Learning Unsigned Distance Fields to Simulate Brittle Fractures in Real-Time

Version of January 13, 2026

Lukas Kai Zimmerhackl

Learning Unsigned Distance Fields to Simulate Brittle Fractures in Real-Time

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Lukas Kai Zimmerhackl
born in Delft, the Netherlands

Computer Graphics and Visualization Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`www.ewi.tudelft.nl`

Learning Unsigned Distance Fields to Simulate Brittle Fractures in Real-Time

Author: Lukas Kai Zimmerhackl
Student id: 4904176

Abstract

In this thesis, we address the problem of learning mesh-specific, impulse-dependent fracture patterns in real time. Our approach is based on regressing a distance field over the mesh surface, encoding the proximity of each vertex to fracture lines, which is subsequently segmented into distinct pieces using graph-based methods such as watershed segmentation. The goal is to achieve real-time performance, which is something the current approach does not achieve for large meshes.

We evaluate different neural architectures, comparing a multilayer perceptron to DeltaConv, a graph convolutional model, and find that the MLP provides superior performance. In addition, we assess multiple segmentation strategies and identify watershed as the most effective, followed by hierarchical segmentation. We also find that the segmentation algorithms do not achieve real-time performance for large meshes.

These results highlight the potential of machine learning-based fracture simulations, but also indicate that distance field segmentation is not capable of real-time performance using our tested algorithms. This suggests that future work should focus on directly learning the labels rather than relying on distance fields as an intermediary representation in real-time scenarios.

Preface

I firstly would like to thank my parents and sister for always being there for me in the hard and tough times, even when things did not look optimistic. I know the uncertainty caused you a lot of stress, sometimes even more than for me, and I hope I will be able to repay you in the future and wish you nothing but love. Secondly, I want to thank Dr. Petr Kellnhofer and Lukas Uzolas for their guidance and feedback. I would not have been able to complete this journey on my own and I am truly thankful for your help. Without it I most likely still would not have found a suitable subject. I would like to thank my new and old thesis supervisors Prof. Dr. Klaus Hildebrandt and Prof. Dr. Elmar Eisemann for their advice and interest in my work, as well as their willingness to make time for me in their busy schedules. I am also thankful to Prof. Dr. Tom Viering for appearing on my thesis committee on such short notice. I also want to thank all my other friends and family for being there by my side, with special mentions to Karthik for his help in structuring my thesis and Hugo for his help with some of the machine learning aspects.

Lukas Kai Zimmerhackl
Delft, the Netherlands
January 13, 2026

Contents

List of Figures

Chapter 1

Introduction

In modern video games, realism plays a crucial role in enhancing player immersion. One important aspect of this is accurate physical interactions of objects, including fractures and destructions. From entire buildings collapsing in action games like Battlefield, to blocks breaking in more arcade-style games like Minecraft. It is important that the fractures we see in video games feel right, so as not to break the player's immersion. However, achieving realistic, versatile and efficient fracture simulations is often not possible in real-time, and thus a balance must be found between computational performance and visual fidelity.

To run fracture segmentation algorithms in real-time (17 ms), we opted to use machine learning. Machine learning methods for mesh segmentation have been developed to decompose 3D objects into meaningful parts [52, 79, 82]. These approaches can segment a mesh, such as a car, into components such as wheels, body, hood, and windows. Fracture segmentation extends part segmentation by predicting how an object will break and assigning labels to the resulting fragments. While conceptually related to part segmentation, this task introduces additional challenges. In part segmentation, the mapping between geometry and labels is constant: the same region of a mesh is always associated with the same label. In contrast, fracture segmentation is inherently variable. The assignment of fracture labels depends not only on the geometry of the mesh but also on factors such as impact location, direction, and force. Consequently, fracture segmentation requires reasoning beyond the ground-truth mesh geometry, as the variability in fracture patterns complicates the use of consistent labelling.

Because we do not attempt to learn specific labels for predefined regions of the mesh, conventional segmentation methods struggle to learn the arbitrary labellings associated with our fracture patterns. Because of this, we decide against learning labellings for our fracture pieces. Instead, we learn a distance field over the surface of our mesh, with values indicating the distance to the nearest fracture. To retrieve the individual fracture pieces, we apply segmentation algorithms to the distance field in order to obtain the corresponding labels. Works like DeepFracture [24] similarly try to learn distance fields encoding fracture proximity, but find that their 3D approach is not capable of running at real-time speeds. Real-time models capable of predicting realistic fracture patterns are very enticing for time-essential use cases such as video games or prototyping interactive scenes. This is why we use a methodology similar to DeepFracture, but attempt to achieve real-time speeds by restricting the computation to the 2D mesh surface rather than the full 3D volume.

In video games, visual accuracy is about plausibility rather than physical correctness. When simulating fractures, the goal is not to produce real-world physics perfectly but to ensure the breaks and cracks appear in the right places and look believable to the player. To determine if our fractures look good enough for a video game setting, we determine their visual coherence. To do this we rely on results produced by our method and use these to establish a threshold which our method must meet in order to look believable.

This research aims to utilise impulse-dependent learned unsigned geodesic distance fields on the surface of meshes, that are then used to segment a mesh into pieces using segmentation methods. We aim to do this in a real-time fashion, such that it can be used in interactive and real-time settings, like video games.

1.1 Research Questions

Through our thesis, we aim to answer the following research question:

- Can we learn a distance field on the surface of a mesh representing impulse-dependent fracture patterns and retrieve fractures using segmentation methods in real-time and with good enough accuracy that it would be usable in a video game?

To address this research question, we first construct a dedicated dataset and implement a run-time framework to support experimentation. Within this framework we train and systematically evaluate multiple models using hyperparameter optimization. We further investigate and compare segmentation methods and determine their performance on the output of the tested learning models. To structure the investigation, the main research question is further divided into a set of sub-questions:

1. How do convolutional learning architectures compare with multilayer perceptrons for predicting impulse-dependent distance fields on mesh surfaces with respect to predictive accuracy and runtime speed?
2. When comparing popular surface-based mesh segmentation algorithms, which of the segmentation algorithms is fastest and most accurate at segmenting the distance fields?
3. What impact does the tessellation have on the performance of the model?

We aim to answer these research questions by constructing our own method and dataset. We then test the performance of our method by performing an experiment where we test the inference speed and accuracy of a DeltaConv and MLP-based approach. We further compare various segmentation methods, testing these on various meshes for both learning models and comparing their accuracy and speed.

1.2 Outline

This thesis is organized into six chapters. We begin by reviewing relevant literature in chapter 2. Then, chapter 3 goes over theoretical foundations by examining several critical papers in greater depth. Our proposed methodology is detailed in chapter 4, following an evaluation in chapter 5. Finally, we present our conclusions and discuss possible future work in chapter 6 and chapter 7.

Chapter 2

Related Work

This chapter discusses existing fracture simulation methods as well as existing methods for applying deep learning to 3D content. We also discuss segmentation algorithms, with a focus on graph-based image segmentation algorithms and other segmentation approaches that can be applied to meshes. For fracture simulations, we focus on types of brittle fracture simulations and distinguish between real-time and offline methods.

2.1 Brittle fracture simulations

Brittle fracture simulations refer to methods attempting to simulate brittle fractures. Brittle fractures are a type of fracture where the material fractures very quickly when put under stress, and very little deformation occurs [60]. In nature, these types of fractures are found in stone, glass, and ceramics [31].

The other main types of fractures are ductile fractures and fatigue fractures. Ductile fractures happen when the material first undergoes significant deformation before fracturing [60]. Fatigue fractures are other fracture types where small cracks form each time the material is put under stress, slowly building up over time until the material fractures [45].

We follow prior work that models brittle fracture as effectively instantaneous and neglects deformation, allowing the pre-fracture mesh to be reused without additional geometric transformations. Under this assumption, the fracturing process can be formulated as a labelling problem [24, 30].

Traditionally, brittle fracture simulations have been split into two categories: physics-based simulations and real-time simulations.

2.1.1 Physics-based simulations

There are many ways of simulating brittle fractures using stress-based physical simulations. These include Finite Element Methods (FEM) [14, 40, 47, 10], Boundary Element Methods (BEM) [21, 19, 20], Point-based methods [51, 16] and Spring-Mass based systems [46, 35].

Finite Element Methods

The finite element method (FEM) has long been used to calculate fractures in engineering applications [40]. FEM is a mesh-based approach that models fractures as a Lagrangian problem, solving equations for individual particles [14]. With advances in computing power, it became feasible to compute fractures in computer graphics using physically based systems. Early work applied FEM to simulate fractures by computing internal stresses and propagating cracks, combined with dynamic re-meshing to accurately map the fracture surface onto the mesh [47, 48]. Later, Delaunay triangulation-based re-meshing schemes were introduced to provide finer fracture detail [8].

An extended formulation, the extended finite element method (XFEM), was subsequently proposed, which eliminates the need for re-meshing [70]. By allowing cracks to propagate freely through the domain without modifying the original mesh, XFEM achieves greater precision than classical FEM. More recent work has combined XFEM with linear elastic fracture mechanics (LEFM) to further enhance accuracy while maintaining the advantage of avoiding re-meshing [10].

Boundary Element Methods

The most common mesh-based alternative to FEM is the boundary element method (BEM) [21], another solver for systems of partial differential equations. While FEM typically models fractures as a Lagrangian problem, BEM treats them as an Eulerian problem by focusing on the boundary of the mesh and performing calculations on this surface. Unlike FEM, BEM does not require re-meshing and operates only on the boundary of the problem [71]. Since discretisation is required only on the active regions of the boundary, BEM enables calculations inside the mesh without volumetric discretisation, providing higher accuracy compared to FEM [40].

This methodology has been extended in various directions. One line of work developed a high-resolution LEFM-based approach that allows separate control over crack initiation and propagation [19]. A related contribution introduced an approximate method that reduces the computational load of BEM by substituting certain cases with a rigid body simulation, enabling faster computation with minimal accuracy loss [20]. Another approach instead relied on solving the LEFM system directly using a boundary formulation; while limited to fractures initiated on the mesh, it avoids volumetric sampling and thus improves computational efficiency [86].

Point-based methods

Point-based methods can be divided into several groups, with methods based on the Material Point Method (MPM) being among the most widely used.

Early work introduced a meshless approach that addressed fracture location restrictions present in some FEM-based techniques. In this method, the mesh is represented by sampling points as nodes inside and on the surface, and fractures are initiated based on eigenvalue thresholds. During fracture propagation, node interactions and connectivity are updated, and once fracturing is complete, the results are mapped back onto the original mesh [51].

MPM-based methods have been coupled with a Lagrangian energy formulation on tetrahedral meshes to simulate both brittle and ductile fractures [74]. In this framework, material points serve as carriers of accumulated damage, providing indicators for when and where the underlying mesh will fracture once critical thresholds are exceeded. This approach enables fracture simulations of comparable quality to finite element methods (FEM), while substantially reducing the dependence on high-quality mesh discretizations.

More recent work has focused on combining MPM with continuum damage mechanics and rigid body models to achieve simulations that are significantly faster than BEM. These methods typically employ Voronoi diagram-based crack surface extraction, which avoids re-meshing but can result in a loss of fracture detail [16].

