr[v_index]:
        a = np.transpose(np.array([verts[v_ind] for v_ind in faces[f_ind]]))
        if a.shape[0] == 2:
            a = np.concatenate((a, np.array([[1, 1, 1]])), axis=0)
        # sum_area += abs(np.linalg.det(a))/2
        # Area of triangle = | AB x AC | / 2
        sum_area += area(a)
```

```
    return sum_area


def neighbors_force(v_index, graph_con, verts_2d, verts_3d):
    total_neighbors_force = np.array([0.0, 0.0])
    total_neighbors_energy = 0
    for neigh_index in graph_con[v_index]:
        # calculate force in direction ij
        vi = np.array(verts_2d[v_index])
        vj = np.array(verts_2d[neigh_index])
        spring_k = 0.5

        v3di = np.array(verts_3d[v_index])
        v3dj = np.array(verts_3d[neigh_index])

        direction_vector = vj - vi
        distance_ij = np.linalg.norm(direction_vector)
        direction_vector = direction_vector / np.linalg.norm(direction_vector)
        distance_ij_3d = np.linalg.norm(v3dj - v3di)
        try:
            force_ij = spring_k*(distance_ij - distance_ij_3d) * direction_vector
        except FloatingPointError:
            print("error")
            raise FloatingPointError

        total_neighbors_force += force_ij
        total_neighbors_energy += spring_k*pow(distance_ij - distance_ij_3d, 2)
    return total_neighbors_force, total_neighbors_energy


def perform_energy_model(verts, faces, max_iteration=1000, label='energy_model', step=0.01):

    x = np.array([[verts[face[0]][0], verts[face[1]][0], verts[face[2]][0]] for face in faces])
    y = np.array([[verts[face[0]][1], verts[face[1]][1], verts[face[2]][1]] for face in faces])
    z = np.array([[verts[face[0]][2], verts[face[1]][2], verts[face[2]][2]] for face in faces])

    surface_area = calculate_total_area(verts, faces)
    print("initial surface area: " + str(surface_area))
    # print('init area: ' + str(surface_area))

    # Create conectivity graph
    graph_con = dict()
    graph_tr = dict()
    for f_ind, face in enumerate(faces):
        for index in face:
            cut_face = face.copy()
            cut_face.remove(index)
            if not index in graph_con:
                graph_con.update({index: set(cut_face)})
                graph_tr.update({index: set([f_ind])})
            else:
                neighbors = graph_con[index]
                [neighbors.add(val) for val in cut_face]
                graph_tr[index].add(f_ind)

    # Initialize points in 2d surface
    verts_2d = [v[0:2] for v in verts]
    initial_areas = calculate_areas(verts, faces)
    print("First mse: " + str(mse(initial_areas, calculate_areas(verts_2d, faces))))

    print(graph_tr)
    # Energy model algorithm

    # Calculate mass
    mass = dict()
    for v_ind in graph_con.keys():
        # Calculate mass of point not scaled
        m_i = vertex_mass(v_ind, graph_tr, faces, verts)
        mass.update({v_ind: m_i})

    for v_ind in graph_con.keys():
        mass[v_ind] = mass[v_ind] / (surface_area / 3)

    # Iterate
    time_step = step

    times = []
    areas = []
    error = []
    mse_list = []
    # q_vel = np.array([0.0, 0.0])
    q_vels = np.zeros((len(graph_con.keys()), 2))
    # pior_energy = 0
    it = 0
```

```
    while True:
        if max_iteration != -1 and it >= max_iteration:
            break
        total_energy = 0
        t = it * time_step
        times.append(t)
        verts_2d_new = verts_2d.copy()
        for v_ind in graph_con.keys():
            # Compute  tensil force
            n_force, n_energy = neighbors_force(v_ind, graph_con, verts_2d, verts)
            q_acel = n_force / mass[v_ind]
            total_energy += n_energy
            verts_2d_new[v_ind] = verts_2d[v_ind] + q_vels[v_ind] * time_step + q_acel * pow(time_step, 2) / 2
            q_vels[v_ind] = q_vels[v_ind] + time_step * q_acel

        verts_2d_old = verts_2d.copy()
        verts_2d = verts_2d_new.copy()
        areas_2d = calculate_areas(verts_2d, faces)
        areas.append(areas_2d.sum())
        error.append(abs((surface_area - areas[len(areas) - 1])) / surface_area)
        mse_list.append(mse(initial_areas, areas_2d))
        print("Current mse: " + str(mse_list[-1]))
        print("Current area " + str(areas_2d.sum()))
        if len(mse_list) > 1:
            if mse_list[-2] - mse_list[-1] < 0:
                return verts_2d_old, mse_list, areas
        it += 1

        # pior_energy = total_energy
        # print("Current delta energy " + str(delta_energies[-1]))
        # if error[-1] <= 1E-5:
        #     break


    # ax2 = fig.add_subplot(211)
    # line, = ax.semilogy(times, mse_list, label=label)
    print('min mse: ' + str(min(mse_list)))
    return verts_2d, mse_list, areas
    # ax2.plot(times, areas)



# if __name__ == "__main__":
#     # plot error
#     fig = plt.figure()
#     ax1 = fig.add_subplot(111)
#     perform_energy_model(ax=ax1)
#     plt.show()
```

LISTING A.4: geometry_algorithms.py