Spring-Mass based systems

Spring-mass based systems used to be popular as some of the first methods that attempted the physical modelling of fractures, thanks to their computational speed. But are rarely used in cases where physical accuracy is important, due to more realistic methods like FEM or BEM being available. Norton et al. were some of the first to attempt the use of spring-mass based systems and noted that, even though their objectives were successful, the realism of the method should be improved [46]. More recently, Levine et al. used spring-mass based systems in combination with peridynamics, allowing them to decouple a string's stiffness from its length and causing them to better capture non-local formulations and increasing stability [35]. However, Spring-mass based systems still share the limitations of other point-based methods, while also being less accurate and suffering from instability under more complex conditions.

The methods above have great realistic qualities, but are not used in real-time settings due to their computational costs or instabilities, often taking multiple minutes per fracture. Even when performing fracture on simplified versions of the original mesh, these methods would not perform in any real-time setting or would not guarantee consistent performance.

2.1.2 Real-time physics simulations

A variety of methods aim to approximate brittle fracture simulations in real-time applications. They usually do as much pre-computation as possible and sacrifice physical accuracy for speed. These methods can be split into three groups: Voronoi diagram-based methods [55, 43, 61, 28, 28], methods predefining the fracture pattern and number of pieces [69, 63] and other methods [18, 50].

Voronoi Diagram-based Methods

Voronoi diagram-based methods offer an efficient alternative to physics-based methods. First introduced by Raghavachary, this technique proposes a theoretical framework where cracks propagate along seeded Voronoi diagrams and are either driven by artistic or procedural control [55]. More recently, Müller et al. proposed a method focusing on large-scale destruction where meshes are represented by volumetric approximate convex decompositions, which support partial fracture patterns and fractures are defined by artists [43].

Comparatively, Schvartzman and Otaduy propose a method that leverages high-dimensional centroidal Voronoi diagrams to give more artist control and focus on robuster, more realistic fracture patterns [61]. Jung and Redenbach developed a model that synthesizes crack structures in 3D images. Extending the shortest path algorithm to 3D and formulating a minimum-weight surface problem on a randomly seeded cellular complex consisting of 3D Voronoi diagrams. Enabling them to construct realistic-looking cracks, which they then map back to real images of concrete to construct a synthetic crack dataset, to be used by machine learning methods [28].

Fracture Pre-computation

Some methods predefine fracture patterns and, during runtime, swap the model with a precomputed fractured version. One such approach introduced a collision-centered prescoring algorithm that allows fractures to propagate in a physically plausible manner [69]. Another line of work precomputes a set of fracture modes by solving a sparsified eigenvalue problem, which outputs the lowest-energy fractures for the mesh using modal analysis. At runtime, an impulse triggers the selection of one of these predefined fracture patterns, enabling real-time performance [63]. Similarly, a modal analysis approach was proposed that precomputes modal data and uses it at runtime to determine contact duration and forces between bodies, after which an energy-based algorithm propagates the fractures [18]. This method yields more realistic-looking fracture surfaces, though at the cost of less realistic fracture locations.

Precomputation is highly efficient, since most of the heavy computation is performed in advance, leaving minimal overhead at runtime. However, such methods lack physical flexibility, as the fractures are not dynamically generated based on material properties.

Other Real-Time Methods

Beyond precomputation, real-time fracture has also been explored through adaptations of physics-based models. One example is a FEM-based physics engine designed for games, which integrates fracture handling as part of a broader simulation system [50]. Instead of employing element splitting, this approach uses an artist-controlled splinter-based technique. Real-time performance is achieved by employing a tetrahedral FEM optimized for efficiency and robustness, though at the expense of physical accuracy.

2.2 Learning models

In this subsection we give an overview of machine learning and deep learning approaches that aim to decompose shapes into meaningful pieces or by implicitly learning shape representations.

2.2.1 Mesh-based methods

MeshCNN [22] is a convolutional neural network tailored for triangular meshes. It introduces specialized convolution and pooling layers that operate directly on mesh edges, leveraging the geodesic structure of the underlying surface. Convolutions are performed on an edge along with the four edges belonging to its incident triangles. Pooling is implemented through edge collapse operations that preserve the mesh topology by collapsing low feature magnitude edges first, allowing the network to select edges relevant to the task.

Primal-Dual Mesh Convolutional Neural Networks [42] represent a mesh using two graph structures: a primal graph, which encodes face adjacencies, and a dual graph, which encodes edge relationships. This allows them to apply convolutions on both faces (primal) and edges (dual). Their convolutions and pooling operations are conceptually related to those of MeshCNN, but due to their dual representations they achieve slightly better accuracy.

2.2.2 Voxel-based methods

Voxel-based methods represent 3D geometry on a regular grid, enabling the use of standard 3D convolutional architectures. VoxNet is one of the earliest systems in this category, interpreting LiDAR point clouds as voxel occupancy grids and applying a 3D CNN for object classification in real-time [41]. 3D ShapeNet follows a similar strategy, but is designed for CAD models and employs higher-resolution voxel grids in order to capture geometric detail. This dense representation, however, leads to substantial memory consumption due to the cubic scaling of voxel grids [81].

To address this limitation, OctNet introduces a data-dependent, unbalanced octree structure that allocates voxels only where geometric information is present, thereby reducing memory usage and computation while retaining detail where needed [58]. Other approaches rely on sparse voxel convolutions, such as those implemented in Minkowski CNNs. These networks also use sparsity in voxelized data and have been applied to both 3D shape analysis and spatio-temporal encoding of 4D data [11]. Despite these advances, most sparse-voxel networks still struggle to achieve truly real-time performance at high resolutions.

Voxel grids have also been used in generative modeling. For example, 3D-GANs operate directly on voxel occupancy grids to learn distributions of 3D shapes and synthesize new geometry [80]. While effective, these methods remain constrained by the resolution and memory limits inherent to voxel-based representations.

2.2.3 Point cloud-based methods

Point clouds are discrete unordered sets of 3D positions representing objects in space. Machine learning methods used to struggle to work with point clouds, due to their unordered nature. That is why previously people used to transform point-cloud information to voxel data.

Some of the most well-known point cloud-based learning methodologies are PointNet [52] and its derivatives like PointNet++ [53] and PointNext [54]. Before PointNet, it was common to convert point clouds into voxels, causing large overheads. Qi et al. solved this by introducing an order-invariant way of learning point clouds using max pooling, while also ensuring the architecture works for both local and global tasks, like part segmentation and classification [52]. This was then improved again in PointNet++, which focussed on adding hierarchical feature learning, increasing accuracy for fine-grain details [53].

Compared to PointNet, which uses supervised learning, there are also unsupervised techniques that try to learn a representation of point clouds. FoldingNet uses auto-encoders to learn an encoding from point clouds to 2D grids [83]. Nazir et al. use a combination of AutoEncoders and a chamfer distance-based loss to create a model that has improved generalization capabilities [44]. Xie et al. were the first to use contrastive learning in a 3D spatial point cloud sense. They show that pre-training any point cloud encoder, from PointNet++ to FoldingNet, works well [82].

DeltaConv is a model designed to construct convolutional layers that operate directly on surfaces derived from point clouds. The network is divided into scalar and vector streams, which are connected through geometric vector-based operators. This design enables the model to explicitly represent and process directional information [79].

2.2.4 Distance field based methods

DeepSDF uses learned signed distance fields to represent shapes by training an autoencoder to capture the latent space of shapes [49]. MetaSDF extends this idea by framing shape space learning as a meta-learning problem, using gradient-based meta-learning algorithms that allow for more flexible shape reconstruction [67]. In contrast, NeuralUDF employs unsigned distance functions, allowing them to also reconstruct inner structures, instead of only the outside of shapes [9].

In our work, we similarly employ unsigned distance fields, but not to represent the geometry of objects. Instead, following the approach of DeepFracture [24], we use distance fields to represent and model mesh fractures. Our method draws inspiration from these existing distance field-based techniques, but we apply them in a different context to represent fracture patterns.

2.2.5 DeepFracture

DeepFracture is an approach working directly with meshes that attempts to learn 3D fractures of a single mesh using a latent vector-controlled approach. Learning so-called 3D Geometrically-Segmented Signed Distance Functions representing a mesh and then using a segmentation algorithm on the learned distance functions to separate the mesh into pieces [24]. Due to their segmentation algorithm and model complexity, the approach is not real-time.

Recently, Kim et al. have attempted to model brittle fracture as an unordered segmentation of a point cloud and tried leveraging a deep neural network trained on a physics-based dataset to predict the structural weakness of objects [30]. They have, however, noted that their approach is limited by datasets and the inadequacy of traditional segmentation benchmarks. Compared to DeepFracture, they directly learn the labelling, instead of a distance field having to be segmented and also passing the desired number of different labels, being less realistic, but providing more artist control.

Another deep learning based fracture methodology is StressNet, a network which uses a temporal independent convolutional neural network of fractures combined with bidirectional long short-term memory to model temporal dynamics. They use this methodology to predict evolution of stress over time, but never reach real-time performance [75].

DeepFracture demonstrates the potential of combining distance fields with machine learning to generate realistic fracture patterns. However, the method is not capable of real-time execution. Building on these ideas, our work focuses on adapting this approach to achieve real-time performance.

2.3 Segmentation methods

Various approaches have been proposed for mesh segmentation. As summarized in [59], existing methods can be broadly grouped into region-growing approaches, watershed methods, iterative clustering techniques, hierarchical clustering, and boundary-based segmentation.

Region Growing

Seeded region-growing was introduced in [1] for the segmentation of intensity images, where predefined seeds guide the expansion of homogeneous regions and thus control the formation of region boundaries.

Region-growing ideas have since been adapted to 3D mesh analysis. In particular, [34] employ Markov Random Fields to smooth mesh attributes and produce spatially coherent vertex clusters, improving the robustness of subsequent region-growing–based segmentation.

Watershed

Watershed segmentation uses a flooding process that starts at a local minimum and incrementally floods a region, stopping once it finds a different flooding process from another local minimum [23]. Mangan and Whitaker were the first to extend the watershed segmentation algorithm from images to meshes, by considering all connected vertices to be the neighbours, instead of adjacent pixels [39]. They used watershed to segment a mesh based on per-vertex curvature values, also being the first to introduce a merging step, preventing over-segmentation.

iterative clustering

Lloyd's k-means algorithm [38] represents one of the earliest iterative clustering approaches, minimizing within-cluster variance. Variants of iterative clustering have been adapted for mesh segmentation.

Liu et al. [36] incorporate perceptual cues, which they capture using eigenvalue-based geometric analysis, to segment meshes efficiently. Lai et al. [32] further introduce a random-walk formulation that enables meaningful mesh segmentation at interactive speeds. Together, these works illustrate how iterative clustering ideas extend naturally to surface segmentation tasks.

hierarchical clustering

Classical hierarchical clustering methods, such as Ward's minimum-variance approach [77], iteratively merge or split clusters to form a multilevel grouping structure.