```
from scipy.spatial import Delaunay
from scipy.spatial import KDTree
from scipy.spatial.transform import Rotation
from sklearn.decomposition import PCA

from data_utils import create_sub_structures
from data_utils import scale
from data_utils import write_off
from data_utils import read_off

from geometry_utils import n_orientation
from geometry_utils import angle_with_z
from geometry_utils import filter_by_height
from geometry_utils import xy_to_bar
from geometry_utils import xyz_to_bar
from geometry_utils import plane_parameters
from geometry_utils import z_proj
from geometry_utils import tetra_is_inorder
from geometry_utils import find_tetra
from geometry_utils import point_inside_square
from geometry_utils import triangle_quality_measure
from geometry_utils import neighbors_quality_measure
from geometry_utils import angle
from geometry_utils import check_tetra_deg

from energy_model import perform_energy_model
from ufo_energy_release.ufo_energy_release import perform_ufo_energy_release
from plot_utils import *

import subprocess
import re
```

```
def sub_sample_surface(points, faces, h_val=None, top_surface=True):
    # h_val = (points[:, 2].max() + points[:, 2].min()) / 2
    # print(h_val)
    fil_angle = [angle_with_z(*points[tr_in], min_z=(not top_surface)) < 1.0 for tr_in in faces]
    if h_val is not None:
        fil_h = [filter_by_height(*points[tr_in], h_val, top_surface) for tr_in in faces]
    else:
        fil_h = [True] * faces.shape[0]

    mask = [tup[0] and tup[1] for tup in zip(fil_angle, fil_h)]

    fil_points, fil_faces, hull_to_map = create_sub_structures(points, faces[mask])
    points_2d = fil_points[:, 0:2]

    tree = KDTree(points_2d)
    distance_vector = (tree.query(points_2d, k=2)[0])[:, 1]
    grid_step = distance_vector.mean()*2
    min_x, min_y = points_2d.min(axis=0)
    max_x, max_y = points_2d.max(axis=0)

    x_grid = np.append(np.arange(start=min_x-(grid_step/10), stop=max_x, step=grid_step),
                       np.array([max_x+(grid_step/10)]))
    y_grid = np.append(np.arange(start=min_y-(grid_step/10), stop=max_y, step=grid_step),
                       np.array([max_y+(grid_step/10)]))

    n_x = x_grid.shape[0]-1
    n_y = y_grid.shape[0]-1

    grid_index = (np.ones((n_x, n_y))*-1).astype('int')

    for x in range(n_x):
        squ_x_min, squ_x_max = x_grid[x], x_grid[x + 1]
        for y in range(n_y):
            squ_y_min, squ_y_max = y_grid[y], y_grid[y + 1]
            cen_x, cen_y = (squ_x_min + squ_x_max)/2, (squ_y_min + squ_y_max)/2
            found_pts_in = tree.query_ball_point([cen_x, cen_y], r=1.5 * grid_step)
            if len(found_pts_in) != 0:
                for i, pt in enumerate(fil_points[found_pts_in]):
                    if point_inside_square(pt, (squ_x_min, squ_x_max), (squ_y_min, squ_y_max)):
                        if grid_index[x, y] == -1:
                            grid_index[x, y] = found_pts_in[i]
                        else:
                            if pt[2] > fil_points[grid_index[x, y]][2] if top_surface else pt[2] < fil_points[grid_index[x, y]][2]:
                                grid_index[x, y] = found_pts_in[i]

    sample_points = fil_points[grid_index.flatten()[grid_index.flatten() != -1]]
    tri = Delaunay(sample_points[:, 0:2], qhull_options='Qj')
    sample_faces = tri.simplices

    tr_qualities = np.array([triangle_quality_measure(sample_points[simplex]) for simplex in sample_faces])
    tr_qualities = scale(tr_qualities)

    # plot_triangulation(sample_points, sample_faces)
    fil_qual = [tr_q > 0.1 for tr_q in tr_qualities.flatten()]
    sample_points, sample_faces, hull_to_map = create_sub_structures(sample_points, sample_faces[fil_qual])
    # plot_surface(sample_points, sample_faces[fil_qual])

    return sample_points, sample_faces


def divide_volume_surface(points, faces, only_top=False, only_bot=False):
    # h_val = points[:, 2].min() + (points[:, 2].max() - points[:, 2].min())/10
    top_tr_indices = np.array([n_orientation(points[f[0]], points[f[1]], points[f[2]]) for f in faces])
    top_points, top_triangles, hull_to_top_map = create_sub_structures(points, faces[top_tr_indices])
    top_points, top_triangles = sub_sample_surface(top_points, top_triangles)

    # h_val = points[:, 2].max() - (points[:, 2].max() - points[:, 2].min()) / 10
    bot_tr_indices = np.array([not n_orientation(points[f[0]], points[f[1]], points[f[2]]) for f in faces])
    bot_points, bot_triangles, hull_to_bot_map = create_sub_structures(points, faces[bot_tr_indices])
    bot_points, bot_triangles = sub_sample_surface(bot_points, bot_triangles, top_surface=False)

    if only_top == False and only_bot == False:
        return (top_points, top_triangles), (bot_points, bot_triangles)
    elif only_top:
        return top_points, top_triangles
    else:
        return bot_points, bot_triangles


def unfold_surface(points, faces, step=0.01, use_ufo=False, iterations=15):
    if use_ufo:
        flt_points, mse, areas = perform_ufo_energy_release(data=(points, faces), step=step, max_iteration=-1)
    else:
        points_list, faces_list = points.tolist(), faces.tolist()
        flt_points, mse, areas = perform_energy_model(points_list, faces_list, step=step, max_iteration=iterations)
```