This hierarchical principle has been applied to mesh segmentation in various forms. Attene et al. [5] segment meshes by clustering mesh primitives detected from local geometric features. Shapira et al. [66] instead use the shape diameter function and interpret segmentation as a skeletonization problem, yielding consistent part decompositions across shapes. These methods highlight how hierarchical strategies can capture multi-scale organization in geometric data.

Boundary-based methods

Boundary-based segmentation approaches focus on identifying curves on the surface that delineate meaningful parts. Lee et al. [84] introduce an "intelligent scissoring" technique that employs 3D snakes to detect and find defining shape contours, achieving speeds of up to multiple minutes per segmentation. Zheng et al. [85] propose an interactive decomposition framework in which users place strokes on the mesh; the system infers the most natural cut by aggregating information from multiple user-provided cues. Such methods emphasize boundary quality and user guidance, but suffer from long computation times.

2.4 Datasets

There are not many datasets containing fractured meshes. The two main datasets are Breaking Bad [62] and Fantastic Breaks [33]. The Breaking Bad fracture dataset is generated using the similarly named Breaking Good methodology [63], also developed by Sellán et al. [62]. Their dataset consists of multiple synthetic fractures of a variety of object groups and their original mesh counterparts. The Fantastic Breaks dataset by Lamb et al. was created by hand annotating fractures on 3D scanned meshes of real-world objects. These meshes were hand-cleaned and paired with synthetically generated restoration meshes that aim to resemble the original object [33]. Both of these datasets are not suitable for our needs, since the goal of our research is to learn the fracture patterns of meshes based on impulse position and direction. Neither of the datasets provides this information and thus we decided to create our own dataset.

Chapter 3

Preliminaries

This chapter provides a more detailed background into existing methods heavily utilized in our work. Our main focus lies on DeltaConv [79], a convolutional graph-based learning model that works on point clouds and meshes.

3.1 DeltaConv

Leaning on point-clouds or meshes requires the use of direction-dependent operators allowing for the capture of local geometric structures such as ridges or edges. Many existing learning models are not capable of fully capturing such geometric features, as they rely on isotropic operators, which are directionless. In image processing, directional operators are easily constructed, since the image grid has a fixed coordinate system: its pixels. This allows for simple definitions of directional filters on images.

However, DeltaConv is designed to work on mesh surfaces where no fixed coordinate system exists. The local neighbourhood is therefore rotationally ambiguous and we are unable to consistently construct directional filters. Conventional graph-convolutional models use Laplacians to construct their convolutions, exacerbating this issue, as Laplacians are isotropic and therefore directionless. This limitation motivates the need for an anisotropic surface-based convolutional operator. DeltaConv addresses this by constructing such a surface-based convolutional filter capable of capturing directional information.

3.1.1 The Core Idea of DeltaConv

DeltaConv, like other graph-convolutional models, is formulated in terms of Laplacian operators. The standard Laplacian is isotropic, but can be written as the divergence of the gradient [78]. This allows us to access a vector-based intermediate representation. Combining these representations using MLPs together with non-linearities is what allows for anisotropic behaviour.

The core idea of DeltaConv is to learn intrinsic operators directly on the surface. In addition to the divergence and gradient, they also learn to combine the co-gradient, curl, and Hodge-Laplacian. Due to the operators acting on vectors and scalars, the authors formally define a vector stream containing vector information and a scalar stream for scalar information, with some operators transferring information between the streams.

For each layer, the outputs of the operators at every vertex are passed, together with the vertex features, to small MLPs that learn how to combine these operator responses. All operators and MLPs are defined in a coordinate-independent manner. This removes the need for a fixed reference frame and guarantees that the resulting anisotropic filters are invariant to rotations of the surface. This ensures that the model produces the same output for each vertex, irrespective of the chosen local basis or any other rotations.

3.1.2 The DeltaConv Operator

The local neighbourhood of each vertex is defined using the K-Nearest neighbours selected from the mesh or pointcloud. This neighbourhood is what defines the local surface patch on which the operators act per-vertex. Each vertex is associated with C features, yielding a total of $N \times C$ features for a pointcloud of size N.

The feature at point i in layer l is defined as $x_i^l \in \mathbb{R}^{C_l}$ for each vertex in the scalar stream and the collection of all scalars is called X. In the vector stream each feature is represented as a pair (α_i, β_i) compared to a local tangent plane, and all vectors are stored in V. The choice of basis is irrelevant, as long as it is orthogonal to the surface normal.

Property	**Scalar→Scalar**	**Scalar→Vector**	**Vector→Scalar**	**Vector→Vector**
Operators	Pointwise MLP + max-pooling	Gradient G, Co-gradient JG	Divergence D, Curl $-DJ$, Norm $\|\|\cdot\|\|$	Identity I, Hodge-Laplacian $L_H = -(GD - JGDJ)$
Input	$X \in \mathbb{R}^{N \times C}$	$X \in \mathbb{R}^{N \times C}$	$V \in \mathbb{R}^{2N \times C}$	$V \in \mathbb{R}^{2N \times C}$
Output	$x_i' \in \mathbb{R}^C$	$(GX)_i \in \mathbb{R}^{2C}$	$(DV)_i$, $(-DJV)_i$, $\|\|v_i\|\| \in \mathbb{R}^C$	$(L_H V)_i \in \mathbb{R}^{2C}$
Role	Aggregate scalar features	Encode directional change	Extract sinks, rotation, magnitude	Mix vectors in a local neighbourhood

Table 3.1: DeltaConv operators

Table 3.1 shows the different operators in DeltaConv that can be applied to both the vector and scalar stream. For a more in-depth explanation on how these operators interact, the reader is referred to the original DeltaConv paper by Wiersma et al..

Learning Anisotropic Filters

Now that all operators have been defined, we can finally see how DeltaConv learns anisotropic Filters. All features for each stream are concatenated and passed into MLPs together with the output of the operators defined before and passed to the next layers. This is done as follows: $v_i' = h_{\Theta_0}(v_i, (GX)_i, (L_H V)_i)$ for the vector stream. We can see that the Vector MLP takes as input: the vector features (or identity I), the gradient of the scalar stream, and the Hodge-Laplacian output of the vector stream.

The Scalar pass looks as follows: $x'_i = h_{\Theta_1}(x_i, (DV')_i, (-DJV')_i, ||v'_i||) + \max_{j \in \mathcal{N}(i)} h_{\Theta_2}(x_j)$ and takes as input: the scalar features, the divergence of the vector stream, the curl of the vector stream, the norm of the vector stream, as well as the maximum pooling of the local neighbourhood. Here $\mathcal{N}(i)$ represents the local neighbourhood of vertex i.

3.1.3 DeltaConv Segmentation Architecture

The DeltaConv architecture that presents both classification and segmentation models was inspired by DGCNN [76]. The two designs are largely similar, with the main distinction being the extended segmentation head. Since our work builds on a modified segmentation model, the discussion that follows will concentrate on the same.

The authors report that the most effective segmentation architecture is composed of three convolutional layers. These layers are structured as: $\mathrm{Conv}(3, 64, 64)$, $\mathrm{Conv}(64, 128, 128)$, $\mathrm{Conv}(128, 256, 256)$, where the first argument corresponds to the number of input channels (in this case, three).

Following these layers, the outputs are concatenated into a feature vector of dimension 488 for each point. This vector is subsequently projected to the embedding space of size 1024 through a global MLP. In the original approach, a one-hot encoded categorical vector is appended to this global embedding before being passed to a segmentation-specific head. In our implementation, however, we omit the categorical vector and employ a slightly modified segmentation head to change the model to a regression task, which we discuss in subsection 4.4.2. For the segmentation task, the authors optimize the cross-entropy loss.

Chapter 4

Methodology

This chapter presents a detailed description of the proposed method. We begin with an overview of the approach in section 4.1. The generation of ground-truth fracture data and the construction of the dataset are described in section 4.2.

The formulation of the fracturing process as a learning problem is introduced in section 4.3. The learning models employed, along with the architectural adaptations made to meet the specific requirements of this work, are presented in section 4.4. section 4.5 presents our definition of real-time.

Subsequently, section 4.6 details the inference pipeline and the segmentation methods used. The evaluation metrics employed to assess model performance are discussed in section 4.7.

Finally, implementation details are provided in section 4.8.

4.1 Overview

Our method is inspired by DeepFracture [24], which we introduced in subsection 2.2.5. Whereas DeepFracture learns a volumetric distance field to the nearest fracture boundary, our method focuses on learning a surface-based distance field to the nearest fracture boundary on the mesh surface. This restriction to surface evaluation should enable faster computation times. Similar to DeepFracture, our approach involves learning a distance field to the fracture boundary that is conditioned on impact location and direction. Limiting our model to learning the impulse-dependent distance field for one mesh at a time is consistent with the approach employed by DeepFracture.

A visualization of our pipeline can be seen in Figure 4.1 and consists of two main parts: dataset generation and inference.

During dataset generation, multiple fracture simulations are computed for the given mesh under varying impact configurations. The resulting fracture boundaries are processed to compute a surface-based distance field for each fracture, which is stored together with the impulse information.

During inference, the trained model is provided with the mesh and a specific impulse. The model predicts the corresponding surface distance field on the mesh, which is subsequently passed to a segmentation algorithm to produce the final fracture boundary prediction.

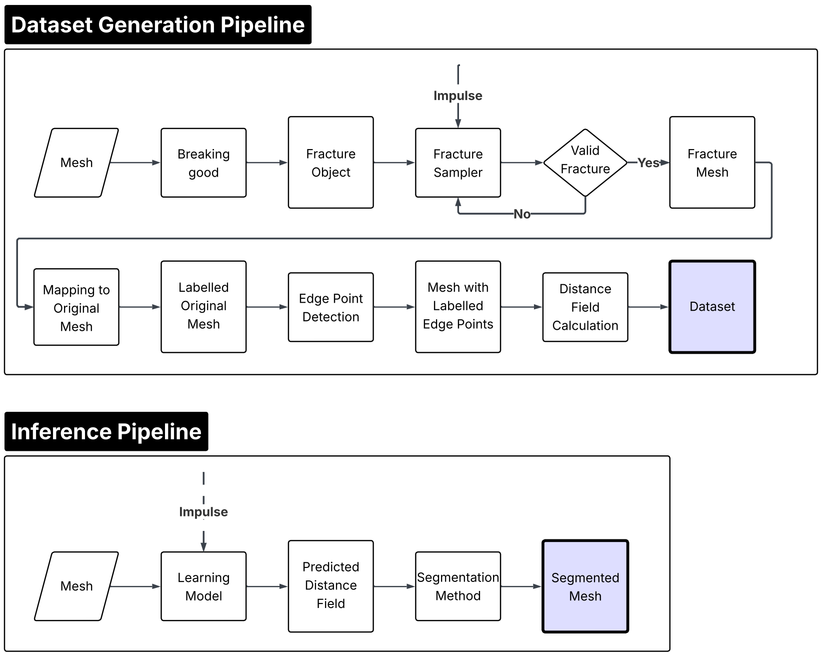


Figure 4.1: Overview of the pipeline

4.2 Dataset Generation

4.2.1 Input

The input to the dataset generation pipeline is a single mesh M_o. The mesh is required to be watertight, and its vertices and edges must form a single connected, navigable graph.

4.2.2 Breaking Good

The fracture generator we use is Breaking Good [63]. Breaking Good identifies plausible fracture modes of a mesh by solving a sparsified eigenvalue problem, from which the lowest-energy fracture modes are extracted. These modes are precomputed for a given shape, and at runtime the impact is projected onto the modes allowing for impact-dependent fracture patterns without the need for online crack propagation simulations.

Breaking Good allows us to generate datasets within a few hours, whereas more physically accurate methods may require several days of computation [40, 47]. While Breaking Good does not model fracture physics in full detail, it provides a controlled and efficient testbed for evaluating whether our approach can learn meaningful, impulse-dependent fracture patterns. Demonstrating successful approximation in this setting motivates future investigation into learning more physically realistic fracture behaviors.

An additional advantage of Breaking Good is its inherent impulse-dependent fracture generation, which is essential for our experiments. Notably, both Breaking Good and our approach rely on precomputation to achieve real-time performance at inference time, with the primary distinction being our use of a learned model to represent fracture behavior.

4.2.3 Computing Fractures

The dataset generation process begins by loading a single mesh M_o, consisting of vertices, edges, and vertex normals, and defining the desired dataset size n. If the mesh does not contain normals, these are computed as a preprocessing step. The mesh is then passed to the Breaking Good pre-computation module [63], which computes a fracture modes object, from which we retrieve the fracture patterns.

For each fracture $m \in \{1,\ldots,n\}$, a random point p_i is sampled on the mesh surface and used as the impact position. The impact direction d_i is defined as the inverse normal at p_i where $i \in \{1,\ldots,\text{number of vertices of } M_o\}$. The impact position and direction are then passed to the run-time component of Breaking Good, which produces a fracture labelling f_i. This labelling assigns a numerical label to each vertex of M_o, indicating its fracture piece membership. As in the original Breaking Good implementation, the fracture generation is repeated with newly sampled impulses until a valid fracture is produced.

To determine whether a fracture labelling is valid, we apply three criteria. First, all fracture pieces must satisfy a minimum size constraint, as very small pieces are likely to be artifacts. Second, the combination of impulse parameters and fracture labelling must be unique within the dataset. Finally, the labelling must contain at least two distinct labels, ensuring that an actual fracture has occurred. Samples that fail any of these checks are discarded and regenerated using a newly sampled impulse.

Ideally, impact positions would be uniformly distributed over the mesh surface. However, discarding invalid fracture samples introduces a bias in the dataset: structurally weaker regions of the mesh tend to be overrepresented, while stronger regions are underrepresented due to the absence of fractures. The implications of this bias are discussed further in chapter 7.

4.2.4 Computing the Unsigned Distance Field

The output of the fracture generator is a variety of labellings of fractured meshes. But instead of using these labels directly, we apply additional post-processing steps on them before we use them.

Mapping back to the original mesh

To improve performance, we follow the approach of Breaking Good and perform fracture labeling on a downsampled version of the original mesh, denoted M_d, rather than on the original mesh M_o. Let $v_o = \{v_o^i\}$ denote the set of vertices of the original mesh and $v_d = \{v_d^j\}$ the set of vertices of the downsampled mesh, where indices i and j enumerate vertices in the respective meshes. Because the downsampling process does not guarantee identical scale or alignment between the two meshes, a mapping between the vertex sets v_d and v_o is required.

We approximate this mapping by assuming that downsampling preserves the overall shape of the mesh while introducing only a global translation and uniform scaling of vertex positions. Proceeding under this assumption, we compute the axis-aligned bounding boxes of both meshes and apply a scaling and translation that aligns the bounding box of M_d with that of M_o. This transformation is applied to all vertices v_d.

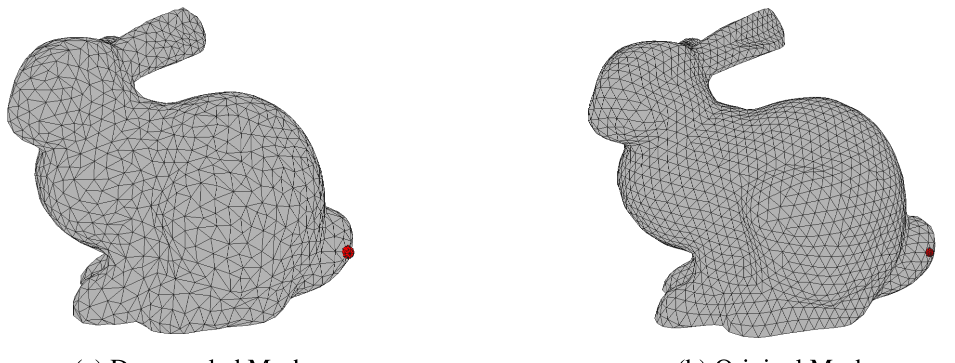
Let $f_d = \{f_d^j\}$ denote the fracture labeling produced by Breaking Good on the downsampled mesh, where each vertex v_d^j is assigned a fracture label f_d^j. Since downsampling may introduce small geometric distortions, fracture labels cannot be transferred directly via the rescaled mesh. Instead, for each vertex $v_o^i \in v_o$, we identify the index

$$k = \arg\min_j \left(\text{distance}(v_o^i, v_d^j)\right) \tag{4.1}$$

corresponding to the closest vertex in the transformed downsampled mesh. The fracture label of v_o^i is then assigned as $f_o^i = f_d^k$, where $f_o = \{f_o^i\}$ denotes the resulting fracture labeling on the original mesh.

Since impact positions are initially sampled on the downsampled mesh M_d, each impact position is first transformed using the same scaling and translation applied to the vertices v_d. The transformed point is then projected onto the surface of the original mesh M_o by selecting the closest surface point, yielding the final impact position associated with fracture instance m. A visual comparison between the original mesh and the downsampled mesh, as well as the applied rescaling, is shown in Figure 4.2.

For each fracture instance m in our dataset of M_o, the procedure described above is applied to generate a single dataset sample. We denote the resulting fracture labelling on the original mesh by F_m, which assigns a fracture label to every vertex of M_o. Together with F_m, we store the corresponding impact position p_m and impact direction d_m.



(a) Downscaled Mesh (b) Original Mesh

Figure 4.2: Comparison between the original mesh and downscaled mesh, the red dot shows the impact point

Edge Points

For every generated fracture, we define two types of vertices: edge vertices and non-edge vertices. Edge vertices lie on the boundary between fracture pieces and are characterized by having at least one neighbouring vertex with a different ground-truth label. In contrast, non-edge vertices have no neighbours with different ground-truth labels.

Edge set E is defined in Equation 4.2 by considering all vertices $v_i \in M_o$ and selecting only the vertices where the neighbour $N(v_i)$ has a different ground-truth label f_i.

$$E = \{v_i \in v \mid \exists v_j \in N(v_i) : f_i \neq f_j\} \tag{4.2}$$

A visualization of the edge set E is shown in Figure 4.3b, whereas Figure 4.3a shows the ground-truth labels associated with the edge set.

Unsigned Distance Field

We compute an unsigned distance field (UDF) over the surface of the mesh to encode the proximity of each vertex to the fracture boundary. To calculate the distance field, at each vertex $v_i \in M_o$, we store the geodesic distance to the nearest fracture edge v_e, represented by the points in the edge set E.

Formally, the UDF value at vertex v_i is defined as

$$\mathrm{UDF}_i = \min_{v_e \in E} \mathrm{dist}(v_i, v_e) \tag{4.3}$$

where dist(.,.) denotes the geodesic distance along the mesh surface.

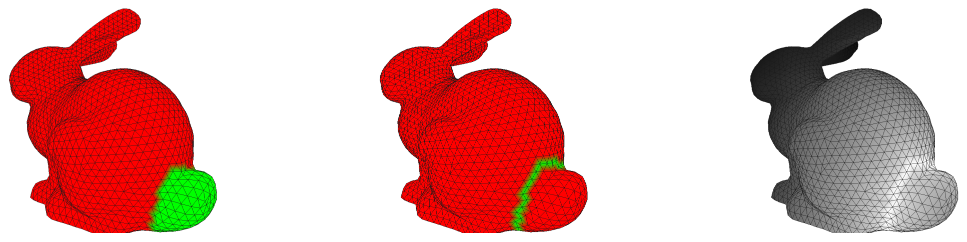
We use geodesic distance rather than Euclidean distance to ensure that the gradients of the distance field are better aligned along the surface of the mesh. As distances are solely defined on the surface and no inside or outside exists, a signed distance field is not applicable, therefore we use an unsigned distance field.

A visualization of this concept is provided in Figure 4.3. Specifically, Figure 4.3a shows the ground-truth labels of the fracture, while Figure 4.3b highlights points on the fracture boundary and inside edge set E. Finally, Figure 4.3c presents the geodesic distance field on the mesh, indicating the geodesic distance of each vertex to the edge set.

4.3 Learning framework

We formulate fracture prediction as a regression problem in which a model learns to map an impact impulse and a mesh surface to an unsigned distance field encoding the proximity of each vertex to the fracture boundary. The learning framework is defined as a function shown in Equation 4.4 that takes as input an impulse I, consisting of an impact position and impact direction, together with the vertex positions $\mathbf{v}_1, \ldots, \mathbf{v}_n$ of a mesh M_o, and outputs a predicted unsigned distance field over the mesh. The mesh connectivity is not provided explicitly to the model.

$$(UDF_1, UDF_2, \ldots, UDF_n) = f(\mathrm{I}, \mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n) \tag{4.4}$$



(a) Ground-truth fracture labels on the Stanford Bunny mesh.

(b) Edge vertices (green) forming the fracture boundary set E; non-edge vertices are shown in red.

(c) Unsigned geodesic distance field to the fracture boundary set E, with darker shades indicating larger distances.

Figure 4.3: Computation of the unsigned geodesic distance field. From left to right: ground-truth fracture labeling, extracted fracture boundary vertices forming the edge set E, and the resulting unsigned geodesic distance field defined on the mesh surface.

During training, we use a custom loss function that emphasizes accuracy near fracture boundaries. These regions correspond to small unsigned distance field values and are critical for the subsequent segmentation step, whereas errors farther away from the boundaries have a limited effect on the final fracture prediction. Let y_i denote the ground-truth unsigned distance field value at vertex i and $\hat{y}_i$ the predicted value. We define the adapted mean absolute error (AMAE) loss as shown in Equation 4.5.

$$\mathscr{L} = \sum_i \frac{|y_i - \hat{y}_i|}{1 + \alpha |y_i|} \tag{4.5}$$

The variable α controls the strength of the weighting. This formulation penalizes errors near fracture boundaries more strongly, while progressively reducing the contribution of errors at larger distances. This is a worthwhile trade-off since these areas are critical for segmentation performance.