```
    n_points = points.shape[0]
    height_index = 2

    h = points[:, height_index].mean()
    h_vec = (np.ones(n_points) * h).reshape((n_points, 1))
    proj_points = np.append(np.array(flt_points)[:, 0:2], h_vec, axis=1)
    return proj_points, np.array(mse), np.array(areas)


def join_surfaces(top_points, top_triangles, bot_points, bot_triangles, top_proj_points=None):
    data_tree = KDTree(top_points[:, 0:2])
    dis, neigh_bot2top = data_tree.query(bot_points[:, 0:2])

    point_face_conv = [[] for i in range(top_points.shape[0])]

    for f_ind, face in enumerate(top_triangles):
        for p_ind in face:
            point_face_conv[p_ind].append(f_ind)

    new_neigh_top2bot = [np.NAN] * top_points.shape[0]
    new_neigh_bot2top = neigh_bot2top
    new_top = top_points.copy()
    new_top_flt = top_proj_points.copy() if top_proj_points is not None else None
    top_h = top_proj_points[0][2] if top_proj_points is not None else None

    for i, p in enumerate(bot_points):
        near_top = top_points[neigh_bot2top[i]]
        if (p[0:2] - near_top[0:2]).sum() == 0.0:
            new_neigh_top2bot[neigh_bot2top[i]] = i
        else:
            # project bot_point into top surface and add
            # find triangles in which bot projection will be inside
            for tr in point_face_conv[new_neigh_bot2top[i]]:
                # print(top_points[top_triangles[tr]])
                c_bar = xy_to_bar(top_points[top_triangles[tr]][:, 0:2], p[0:2])
                p_inside = [c_bar[i] >= 0.0 and c_bar[i] <= 1.0 for i in range(3)]
                if p_inside[0] and p_inside[1] and p_inside[2]:
                    a, b, c, d = plane_parameters(*(top_points[top_triangles[tr]]))
                    new_point = np.array([p[0], p[1], z_proj(*(p[0:2]), a, b, c, d)])
                    new_top = np.append(new_top, new_point.reshape((1, 3)), axis=0)

                    if top_proj_points is not None:
                        tr_flt_verts = top_proj_points[top_triangles[tr]][:, 0:2]
                        new_pt_flt = np.array([c_bar[j] * tr_flt_verts[j] for j in range(0, c_bar.shape[0])]).sum(
                            axis=0) / c_bar.sum()
                        new_pt_flt = np.append(new_pt_flt, top_h)
                        new_top_flt = np.append(new_top_flt, new_pt_flt.reshape((1, 3)), axis=0)

                    new_neigh_top2bot.append(i)
                    new_neigh_bot2top[i] = new_top.shape[0] - 1
                    break

    up_points = bot_points.copy()
    up_triangles = bot_triangles.copy()
    up_proj_points = bot_points.copy()

    for i, p in enumerate(bot_points):
        up_points[i] = new_top[new_neigh_bot2top[i]]
        up_proj_points[i] = new_top_flt[new_neigh_bot2top[i]] if top_proj_points is not None else None

    return up_points, up_triangles, up_proj_points


def create_tetra_mesh(up_points, up_triangles, bot_points, bot_triangles, up_proj_points=None, bot_proj_points=None):
    tetra_points = np.append(up_points, bot_points, axis=0)
    n_faces = up_triangles.shape[0]
    n_points = up_points.shape[0]
    tetra_simplices = []
    for face_index in range(0, n_faces):
        face1 = up_triangles[face_index]
        face2 = bot_triangles[face_index] + n_points

        tetra1 = np.append(face1, face2[0]).tolist()
        tetra2 = np.append(face2, face1[1]).tolist()
        tetra3 = [face1[1], face1[2], face2[0], face2[2]]

        if not tetra_is_inorder(tetra_points[tetra1]):
            tetra1[0], tetra1[1] = tetra1[1], tetra1[0]
        if not tetra_is_inorder(tetra_points[tetra2]):
            tetra2[0], tetra2[1] = tetra2[1], tetra2[0]
        if not tetra_is_inorder(tetra_points[tetra3]):
            tetra3[0], tetra3[1] = tetra3[1], tetra3[0]

        #     print(tetra_is_inorder(tetra_points[tetra1]))
        #     print(tetra_is_inorder(tetra_points[tetra2]))
        #     print(tetra_is_inorder(tetra_points[tetra3]))
```

```
        # if not check_tetra_deg(tetra_points[tetra1]):
        tetra_simplices.append(tetra1)
        # if not check_tetra_deg(tetra_points[tetra2]):
        tetra_simplices.append(tetra2)
        # if not check_tetra_deg(tetra_points[tetra3]):
        tetra_simplices.append(tetra3)

    tetra_simplices = np.array(tetra_simplices)

    tetra_points_flt = np.append(up_proj_points, bot_proj_points, axis=0)
    print(tetra_points_flt.shape)
    print(tetra_points.shape)

    return tetra_points, tetra_simplices, tetra_points_flt


def search_points_inside_tetra_mesh(points, tetra_points, tetra_simplices, ref_points, ref_faces):
    # Assign a tetrahedron for each point in block model
    data_tree = KDTree(ref_points[:, 0:2])
    dis, neighbors = data_tree.query(points[:, 0:2])
    print(neighbors)
    print(neighbors.shape)
    print(np.unique(neighbors).shape)

    point_face_conv = [[] for i in range(ref_points.shape[0])]

    for f_ind, face in enumerate(ref_faces):
        for p_ind in face:
            point_face_conv[p_ind].append(f_ind)

    tetra_indexeses = [None] * points.shape[0]
    inside_tetras = [None] * points.shape[0]
    for i, p in enumerate(points):
        if i%100000:
            print
        # potential candidates
        potential_prisms = point_face_conv[neighbors[i]]
        # print(potential_prisms)
        potential_tetra = [v * 3 + j for v in potential_prisms for j in range(3)]
        # print(potential_tetra)
        tet_ind, inside_tet = find_tetra(p, tetra_points, tetra_simplices[potential_tetra])
        inside_tetras[i] = inside_tet
        tetra_indexeses[i] = potential_tetra[tet_ind]
    return tetra_indexeses, inside_tetras


def apply_barycentric_deformation(points, tetra_points, tetra_simplices, tetra_points_flt, tetra_indexeses):
    # Apply barycentric transformation
    points_deformed = points.copy()

    for i, p in enumerate(points):
        tetra_index = tetra_indexeses[i]
        if tetra_index == -1:
            continue

        tetra = tetra_simplices[tetra_index]
        c = xyz_to_bar(tetra_points[tetra], p)
        new_point = np.array([c[j] * tetra_points_flt[tetra[j]] for j in range(0, c.shape[0])]).sum(axis=0) / c.sum()
        points_deformed[i] = new_point

    # not_inside = [(not bo) for bo in inside_tetras]
    # near_tetras = np.unique((np.array(tetra_indexeses))[not_inside])
    # near_pts = points[not_inside]

    return points_deformed


def calculate_quality_measures(points, deformed_points):
    initial_index, nei = neighbors_quality_measure(points)
    final_index, nei = neighbors_quality_measure(deformed_points, neighbors=nei)

    # return initial_index, final_index, np.abs(initial_index-final_index)/(initial_index.max()-initial_index.min())
    return initial_index, final_index, np.abs(initial_index-final_index) / initial_index


def generate_concave_hull(points, original_name):
    name = re.search('/(.*).csv', original_name)[1]
    write_off(points, name)
    output_name = "datasets/" + name + "_concave"
    ret = subprocess.run(["./concave_hull/concave_hull_alpha_shapes", name, output_name], stdout=subprocess.PIPE)
    print('{}'.format(ret.stdout.decode("utf-8")))
    chull_points, chull_faces = read_off(output_name+'.off')
    subprocess.run(['rm', name])
    return chull_points, chull_faces
```

```
def rotate_point_cloud(points):
    # points = read_block_model("datasets/magenta.csv")
    # translate points to center
    centroid = points.sum(axis=0) / points.shape[0]

    pca = PCA(n_components=2, whiten=True)
    pca.fit(points)
    pca_vectors = np.array(pca.components_)

    z_vector = np.array([0, 0, 1])
    pca_z_vector = np.cross(pca_vectors[0, :], pca_vectors[1, :])
    pca_vectors = np.append(pca_vectors, pca_z_vector.reshape((1, 3)), axis=0)

    # angle_z = angle(z_vector, pca_z_vector)
    theta_p1 = angle(z_vector, pca_z_vector)
    # theta_p1 = np.pi / 2 - theta_p1
    print("theta_p1: {} ".format(np.rad2deg(theta_p1)))

    theta = theta_p1
    direction = np.cross(pca_z_vector, z_vector)
    print("Rotation axis: {}".format(direction))

    # R = rotation_matrix(theta, direction)
    # R = R[0:3, 0:3]  # Transform back from uniform coordinates

    rot = Rotation.from_rotvec(theta* direction)
    R = rot.as_dcm()

    # Rotate points around block-model center
    # plot_scatter(points-centroid)
    rotated_points = np.dot(R, (points - centroid).T)
    rotated_points = rotated_points.T + centroid

    # arrows_scale = 0.00001
    # plot_scatter(np.append(points, rotated_points, axis=0), scalars=np.array([[1]*points.shape[0],[2]*points.shape[0]]), arrows=(np.array([ce
    # plot_scatter( rotated_points, scalars=np.array([1]*points.shape[0]), arrows=(np.array([centroid]*2), np.array([z_vector,pca_z_vector])*ar

    #plot_scatter(points)

    return rotated_points
```

LISTING A.5: geometry_utils.py

```
import math
from sklearn.neighbors import kneighbors_graph
import numpy as np
from numpy.linalg import inv
from plot_utils import *
from scipy.spatial import KDTree


def fix_orientation(face, points):
    v1 = points[face[0]]
    v2 = points[face[1]]
    v3 = points[face[2]]
    n = triangle_normal(v1, v2, v3)
    face_mid = (v1 + v2 + v3) / 3

    centroid = points.sum(axis=0) / points.shape[0]
    new_face = face.copy()
    change = False
    if (np.dot(face_mid - centroid, n) < 0):
        aux = new_face[0]
        new_face[0] = new_face[1]
        new_face[1] = aux
        change = True

    return new_face, change


def plane_parameters(p1, p2, p3, d=1.0):
    M = np.concatenate([p1.reshape((3,1)), p2.reshape((3,1)), p3.reshape((3,1))], axis=1).T
    D = np.linalg.det(M)
#     print(p1, p2, p3)
    if D != 0.0:
        one_vec = np.ones((3,1))
        a = -d * np.linalg.det(np.concatenate([one_vec, M[:,1].reshape((3,1)), M[:,2].reshape((3,1))], axis=1)) / D
        b = -d * np.linalg.det(np.concatenate([M[:,0].reshape((3,1)), one_vec, M[:,2].reshape((3,1))], axis=1)) / D
        c = -d *np.linalg.det(np.concatenate([M[:,0].reshape((3,1)), M[:,1].reshape((3,1)), one_vec], axis=1)) / D
        return a, b, c, d
    else:
        d = 0.0
```