4.4 Network architectures

Although the learning models are treated as black boxes in our framework, the choice of network architecture has a significant impact on both segmentation accuracy and computational efficiency. As discussed in section 2.2, prior work has shown that convolutional models outperform MLPs for mesh and point cloud segmentation tasks on varying geometries. Motivated by these findings, we aim to assess whether the same performance benefits also apply to our task. In our setting, the mesh itself remains fixed, while the applied impulse varies to produce different fracture patterns. Consequently, our primary interest lies in examining how the use of (graph-)convolutional layers affects both segmentation accuracy and computational efficiency.

We compare a simple multi-layer perceptron (MLP) baseline with a graph-convolutional architecture based on DeltaConv, which was selected for its use of anisotropic operators that are well suited for capturing geometric details. We investigate whether these properties translate to improved performance for fracture pattern prediction.

4.4.1 MLP architecture

The MLP models were implemented in PyTorch, with varying depths and widths explored by adjusting both the number of layers and the number of nodes per layer. All hidden layers share the same dimensionality within a configuration. Each layer is followed by Layer Normalization and a ReLU activation.

For each vertex, the network receives the 3D coordinates concatenated with the impact impulse parameters as input, and outputs a single unsigned distance field value. The network is therefore applied independently to all vertices of the mesh.

4.4.2 DeltaConv architecture

We adapt the DeltaConv segmentation architecture described in subsection 3.1.3 to perform regression of unsigned distance field values. The main difference lies in the output: while the original segmentation architecture produces $N \times C$ outputs, representing the probability of each node belonging to class C, our variant predicts N continuous values, corresponding to the distance field at each point. An overview of the adapted architecture is provided in Figure 4.4.

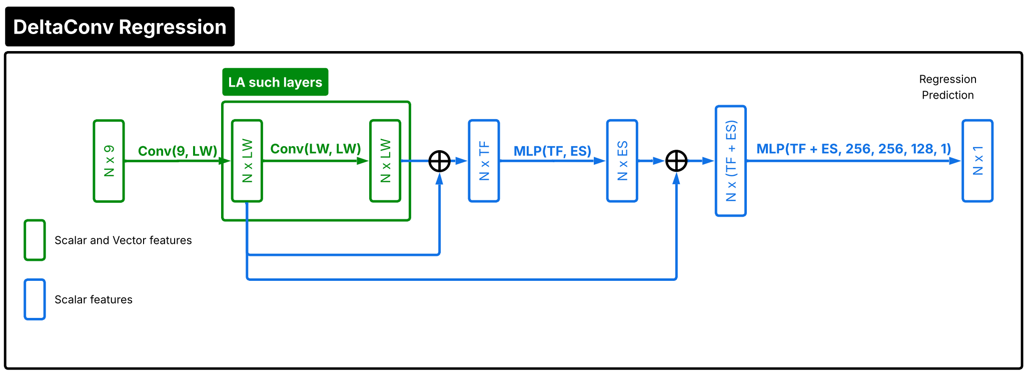


Figure 4.4: DeltaConv regression architecture. LW = convolutional layer width, LA = number of convolutional layers, ES = embedding size, and TF = total number of concatenated convolutional features (i.e., LA * LW).

The architecture is parameterized by six hyperparameters controlling the depth, neighborhood size, and feature dimensionality, kernel of the convolutional blocks:

1. **Number of convolutional layers** – specifies the total depth of convolutional stages.

2. **Convolutional layer size** – defines the number of output channels of each convolutional layer; unlike the segmentation model, we employ a constant size across all layers.

3. **MLP depth** – determines the number of fully connected layers used within each convolutional block.

4. **Embedding size** – sets the dimensionality of the embedding space, consistent with the original design.

5. **Number of neighbours** – controls the neighborhood size considered in each DeltaConv layer.

6. **Gradient kernel width** – governs the spatial extent of gradient computation, e.g., restricting to first- or second-order neighbours.

Since our task does not involve categorical classification, the categorical vector used in the original segmentation architecture was omitted. Apart from the reduced output dimensionality, the inference head remains similar to that of the segmentation model, with one modification: leaky ReLU activations were replaced by standard ReLU, which yielded improved performance in our experiments.

We further modified DeltaConv to enable partial precomputation, thereby reducing inference time. Since the same mesh is used throughout, the k-nearest neighbor search and the corresponding matrix construction could be performed in advance and reused during inference.

4.5 Real-Time Constraint

Electronic Arts specifies that Battlefield 6 requires a minimum frame rate of 30 fps for basic playability, with 144 fps recommended for optimal experience [4]. This maps to per-frame budgets of 34 ms and 7 ms respectively. Empirical studies in interactive systems show that below certain frame-rate thresholds, performance and responsiveness degrade. For example, in the domain of first-person shooters, Claypool et al. found significant drops in aiming accuracy and user satisfaction when frame rates fall below 30 fps [12]. In related HCI experiments, moving-target selection tasks performed 14 % better at 60 fps than at 30 fps [27]. Moreover, studies of frame-rate variation indicate that jitter or frame time spikes negatively impact perceived quality even if the average frame rate remains high [37]. Hence, while 30 fps may be treated as a lower bound in practice, it is important to design fracture techniques well above that threshold, not only to avoid performance degradation, but also to maintain consistency and responsiveness under worst-case conditions.

If the target performance is to sustain a minimum of 30 fps during fracture events, while the game ordinarily operates at 60 fps, the available rendering budget changes accordingly. Under normal conditions, each frame is budgeted approximately 17 ms for rendering. During a fracture event, we must still maintain our minimum threshold of 30 fps, which corresponds to a maximum frame time of 34 ms. Thus, the fracture algorithm must complete within this 17 ms window to maintain real-time performance and prevent the overall frame rate from dropping below the 30 fps threshold.

4.6 Inference

During inference, the trained model is given the entire mesh together with a contact position and direction and predicts an unsigned distance field value for each vertex. The mesh and the predicted UDF are fed into the segmentation method, which converts the distance field into a discrete fracture labeling. Based on groupings of segmentation methods made by previous work [64, 59], we evaluate the following segmentation methods: Region Growing, Watershed, K-Means, Hierarchical Clustering and Felzenszwalb segmentation.

4.6.1 Segmentation methods

In this section, we describe each segmentation method in detail and outline any adaptations needed to integrate them into our pipeline.

Watershed Segmentation

As DeepFracture applies morphological flooding algorithms to segment their volumetric distance field, watershed segmentation is a natural candidate for segmenting surface-based unsigned distance fields [7, 24].

Watershed segmentation aims to accomplish a segmentation task by viewing it as a flooding problem. It treats UDF intensity values as a topographic surface, identifying basins and watershed lines to segment objects. It is inspired by flooding where it fills basins, which in practice are local minima, until boundaries are formed.

While classical watershed algorithms grow regions from local minima, we initialize flooding from local maxima, which correspond to regions far from fracture boundaries, and propagate labels toward the boundaries. Watershed segmentation has a user-controlled parameter called threshold. This parameter will control the minimum value for neighbouring clusters to be merged. A high value will cause oversegmentation, whereas a low value will produce undersegmentation.

The watershed algorithm starts by finding local optima and queuing these vertices in a priority queue while assigning them all unique labels. Then it iteratively pops a vertex from the queue and looks at its neighbours. If the neighbour has not been visited yet, we assign it the same label as the current vertex. If the neighbour has been visited and both the neighbour and the current vertex are above the threshold while having different labels, we merge all vertices with either of the labels into a new, larger group. This process can be seen in more detail in Algorithm 1

The only difference between our watershed segmentation and the default watershed segmentation is that we merge connected groups while flooding. This is usually done in a separate loop after the flooding algorithm has finished, but in our case, it is possible to apply these operations simultaneously in a single loop.

Algorithm 1 Watershed Segmentation

```
1:  Initialize all nodes as unvisited
2:  Assign unique labels to local maxima
3:  Enqueue all local maxima
4:  while the queue is not empty do
5:      Dequeue a node
6:      Mark the node as visited
7:      for each neighbor of the node do
8:          if neighbor is unvisited and has lower or equal UDF then
9:              Assign the neighbor the current node's label
10:             Enqueue the neighbor
11:         end if
12:         if neighbor has a different label and either UDF exceeds a threshold then
13:             Merge the two labels
14:         end if
15:     end for
16: end while
17: Return the final labels for all nodes
```

Region growing

Region Growing is a seeded segmentation technique that iteratively expands regions from an initial set of vertices [1]. In our framework we seed these vertices similar to watershed, by picking all local maximal UDF values as initial seeds.

Starting from these seeds, regions are grown by repeatedly assigning unlabeled neighbours to the regions most popular around them. Vertices are processed in decreasing UDF order, ensuring regions expand from the fracture interior towards the fracture boundaries. Post-processing steps similar to watershed are applied to merge small clusters.

The complete region growing procedure is summarized in Algorithm 2.

Hierarchical Clustering

Hierarchical agglomerative clustering [77] is a bottom-up clustering method in which each vertex initially forms its own cluster and clusters are iteratively merged based on a pairwise similarity measure. In our setting, the mesh connectivity defines the adjacency graph, and edge weights are derived from the predicted unsigned distance field.

We assign each mesh edge (x,y) a score equal to the sum $\mathrm{UDF}(x)+\mathrm{UDF}(y)$ and then sort all edges in decreasing order. Edges connected far from fracture boundaries therefore appear first, and edges crossing fractures appear last. Clusters are then iteratively merged until the edge sum falls below a certain threshold, after which the process terminates while preventing merges across fracture boundaries. This procedure is summarized in Algorithm 3.

This causes large interior pieces to form early in the process, while pieces near fracture boundaries remain separated.

Algorithm 2 Seeded Region Growing

```
1: Initialize all nodes as unvisited
2: Initialize all nodes with label $-1$ (unlabeled)
3: Select $K$ seed nodes and assign them unique labels
4: Initialize region statistics for each label (e.g., mean UDF per region)
5: Enqueue all seed nodes
6: while the queue is not empty do
7:     Dequeue a node $u$
8:     Mark $u$ as visited
9:     for each neighbor $v$ of $u$ do
10:        if UDF of $v \geq 0.1$ AND UDF of $u \geq 0.1$ then
11:            Assign $v$ the label $L(u)$
12:            Update region statistics for label $L(u)$
13:            Enqueue $v$
14:        end if
15:    end for
16: end while
17: Return the final labels for all nodes
```

A limitation of this approach is the forming of many small clusters near the fracture boundary, caused by the conservative merging criterion. These small fragments have little influence on the global fracture geometry and could probably be removed through post-processing steps, but these steps have not been implemented in the current pipeline. To mitigate this, we select conservative parameter values when applying the method.

Algorithm 3 Hierarchical Clustering

```
1: Initialize each vertex with a unique label
2: Sort all edges in decreasing order of the sum of UDF values of their endpoints
3: for each edge $(x, y)$ in the sorted edge list do
4:     if UDF($x$)+ UDF($y$) is below threshold then
5:         break
6:     end if
7:     if $x$ and $y$ belong to different clusters then
8:         Merge clusters of $x$ and $y$
9:     end if
10: end for
11: return cluster labels
```

K-Means

From the family of iterative clustering algorithms, we selected K-Means [38] due to its widespread popularity. We applied the K-Means algorithm to our data, using each point's

x, y, z, and UDF values as features.

The K-Means algorithm attempts to segment its input data in k different clusters, with each data point belonging to the cluster with the nearest mean. After a data point has been assigned to a cluster, the cluster mean is updated, and the process is repeated until convergence. The K-Means algorithm can be seen in Algorithm 4.

Algorithm 4 K-Means Clustering

```
 1: Initialize $K$ cluster centers $m_1, m_2, \dots, m_K$
 2: repeat
 3:     for each item $t_i$ in the dataset do
 4:         Assign $t_i$ to the cluster with the closest mean
 5:     end for
 6:     for each cluster do
 7:         Update the cluster mean based on assigned points
 8:     end for
 9: until convergence criteria is met
10: return final cluster assignments
```

Felzenszwalb segmentation

Felzenszwalb segmentation [17] is a graph-based region merging algorithm that constructs a minimum spanning tree over the input graph and incrementally merges regions based on internal difference criteria. Although originally developed for image segmentation, its formulation on weighted graphs makes it directly applicable to mesh-based data.

$$\begin{aligned} m &\leftarrow \max(\text{UDF}) \\ \text{UDF}_i^t &= (10 \cdot (m - \text{UDF}_i))^{10} \end{aligned} \tag{4.6}$$

We use the same concept for edge weighting as for Hierarchical clustering where the weight is the sum of the UDF values of the vertices belonging to the edge, as defined in Equation 4.6. Before segmentation, we apply a transformation to the values which transforms values close to the fracture boundary to very large numbers and attempts to minimize values for interior vertices. This inverts the original distance field and ensures that internal differences used by the method become very small inside fracture pieces.

Felzenszwalb segmentation then processes the weighted graph by sorting the edges in decreasing order and initializes each vertex as its own component. Then components are iteratively merged when the edge weight is sufficiently small relative to the internal variation of the connected components. The full procedure is summarized in Algorithm 5.

4.7 Metrics

We evaluate our approach at two distinct stages of the pipeline. First, we assess the accuracy of the predicted distance fields against their corresponding ground truth. This stage isolates

Algorithm 5 Graph-Based Segmentation (Felzenszwalb–Huttenlocher)

```
1: Sort all edges $E$ into $\pi = (o_1, \dots, o_m)$ by non-decreasing weight
2: Initialize a segmentation $S_0$ where each vertex is in its own component
3: for $q = 1$ to $m$ do
4:     Let $o_q = (v_i, v_j)$ be the $q$-th edge in $\pi$
5:     Let $C_i$ be the component in $S_{q-1}$ containing $v_i$
6:     Let $C_j$ be the component in $S_{q-1}$ containing $v_j$
7:     if $C_i \neq C_j$ and $w(o_q) \leq \mathrm{MInt}(C_i, C_j)$ then
8:         Merge $C_i$ and $C_j$ to form segmentation $S_q$
9:     else
10:        $S_q \leftarrow S_{q-1}$
11:    end if
12: end for
13: return $S_m$
```

the quality of the regression component and indicates how accurately the model predicts the fractures.

Second, we evaluate the performance of our segmentation methods applied to the predicted distance fields. This stage reflects the final performance of our method.

4.7.1 Evaluation Metrics for Distance Field Prediction

To evaluate the performance of the predicted distance field we use two metrics. Firstly, we use the **Mean Squared Error (MSE)** to indicate how accurate the model is overall at predicting the distance field over the mesh. Secondly, we use **Adjusted Mean Absolute Error (AMAE)**, defined as the loss function in Equation 4.5, to better reflect the overall prediction performance, with greater emphasis on the areas close to the fracture boundary.

4.7.2 Evaluation metrics for segmentation accuracy

Segmentation performance is evaluated using metrics that operate on labeled point clouds. Their goals are to assess the segmentation performance in different ways to give a more comprehensive overview.

Chamfer Distance

In this work, we employ Chamfer distance [6] to quantify the similarity between two finite point sets. Let A and B denote two such sets in a metric space. The Chamfer distance is defined as the sum of two directed terms: the average distance from each point in A to its nearest neighbour in B, and the average distance from each point in B to its nearest neighbour in A, as stated in Equation 4.7 and Equation 4.8 [15].

In our application, A corresponds to the set of points in our predicted labelling and B to a set of points in the ground-truth labelling. We compute the Chamfer distance in two distinct contexts: Firstly, we calculate boundary correspondence in our Boundary Chamfer value metric. Secondly, we calculate region correspondence in our Region Chamfer value metric.

$$\text{chamfer}(A,B) = \frac{1}{2n}\sum_{i=1}^{n}\left|A_i - \text{NN}(A_i,B)\right| + \frac{1}{2m}\sum_{j=1}^{m}\left|B_j - \text{NN}(B_j,A)\right| \tag{4.7}$$

$$\text{NN}(x,C) = \arg\min_{y\in C}\left\|x-y\right\| \tag{4.8}$$

Boundary Chamfer score

To define the **Boundary Chamfer (BC)** score, we first extract boundary points for both predicted and ground-truth segmentations using the edge-detection procedure detailed in chapter 4. By restricting A and B to those points that lie on the fracture boundaries, we obtain two boundary point sets whose Chamfer distance directly quantifies the similarity between the predicted and ground-truth fracture lines.

Region Chamfer score

The **Region Chamfer (RC)** score measures the geometric correspondence between predicted and ground-truth fracture pieces. For each ground-truth fragment, the most similar predicted fragment is identified using the Chamfer distance. It is defined using the following directed covering procedure [3]: For each pair of ground-truth and predicted fracture regions, we compute their Chamfer distance. Because the total number of fragments never exceeds ten, a brute-force evaluation over all possible pairings is computationally trivial. Formally, let $G = \{g_i\}_{i=1}^{N}$ denote the set of ground-truth fragments and $P = \{p_j\}_{j=1}^{M}$ the set of predicted fragments, with $N,M < 10$. We first compute the Chamfer distance, $\text{chamfer}(g_i, p_j)$ for every $(i, j) \in \{1,...,N\}\{1,...,M\}$. Then for each ground-truth fragment g_i, we select the predicted fragment p_{j*} that minimizes:

$$j^*(i) = \arg\min_{1\leq j\leq M} \text{chamfer}(g_i, p_j), \tag{4.9}$$

The Region Chamfer score is then defined as the sum of the minimal distances over all ground-truth fragments:

$$D_{\text{Region}} = \sum_{i=1}^{N} \text{chamfer}\big(g_i,\, p_{j^*(i)}\big). \tag{4.10}$$

A lower value of D_{Region} indicates a closer geometric correspondence between the predicted and ground-truth fracture pieces.

Adjusted Rand Index

The **Adjusted Rand Index (ARI)** [25] evaluates the similarity between two clustering outcomes. The ARI evaluates the agreement between two partitions by considering all pairs of samples and determining whether each pair is consistently assigned to the same or different clusters across both partitions. Unlike the standard Rand Index, ARI corrects for chance, making it especially suitable for comparisons across clusterings with varying numbers of clusters.

For simplicity and clarity, in Equation 4.11 we present the formula of the standard Rand Index [57], as the Adjusted Rand Index involves more complex computations designed primarily to correct for chance agreement. However, the core idea of evaluating the similarity between clustering outcomes is identical for both instances.

$$\mathrm{RI} = \frac{a+b}{\binom{n}{2}} \tag{4.11}$$

For the normal Rand Index, a indicates the number of pairs of elements that are in the same cluster in both partitions, b indicates the number of pairs that are in different clusters in both partitions, and n is the total number of elements.

The ARI produces values in the range from -1 to 1, where a score close to 1 denotes similarity between clusterings, a score of 0 suggests random agreement, and negative values indicate poorer agreement than expected by random chance. Therefore, ARI values close to 1 indicate that the predicted fracturing is close to the ground-truth fracturing. While ARI values close to, or below, 0 indicate a bad match between predicted and ground-truth fracture.

Adjusted Mutual Information

Additionally, we utilize the **Adjusted Mutual Information (AMI)** [73] score to measure the similarity between two clustering results. AMI evaluates the mutual independence between two clusterings by quantifying the shared information, adjusted for chance, which is intimately linked to entropy. It quantifies the amount of information obtained about one random variable by observing the other random variable.

Similar as before, in Equation 4.12 we present the formula for Mutual Information [65, 68], instead of Adjusted Mutual Information, as the formula for Adjusted Mutual Information involves more complex computations designed to correct for chance agreement, while the core idea remains unchanged.

$$MI(U,V) = \sum_{i=1}^{|U|} \sum_{j=1}^{|V|} P(i,j) \log \frac{P(i,j)}{P(i)P(j)} \tag{4.12}$$

For the Mutual Information, U and V represent the predicted and ground-truth clusterings, $P(i,j)$ is the joint probability distribution that a randomly selected object belongs to cluster i in partition U and cluster j in partition V. $P(i)$ and $P(j)$ are the marginal probabilities of an object belonging to cluster i in partition U and cluster j in partition V, respectively. The probabilities represent the percentage of occurrence for each label in a single fracture. For example, $P(i)$ represents the number of occurrences of label i divided by the total number of vertices in the predicted clustering U.

The AMI score ranges from 0, no agreement beyond chance, to 1, perfect agreement. Therefore, when comparing predicted to ground-truth segmentations, AMI values close to

1 indicate a predicted fracturing close to the ground-truth, while values close to 0 indicate that the predicted fracture is not much better than a random fracturing when compared to the ground-truth.

Segmentation Acceptance criterion

To determine whether a segmentation is suitable for real-time use in a video game, we introduce a binary acceptance criterion based on the Adjusted Rand Index. Segmentations with ARI ≥ 0.7 are considered acceptable. We motivate our threshold in chapter 5.

4.8 Implementation Details and Hyperparameters

Compute and software

All experiments were run on a NVIDIA GeForce RTX 4070 Ti SUPER (16 GB VRAM). Our models were implemented using PyTorch. Graph-based segmentation methods were implemented using NetworkX. Geodesic distances were computed using Potpourri3D.

Data representation

Our method does not perform normalization of vertex coordinates. Instead we manually normalized our meshes using blender before importing by uniformly scaling such that the largest x, y, or z axis equals 1. Breaking good also uses normalization of meshes when generating the fracture patterns. However, normalized meshes need to be provided, since we map the fractures back to the original mesh.

The impact direction is represented as a unit vector, while impact position is defined as the vertex location on the mesh. UDF values are measured based on the provided mesh. Consequently, hyperparameters such as the AMAE weighting α and segmentation thresholds are scale-invariant to the input meshes.

Loss

We train our models using the adapted mean absolute error (AMAE) loss (Equation 4.5) with $\alpha = 10$, which emphasizes accuracy near the fracture boundaries.

Randomness

During training we set `torch.manual_seed(1)`. Randomness in the CUDA kernel was not controlled. Runtime measurements were obtained using `torch.cuda.synchronize()` before and after code blocks to ensure accurate GPU timing.

MLP

For the MLP, each vertex is represented by a 9D input feature vector consisting of vertex coordinates (x, y, z) concatenated with impact position and impact direction (3+3+3). Training is done using the Adam optimizer (lr = 1e-4) for 200 epochs using AMAE. The batch size is defined as the number of vertices per optimization step.