```
        a, b,c = np.cross(p2-p1, p3-p2)
        return a, b, c, d


def triangle_normal(v0, v1, v2):
    u = v1 - v0
    v = v2 - v1
    n = np.cross(u, v)
    return n / np.linalg.norm(n)


def area(face_mat):
    try:
        # A = pow(np.linalg.det(np.vstack((face_mat[0: 2], np.ones(3)))), 2)
        # B = pow(np.linalg.det(np.vstack((face_mat[1: 3], np.ones(3)))), 2)
        # C = pow(np.linalg.det(np.vstack((face_mat[2, :], face_mat[0, :], np.ones(3)))), 2)
        # return np.sqrt(A+B+C) / 2

        # Area of triangle = | AB x AC | / 2
        return np.linalg.norm(np.cross(face_mat[:, 1] - face_mat[:, 0], face_mat[:, 2] - face_mat[:, 0])) / 2.0
    except FloatingPointError:
        print("Warning raised")
        return np.linalg.norm(np.cross(face_mat[:, 1] - face_mat[:, 0], face_mat[:, 2] - face_mat[:, 0])) / 2.0


def calculate_total_area(verts, faces):
    total_area = 0
    for f in faces:
        M = np.transpose(np.array([verts[ind] for ind in f]))
        if M.shape[0] < 3:
            M = np.vstack((M, np.zeros(3)))
        total_area += area(M)
    return total_area


def calculate_areas(verts, faces):
    areas = []
    for f in faces:
        M = np.transpose(np.array([verts[ind] for ind in f]))
        if M.shape[0] < 3:
            M = np.vstack((M, np.zeros(3)))
        areas.append(area(M))
    return np.array(areas)


def unfold_diff_areas(verts0, faces0, verts1, faces1):
    areas0, areas1 = calculate_areas(verts0, faces0), calculate_areas(verts1, faces1)
    return np.array([(a1-a0)/a0 for a0, a1 in zip(areas0, areas1)])


def n_orientation(v0, v1, v2):
    n = triangle_normal(v0, v1, v2)
    return n[2] > 0


def tetra_is_inorder(points):
    return right_of_plane(points[0], points[1], points[2], points[3])


def right_of_plane(p0, p1, p2, p):
    v1 = p1 - p0
    v2 = p2 - p0
    v3 = p - p0
    norm12 = np.cross(v1, v2)
    determinant = np.dot(norm12, v3)
    #     print(determinant)
    #     if determinant == 0:
    #         print("point is in plane")
    return determinant >= 0


def is_point_in_tetra(p0, p1, p2, p3, p):
    cond1 = right_of_plane(p0, p1, p2, p)
    cond2 = right_of_plane(p2, p1, p3, p)
    cond3 = right_of_plane(p0, p2, p3, p)
    cond4 = right_of_plane(p0, p3, p1, p)
    # print(cond1, cond2, cond3, cond4)
    return cond1 and cond2 and cond3 and cond4


def find_tetra(point, vertices, tetras):
    closest_distance = np.Inf
    closest_index = -1
    for tet_ind, tet in enumerate(tetras):
        tet_verts = vertices[tet]
        c = xyz_to_bar(tet_verts, point)
```

```
        if c is None:
            continue
        point_inside = True
        bar_dist = 0
        for c_ind in range(4):
            bool_check = (c[c_ind] >= 0 and c[c_ind] <= 1.0)
            if not bool_check:
                bar_dist = bar_dist + min(abs(c[c_ind] - 0), abs(c[c_ind] - 1.0))
            point_inside = point_inside and bool_check

        if bar_dist < closest_distance:
            closest_distance = bar_dist
            closest_index = tet_ind

        if point_inside:
            return tet_ind, True
    return closest_index, False


def xyz_to_bar(vertices, r):
    # check degeneration
    if check_tetra_deg(vertices):
        return None
    try:
        T = np.array([vertices[0:3, i] - vertices[3, i] for i in range(0, 3)])
        c = np.matmul(inv(T), (r - vertices[3]))
        c = np.append(c, 1 - c[0] - c[1] -c[2])
    except np.linalg.LinAlgError:
        print("linalg.LinAlgError inverting matrix")
        return None
    return c


def check_tetra_deg(vertices):
    Q, R = np.linalg.qr(vertices.T)
    for i in range(0, 3):
        if abs(R[i, i]) < 1e-10:
            return True
    return False


def xy_to_bar(vertices, r):
    # print(vertices, r)
    T = np.array([vertices[0:2, i] - vertices[2,i] for i in range(0,2)])
    c = np.matmul(inv(T), (r - vertices[2]))
    c = np.append(c, 1 - c[0] - c[1])
    return c


def angle_with_z(p0, p1, p2, min_z=False):
    normal = np.cross(p1 - p0, p2 - p1)
    z_vec = np.zeros(3)
    z_vec[2] = 1.0 if not min_z else -1.0

    angle = np.arccos(np.dot(normal, z_vec) / (np.linalg.norm(normal) * np.linalg.norm(z_vec)))
    #     print(angle)
    return angle


def filter_by_height(p0, p1, p2, h, above=True):
        return (p0[2] > h and p1[2] > h and p2[2] > h) if above else (p0[2] < h and p1[2] < h and p2[2] < h)


def z_proj(x, y, a, b, c, d=1.0):
    return -(d + a*x + b*y) / c


def point_inside_square(pt, margins_x, margins_y):
    return (pt[0] > margins_x[0] and pt[0] <= margins_x[1] and pt[1] > margins_y[0] and pt[1] <= margins_y[1])


def dotproduct(v1, v2):
    return sum((a*b) for a, b in zip(v1, v2))


def length(v):
    return np.linalg.norm(v)


def angle(v1, v2):
    val = (v1*v2).sum() / (length(v1) * length(v2))
    # print(val)
    if val > 1.0:
        val = 1.0
    elif val < -1.0:
        val = -1.0
```

```
    val = val if val < 1.0 else 1.0
    return abs(math.acos(val))


def neighbors_quality_measure_old(points, n_neigh=26):
    graph = kneighbors_graph(points, n_neigh, mode='connectivity', include_self=False)
    neigh_list = []
    i = 0
    print(graph.shape)
    for p_index, row in enumerate(graph):
        i = i+1
        if (i %10000 ==0):
            print(i)
        nei_indexes = np.where(row.T.todense() == 1.0)[0]
        neigh_prom = np.linalg.norm(points[nei_indexes] - points[p_index], axis=1).sum()/n_neigh
        neigh_list.append(neigh_prom)
    return np.array(neigh_list)


def neighbors_quality_measure(points, n_neigh=27, neighbors=None):
    if neighbors is None:
        tree = KDTree(points)
        distances, neighbors = tree.query(points, k=n_neigh)
    else:
        print('skiping neighbors construction')
    neigh_list = []
    # print(graph.shape)
    for p_index, row in enumerate(neighbors):

        if (p_index %10000 ==0):
            print(p_index)

        nei_indexes = row
        neigh_prom = np.linalg.norm(points[nei_indexes] - points[p_index], axis=1).sum()/n_neigh
        neigh_list.append(neigh_prom)
    return np.array(neigh_list), neighbors


def triangle_quality_measure(pts):
    pts_2d = pts[:, 0:2]
    p0 = pts_2d[0]
    p1 = pts_2d[1]
    p2 = pts_2d[2]

    a = np.linalg.norm(p0 - p1)
    b = np.linalg.norm(p1 - p2)
    c = np.linalg.norm(p2 - p0)

    # print(n_orientation(p0, p1, p2))
    # a2 = length(pts_2d[0] - pts_2d[1])
    # b2 = length(pts_2d[1] - pts_2d[2])
    # c2 = length(pts_2d[2] - pts_2d[0])

    alpha = angle(p0 - p1, p2 - p1)
    beta = angle(p1 - p2, p0 - p2)
    gamma = angle(p1 - p0, p2 - p0)
    # print(alpha+beta+gamma)


    # a_vec =  angle(pts_2d[1] - pts_2d[0], pts_2d[0] - pts_2d[2] )

    # alpha = np.arccos((b**2 + c**2 -a**2)/(2*b*c))
    # beta = np.arccos((a**2 + c**2 -b**2)/(2*a*c))
    # gamma = np.arccos((a**2 + b**2 - c**2)*(2*a*b))


    s = (a + b + c) / 2 #perimeter
    # calculate the area
    # A = (s * (s - a) * (s - b) * (s - c)) ** 0.5

    return np.min([alpha, beta, gamma])/(s*2)
```

LISTING A.6: plot_utils.py

```
import pyvista as pv
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.tri import Triangulation
import vtk


def create_polydata(points, faces):
    # mesh points
```

```
    vertices = points

    # mesh faces
    n_faces = faces.shape[0]
    face_types = (np.ones(faces.shape[0]) * 3).reshape((n_faces, 1)).astype(int)
    faces = np.append(face_types, faces, axis=1)

    surf = pv.PolyData(vertices, faces)
    return surf


def plot_surface(points, faces, scalars=None, show_normals=False, normals_mag=5.0, export_file=None):
    n_faces = faces.shape[0]
    if scalars is None:
        scalars = np.arange(n_faces)

    surf = create_polydata(points, faces)
    plotter = pv.Plotter()
    plotter.add_mesh(surf, scalars=scalars, show_edges=True, stitle='Colors', opacity=0.9)
    # plotter.add_points(points, point_size=5.0)

    if show_normals:
        normals = []
        centroids = []
        for f in faces:
            v0, v1, v2 = points[f[0]], points[f[1]], points[f[2]]
            centroids.append((v0 + v1 + v2) / 3.0)
            normal = np.cross(v1 - v0, v2 - v1)
            normal = normal / np.linalg.norm(normal)
            normals.append(normal)
        normals = np.array(normals)
        centroids = np.array(centroids)
        plotter.add_arrows(centroids, normals, mag=normals_mag)

    plotter.show_grid()
    plotter.show() if not export_file else plotter.mesh.save(export_file)


def plot_tetras(points, simplices, all_points=None, scalars=None, opacity=0.3, point_size=1.0, export_file=None):
    # mesh points
    vertices = points.copy()

    # mesh faces
    faces = simplices.copy()

    # convert to vtki unstructered grid
    nvertices = vertices.shape[0]
    nfaces = faces.shape[0]
    # offset array.  Identifies the start of each cell in the cells array
    offset = np.array([i * 5 for i in range(0, nfaces)])

    # composing the cell
    face_types = (np.ones(faces.shape[0]) * 4).reshape((nfaces, 1)).astype(int)
    cells = np.hstack(np.append(face_types, faces, axis=1))

    cell_type = np.array([vtk.VTK_TETRA] * nfaces)

    # create the unstructured grid directly from the numpy arrays
    grid = pv.UnstructuredGrid(offset, cells, cell_type, vertices)

    plotter = pv.Plotter()
    plotter.add_mesh(grid,
                     scalars=np.arange(nfaces),
                     stitle='Colors',
                     show_edges=True,
                     opacity=opacity)

    if all_points is not None:
        plotter.add_points(all_points, scalars=scalars, point_size=point_size)
        plotter.add_scalar_bar()
    plotter.show_grid()
    plotter.show() if not export_file else plotter.mesh.save(export_file)


def plot_scatter(points, scalars=None, opacity=1.0, rng=None, cmap=None, arrows=None, arrows_mag=1, export_file=None):
    vertices = points.copy()

    plotter = pv.Plotter()
    plotter.add_points(vertices,
                       point_size=10.0,
                       scalars=scalars,
                       opacity=opacity,
                       cmap=cmap, rng=rng)

    if scalars is not None:
        plotter.add_scalar_bar()
```

```
    if arrows is not None:
        cen, direction = arrows
        plotter.add_arrows(cen, direction, mag=arrows_mag)