DeltaConv

DeltaConv is trained on samples consisting of entire meshes, consisting of vertices with the same 9D features as above. Each batch consists of 5 meshes/impulses. Training was done use Adam (lr = 1e-4) for 100 epochs. Since the mesh is fixed, kNN neighborhood queries and associated matrices are precomputed and reused during inference.

4.8.1 Segmentation hyperparameters

Local maxima are defined as vertices whose UDF value is strictly larger than all directly connected neighbours. For watershed, two adjacent regions are merged when at least one of the two vertices incident to the connecting edge has a UDF of 0.2.

For region growing, regions are merged if both connected vertices connected by an edge have UDF values of at least 0.1. Region expansion is controlled by a consideration threshold that is decremented by 0.004 until it reaches 0, when it considers all vertices immediately.

Hierarchical segmentation merges regions based on the sum of the UDF values of the vertices incident to each edge. Edged are processed in decreasing sum order and this process is stopped below sums of 0.2.

K-Means targets 3 clusters and prior to clustering all UDF values above 0.5 are clipped to 0.5 to limit the influence of outliers.

Felzenszwalb segmentation is implemented using a library made by Rakshit. Although originally designed for images, the method is based on graphs and was therefore adapted to work on mesh-based graphs. We use their method with a scale parameter k of 4 and a minimum size of 20 vertices. We use a data transform for better performance, discussed previously in Equation 4.6.

These values were selected through manual experimentation on a small validation subset and are expressed in normalized mesh space.

Chapter 5

Experiments

This chapter evaluates the proposed surface-based impulse-dependent brittle fracture prediction framework on varying meshes and mesh-sizes using both an MLP and a graph-convolutional DeltaConv architecture. We additionally evaluate several segmentation algorithms on varying meshes and mesh sizes.

5.1 Setup

5.1.1 Baseline

Our work is inspired by DeepFracture of Huang and Kanai. Since neither source code nor training data for DeepFracture is publicly available, a direct quantitative comparison is not feasible. Instead, we adopt its core conceptual ideas and study how different learning architectures in our surface-based framework perform. In particular, we evaluate the performance of a per-vertex MLP and a graph-based architecture based on DeltaConv to analyse the effect of explicitly modelling mesh connectivity.

5.1.2 Data

We evaluate our method on three watertight triangle meshes of varying geometric complexity: the Stanford bunny [72], a simple chair model made in Blender, and a vase chosen from the Breaking Bad dataset [62]. These meshes were chosen to represent a range of shapes: the Stanford Bunny represents a smooth shape with localized weak points such as the ears; the chair contains multiple protrusions such as the legs and armrest; and the vase is a thin walled object, allowing us to evaluate the robustness of the method.

For each mesh, we generate a dataset of 1000 fracture instances using Breaking Good [63]. Each instance consists of randomly sampled pairs of impulses and per-vertex geodesic distance fields representing the fracture boundary.

5.1.3 Evaluation Procedure

For each mesh, the dataset is first split into a fixed training set (85%) and a held-out test set (15%). The test set is never used for hyperparameter tuning.

To avoid data leakage, we perform 5-fold cross-validation with fixed splits: in each fold, a distinct 1/5th of the data serves as the validation set, and the remaining 4/5ths as the training set. These partitions remain constant across all models. When passing Optuna the performance of the cross-validation, we make sure to pass it the mean performance of the folds.

Hyperparameter optimization is performed on the training set using 5-fold cross-validation. In each fold, four fifths of the training data are used for training and the remaining fifth for validation. These folds assignments are kept consistent across all models. The validation of each fold is repeated 5 times to ensure consistent accuracy and run-time measurements. We use Optuna [2] to optimize the hyperparameters of both the MLP and DeltaConv models, using the mean validation performance as the optimization objective while optimizing for both run-time and AMAE loss. NSGAII, a multi-objective genetic algorithm [13], was used to suggest hyperparameter configurations.

After retrieving the best performing hyperparameters, we retrain the model on the full training set and evaluate on the held-out test set to measure final performance. This procedure ensures that no information from the test set is used during model selection.

5.1.4 Acceptance criterion

Figure 5.1 contains both correct and incorrectly placed fractures used for calibrating the threshold. In column 1 of the grid, segmentations produced by watershed, region growing and hierarchical segmentation closely match the ground-truth and all achieve ARI values above 0.75. Column 5 shows some incorrectly placed fractures for watershed and region growing with ARI values close to 0.6. Based on these observations we select the acceptance threshold as ARI ≥ 0.7.

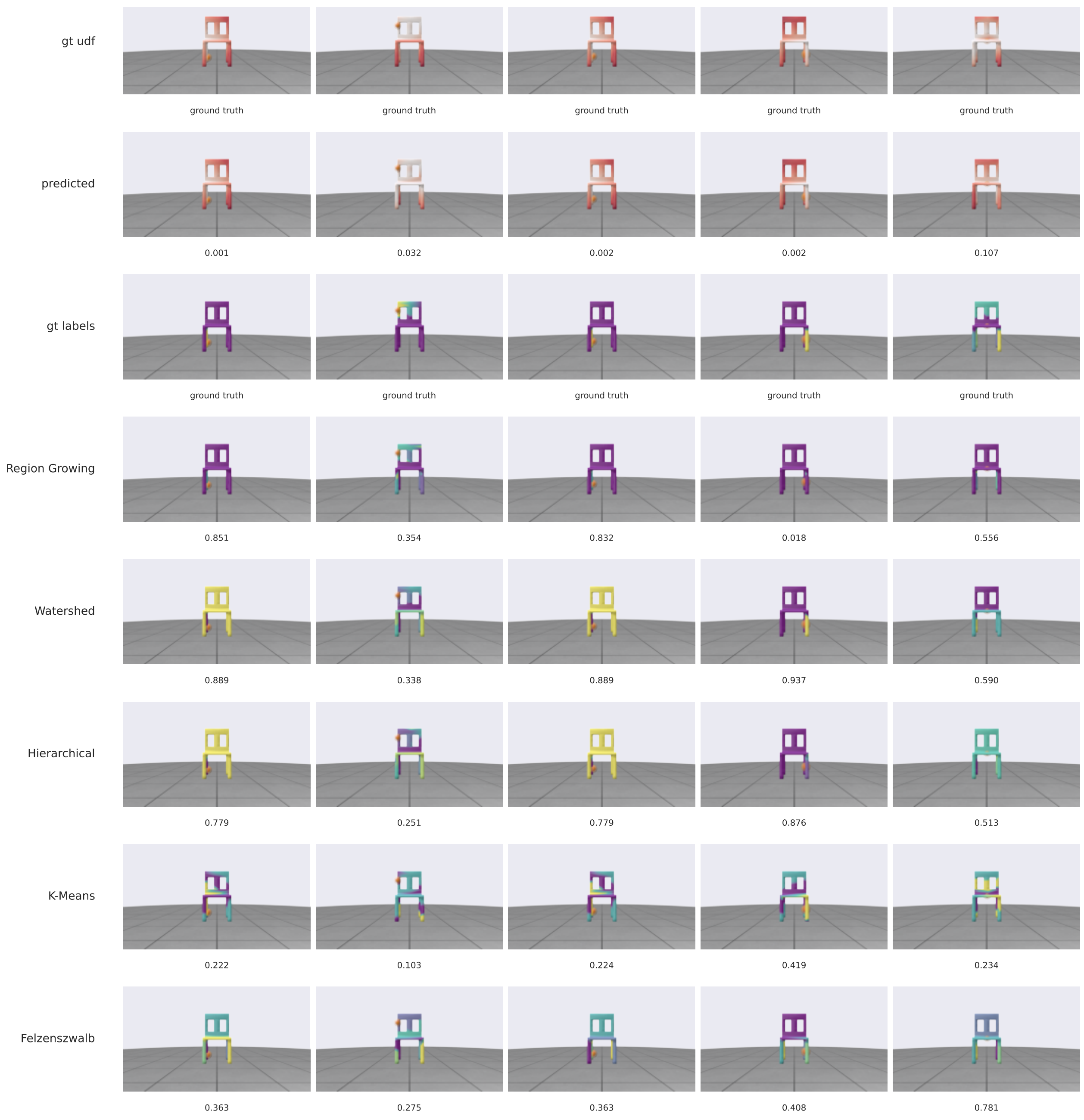


Figure 5.1: Segmentation results with MLP. Rows (top to bottom): ground-truth unsigned distance field (UDF), predicted UDF, ground-truth segmentation, and results from region growing, watershed, hierarchical, k-means, and Felzenszwalb. Columns: dataset samples. In UDFs, white = near fracture, red = far. Segmentation colors are arbitrary and only distinguish regions. Numbers indicate Rand index for segmentations (higher is better) and AMAE for UDFs (lower is better). Red dots mark impact points.

5.2 Architecture Comparison

This section presents a comparison between the MLP and DeltaConv architectures, which we evaluate based on their ability to predict distance fields and applicability to downstream segmentation methods.

Model selection is based on AMAE regression accuracy and inference efficiency, evaluated using the methodology described in the evaluation procedure section. Although we optimize hyperparameters for distance field regression, we also report segmentation metrics to test the impact of regression quality on the segmentation stage. We illustrate this correlation in Figure 5.5.

We first evaluate the performance of the MLP architecture, then test the DeltaConv model, and finally provide a direct comparison between their best-performing configurations. In this section we always use watershed segmentation to test segmentation performance. In the next section, we evaluate different segmentation approaches.

5.2.1 MLP hyperparameter tuning

This subsection analyzes how the depth and width of the MLP influence regression accuracy and inference speed. All evaluated models use constant-width hidden layers with layer normalization and ReLU activation functions. The number of hidden layers varies between 1 and 20, while the width of the hidden layers is varied between 32 and 1024 neurons. A total of 1000 different configurations were tested.

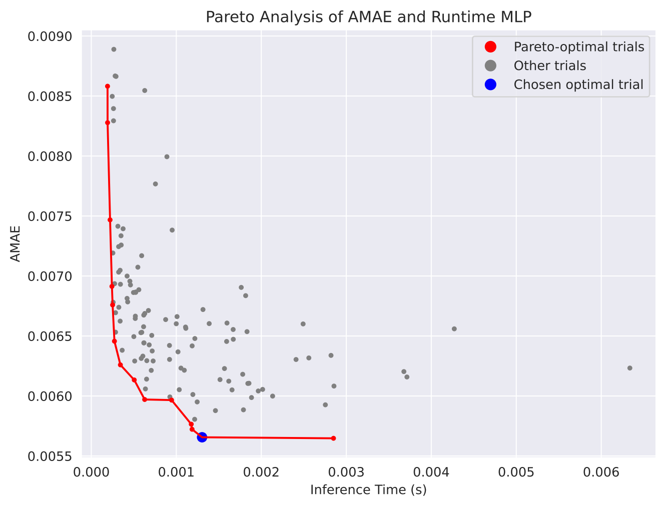


Figure 5.2: Plot showing the pareto front for the MLP plotting inference time (s) against validation AMAE

Figure 5.2 shows the Pareto front of the MLP model between AMAE and inference time. The lowest observed AMAE values are as low as 0.0056, while faster configurations achieve inference times under 1 ms, with increased error. We see regression improvements beyond AMAE of approximately 0.0057 are accompanied by a sharp increase in inference time, indicating diminishing returns on the Pareto curve past this point.