    #     plotter.add_mesh(grid,
    #                      style='points',
    #                      point_size=10.0)
    plotter.show_grid()

    plotter.show() if not export_file else plotter.mesh.save(export_file)


def plot_surface_and_points(points, faces, all_points, scalars=None):
    n_faces = faces.shape[0]
    if (scalars is None):
        scalars = np.arange(n_faces)

    surf = create_polydata(points, faces)
    plotter = pv.Plotter()
    plotter.add_mesh(surf, scalars=scalars, show_edges=True, stitle='Colors', opacity=0.4)
    plotter.add_points(all_points, point_size=3.0)
    plotter.show_grid()
    plotter.show()


def plot_two_surfaces(points1, faces1, points2, faces2, all_points=None, scalars=None, opacity=0.5):
    n_faces1 = faces1.shape[0]
    scalars1 = np.arange(n_faces1)

    n_faces2 = faces2.shape[0]
    scalars2 = np.arange(n_faces2)

    surf1 = create_polydata(points1, faces1)
    surf2 = create_polydata(points2, faces2)
    plotter = pv.Plotter()
    plotter.add_mesh(surf1, scalars=scalars1, opacity=opacity, show_edges=True, stitle='Colors')
    # plotter.add_points(points1, point_size=5.0, scalars=scalars)

    plotter.add_mesh(surf2, scalars=scalars2, opacity=opacity, show_edges=True, stitle='Colors')
    # plotter.add_points(points2, point_size=5.0, scalars=scalars)

    if all_points is not None:
        plotter.add_points(all_points)

    plotter.show_grid()
    plotter.show()


def plot_triangulation(points, triangles, scalars=None, title=''):
    plt.tripcolor(points[:, 0], points[:, 1], triangles, scalars)
    plt.colorbar()
    plt.title(title)
    plt.show()


def plot_mse(mse, areas=None, title=''):
    fig, ax1 = plt.subplots()
    its = np.array([i+1for i in range(mse.shape[0])])
    # color = 'tab:darkGreen'
    ax1.plot(its, mse, color='C2')
    ax1.set_xlabel('Iterations')
    ax1.set_ylabel('Mean␣Square␣Error', color='C2')
    ax1.tick_params(axis='y', labelcolor='C2')

    if areas is not None:
        ax2 = ax1.twinx()
        ax2.set_ylabel('Flat␣Area␣/␣Surface␣Area', color='C0')
        ax2.plot(its, areas, color='C0')
        ax2.tick_params(axis='y', labelcolor='C0')
    fig.tight_layout()
    plt.title(title)
    # plt.grid(True)
    plt.show()


def plot_triangles_diff(orig_verts, flat_verts, triangles):
    one_triangle = triangles[0]

    one_triangle_pts_flt = np.array([flat_verts[ind, :] for ind in one_triangle])
    x_flt = one_triangle_pts_flt[:, 0]
    y_flt = one_triangle_pts_flt[:, 1]
    triang_flt = Triangulation(x_flt, y_flt, triangles=[[0, 1, 2]])

    original_verts = np.array(orig_verts)
    one_triangle_pts = np.array([original_verts[ind, :] for ind in one_triangle])
```

```
    x = one_triangle_pts[:, 0]
    y = one_triangle_pts[:, 1]
    triang = Triangulation(x, y, triangles=[[0, 1, 2]])

    plt.figure()
    plt.gca().set_aspect('equal')
    line2 = plt.triplot(triang, 'bo-', lw=1)
    line = plt.triplot(triang_flt, 'rx--', lw=1)
    plt.title('One Triangle')
    print(line)
    plt.legend(line + line2, ['flat_triangle', 'flat_points', 'original_triangle', 'original_points'])
    plt.show()


def plot_hist(values, coords=(.1, 2000), bins=100):
    plt.hist(values, bins=bins)
    plt.xlabel('Error Values')
    plt.ylabel('Frequency')
    plt.title('Frequency Error')
    plt.grid(True)
    plt.text(coords[0], coords[1], r'$\mu={0:6f},\ \sigma={0:6f}$'.format(np.mean(values), np.std(values)))
    plt.show()


def plot_cumulative(values, bins=100):
    plt.hist(values, bins=bins, cumulative=True, histtype='step')
    plt.xlabel('Error Values')
    plt.ylabel('Total Samples')
    plt.title('Cumulative Error Distribution')
    plt.grid(True)
    plt.show()
```

A.2 C++ Source Code for CGAL

LISTING A.7: concave_hull_alpha_shapes.cpp

```
#include <CGAL/Exact_predicates_inexact_constructions_kernel.h>
#include <CGAL/Delaunay_triangulation_3.h>
#include <CGAL/Alpha_shape_3.h>
#include <CGAL/IO/Writer_OFF.h>

#include <string>
#include <fstream>
#include <list>
#include <cassert>
#include <iostream>
#include <set>
#include <map>

typedef CGAL::Exact_predicates_inexact_constructions_kernel Gt;

typedef CGAL::Alpha_shape_vertex_base_3<Gt>           Vb;
typedef CGAL::Alpha_shape_cell_base_3<Gt>             Fb;
typedef CGAL::Triangulation_data_structure_3<Vb,Fb>   Tds;
typedef CGAL::Delaunay_triangulation_3<Gt,Tds>        Triangulation_3;
typedef CGAL::Alpha_shape_3<Triangulation_3>          Alpha_shape_3;


typedef Alpha_shape_3::Cell_handle     Cell_handle;
typedef Alpha_shape_3::Vertex_handle   Vertex_handle;

typedef Alpha_shape_3::Facet                                Facet;
typedef Alpha_shape_3::Edge                                 Edges;
typedef Alpha_shape_3::Vertex                               Vertex;

typedef Gt::Point_3                                        Point;
typedef Alpha_shape_3::Alpha_iterator                     Alpha_iterator;

int main(int argc, const char* const argv[])
{

    if (argc != 3)
    {
        std::cerr << "\n There have to be two parameters (name of program, file with coordinates, output file)" << std::endl;
        exit(1);
    }

    std::list<Point> lp;

    //read input
```

```
    std::ifstream is(argv[1]);
    is.precision(20);
    std::cout.precision(20);
    int n;
    is >> n;
    std::cout << "Reading " << n << " points " << std::endl;
    Point p;
    for( ; n>0 ; n--)    {
        is >> p;
        if(n==1) {
            std::cout << p[0] << std::endl;
        }

        lp.push_back(p);
    }

    // compute alpha shape
    Alpha_shape_3 as(lp.begin(),lp.end());
    std::cout << "Alpha shape computed in REGULARIZED mode by default"
        << std::endl;

    // find optimal alpha value
    Alpha_iterator opt = as.find_optimal_alpha(1);
    Alpha_shape_3::NT alpha_solid = as.find_alpha_solid();
    std::cout << "Optimal alpha value to get one connected component is "
        << *opt     << std::endl;

    std::cout << "Smallest alpha value to get a solid through data points is "
              << alpha_solid << std::endl;
    as.set_alpha(*opt);
    assert(as.number_of_solid_components() == 1);

    // collect all regular facets
    std::vector<Facet> Facets;
    as.get_alpha_shape_facets(std::back_inserter(Facets), Alpha_shape_3::REGULAR);
//
    std::stringstream pts;
    pts.precision(20);
    std::stringstream ind;

    std::map<Vertex_handle, int> vertex_map;
    std::vector<Vertex_handle> vertex_vector;
    std::vector<std::vector<int> > triangles_vector;
    int vertex_index = 0;

    std::pair<std::map<Vertex_handle,int>::iterator,bool> ret;

    std::cout << "Number of Facets: " << Facets.size() << std::endl;

    for (size_t i = 0; i < Facets.size(); i++)
    {

        //To have a consistent orientation of the facet, always consider an exterior cell
        if (as.classify(Facets[i].first) != Alpha_shape_3::EXTERIOR)
        {
            Facets[i] = as.mirror_facet(Facets[i]);
        }

        CGAL_assertion(as.classify(Facets[i].first) == Alpha_shape_3::EXTERIOR);

        int indices[3] = { (Facets[i].second + 1) % 4, (Facets[i].second + 2) % 4, (Facets[i].second + 3) % 4 };
//        printf("indices are: %d %d %d \n", indices[0], indices[1], indices[1]);

        if (Facets[i].second % 2 == 0)
        {
            std::swap(indices[0], indices[1]);
        }

//        pts << Facets[i].first->vertex(indices[0])->point() << "\n" <<
//            Facets[i].first->vertex(indices[1])->point() << "\n" <<
//            Facets[i].first->vertex(indices[2])->point() << "\n";

        ind << 3;

//        ret = vertex_map.insert(Facets[i].first->vertex(indices[0]), vertex_index);
//        Vertex_handle v1 = Facets[i].first->vertex(indices[0]);
        ret = vertex_map.insert(std::pair<Vertex_handle, int> (Facets[i].first->vertex(indices[0]), vertex_index));
        if(ret.second == true) {
            ind << " " << vertex_index;
            vertex_index++;
            vertex_vector.push_back(Facets[i].first->vertex(indices[0]));
        } else {
            ind << " " << ret.first->second;
//            std::cout << "Index found: " << ret.first->second << std::endl;

        }
```

```
        ret = vertex_map.insert(std::pair<Vertex_handle, int>(Facets[i].first->vertex(indices[1]), vertex_index));
        if(ret.second == true) {
            ind << " " << vertex_index;
            vertex_index++;
            vertex_vector.push_back(Facets[i].first->vertex(indices[1]));
        } else {
            ind << " " << ret.first->second;
//            std::cout << "Index found: " << ret.first->second << std::endl;
        }

        ret = vertex_map.insert(std::pair<Vertex_handle, int>(Facets[i].first->vertex(indices[2]), vertex_index));
        if(ret.second == true) {
            ind << " " << vertex_index;
            vertex_index++;
            vertex_vector.push_back(Facets[i].first->vertex(indices[2]));
        } else {
            ind << " " << ret.first->second;
//            std::cout << "Index found: " << ret.first->second << std::endl;
        }
        ind << std::endl;

//        std::cout << Facets[i].first->vertex(indices[0])    << std::endl;
    }

    for(int i=0; i < vertex_vector.size(); ++i) {
        pts << vertex_vector[i]->point() << "\n";
    }

    std::cout << "Number of vetices in map: " << vertex_map.size() << std::endl;
    std::cout << "Number of vertices: " << vertex_vector.size() << std::endl;
    std::cout << "Vertex index: " << vertex_index << std::endl;

//
    std::ofstream off_file;
    off_file.precision(20);

    std::stringstream ss;
    ss << argv[2] <<  ".off";
    off_file.open(ss.str(), std::ios::out);
    off_file << "OFF " << "\n" << vertex_vector.size() << " " << Facets.size() << " 0\n";
    off_file << pts.str();
    off_file << ind.str();



    return 0;
}
```