Based on this trade-off, we identify the optimal range for our MLP models as the models on the Pareto front with AMAE values between 0.0056 and 0.0060. Within this interval, inference time has not yet begun to increase significantly, while achieving near-optimal regression accuracy.

$$\arg\min_{t\in\text{trials}}\left(\sqrt{\text{runtime}_t^2 + 3\cdot\text{AMAE}_t^2}\right) \tag{5.1}$$

We select a single representative configuration from the Pareto front by minimizing the weighted objective function shown in Equation 5.1. We assign higher weight to AMAE than to inference time. As we will show later, the overall runtime is dominated by the segmentation step and thus the runtime during the inference stage has limited effect on the overall performance. The selected configuration is indicated by a blue dot in Figure 5.2.

Hidden Size	**Layers**
704	7

Table 5.1: Model configuration for the chosen Pareto optimal network.

The selected hyperparameters consists of a 704 neurons per layer, with 7 Hidden layers, as can be seen in Table 5.1. These hyperparameters were relatively high, with respect to the range of possible hyperparameter options.

AMAE	**Inference Time (ms)**	**BC**	**RC**	**MSE**	**ARI**	**AMI**
0.00588	1.461	0.0588	0.0228	0.00796	0.900	0.851

Table 5.2: Performance metrics for the best-performing configuration.

Table 5.2 summarizes the performance metrics of the selected Pareto-optimal MLP configuration. The model achieved low AMAE values of 0.00588 with an average inference time of 1.46 ms. High segmentation quality was obtained, as indicated by ARI and AMI scores of 0.900 and 0.851, respectively, along with low boundary and region consistency errors (BC = 0.0588, RC = 0.0228).

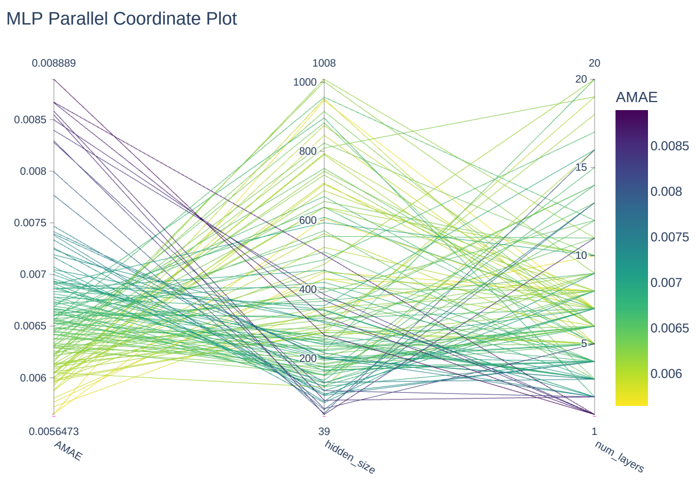


Figure 5.3: A parallel coordinate plot showing the impact of the hyperparameters on the AMAE distance field regression score

Figure 5.3 shows the parallel coordinate plot of the relationship between MLP hyperparameters and segmentation performance. The plot indicates the correlation between models with small hidden layer sizes and high losses, as well as correlations between a smaller layers sizes and higher losses. We see a slightly clearer ordering of AMAE performance for hidden size, than for the number of layers indicating it to be a slightly more important parameter.

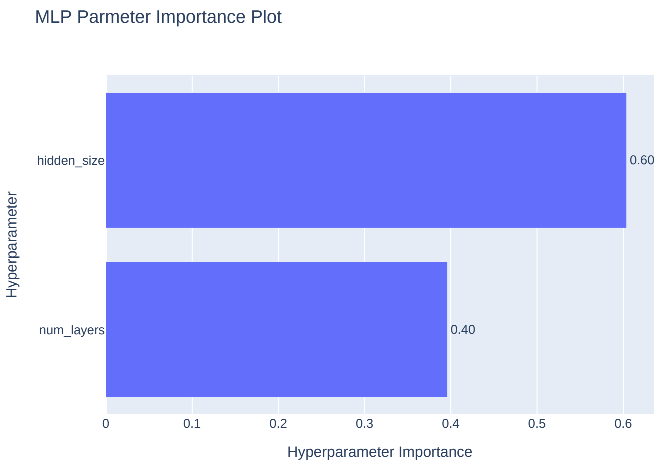


Figure 5.4: Hyperparameter Importance for the MLP AMAE calculated using fANOVA hyperparameter importance evaluation algorithm

Figure 5.4 reports the relative importance of MLP hyperparameters with respect to

AMAE, computed using the fANOVA hyperparameter importance evaluation algorithm [26]. The results indicate that hidden layer sizes has a stronger influence on the regression accuracy than the number of layers, confirming the trends we saw in the parallel coordinate plot.

Dataset	**AMAE**	**Inference Time (ms)**	**BC**	**RC**	**MSE**	**ARI**	**AMI**
Validation	0.00588	1.461	0.0588	0.0228	0.00796	0.900	0.851
Test	0.00676	1.460	0.0854	0.0490	0.0244	0.838	0.805

Table 5.3: MLP Chosen configuration performance

A comparison between the selected MLP configuration on the validation and test sets is shown in Table 5.3. As expected, the model performs slightly worse on the held-out test set compared to the validation set. This indicates limited overfitting and reasonable generalisation to unseen fractures.

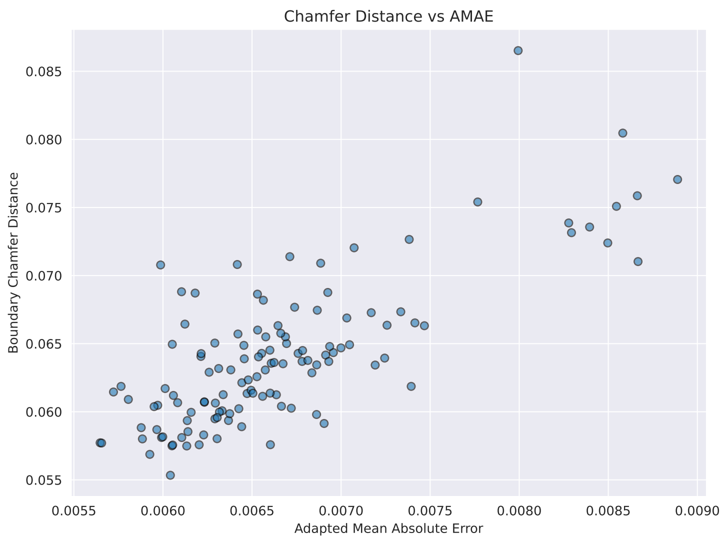


Figure 5.5: Plot showing the correlation between Adapted Mean Absolute Error and Boundary Chamfer distance

Figure 5.5 visualizes the relationship between regression error (AMAE) and segmentation performance (Boundary Chamfer distance). A Pearson correlation analysis, as shown in Table 5.4, shows a correlation coefficient of $r = 0.767$ with a p-value of 7.05×10^{-25}. This statistically significant positive correlation indicates that improvements in regression error lead to better fracture segmentation performance.

5.2.2 DeltaConv hyperparameter tuning

This subsection analyzes the impact of its hyperparameters on DeltaConv. The hyperparameters are: the number of convolutional layers, size of convolutional layers, number of

Metric	**Value**
Correlation coefficient (r)	0.767
P-value	7.05×10^{-25}

Table 5.4: Pearson Correlation Results

MLP layers, embedding size, number of neighbours used, and gradient kernel width. We ran roughly 150 trails, while again optimizing for both regression accuracy and runtime.

The hyperparameters and their ranges are summarized in Table 5.5. The number of convolutional layers is limited to six to prevent excessive GPU memory usage. Similarly, we limit the number of features per convolutional layer to the range of 64-1024 and the embedding size to 64-2048 to ensure training remains feasible within our hardware budget. The number of neighbours is varied between 2 and 30 and controls how many k-nearest neighbours are considered, while the gradient kernel width varies between 1 and 3 and determines the size of the kernel relative to the average edge length in each shape.

Hyperparameter	**Lower Bound**	**Upper Bound**
Number of Convolutional Layers	1	6
Number of features in Convolutional Layers	64	1024
Number of MLP Layers	1	3
Embedding Size	64	2048
Number of Neighbours	2	30
Gradient Kernel Width	1	3

Table 5.5: DeltaConv Hyperparameter Search Ranges

Training each DeltaConv model requires substantially more time compared to MLP models. Therefore, the search space for certain hyperparameters was deliberately restricted to practically relevant ranges. Additionally, hyperparameters such as the number of convolutional layers were limited to prevent excessive VRAM consumption, which could lead to CUDA memory issues in our setup.

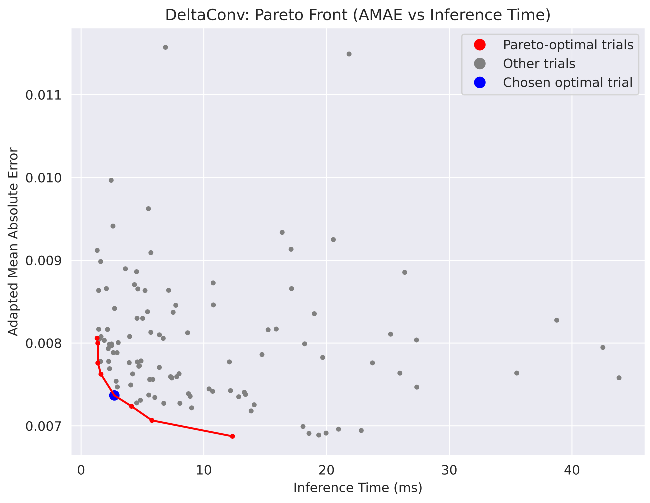


Figure 5.6: Plot showing the Pareto front for DeltaConv plotting inference time (ms) against Validation AMAE during Optuna Optimization

Figure 5.6 shows the Pareto front of the DeltaConv model, illustrating a trade-off between AMAE and inference time. Each dot represents an Optuna run with varying parameters. The curve indicates that trials with AMAE scores below approximately 0.007 start to have inference times exceeding 10 ms, while trials with runtimes close to 1 ms start to have errors exceeding 0.008. We selected the representative trial from the Pareto front using the same weighted objective function as for the MLP, shown in Equation 5.1.

Number of Convolutional Layers	Conv Layer Size	Embedding Size	Number of MLP Layers	Number of Neighbours	Grad Kernel Width
2	292	343	1	5	2

Table 5.6: Hyperparameters for the chosen Pareto Optimal DeltaConv model: number of convolutional layers, convolutional layer size, embedding size, number of MLP layers, neighbourhood size, and gradient kernel width.

The hyperparameters of the selected DeltaConv configuration are presented in Table 5.6. The model employs two convolutional layers with 292 features each, an embedding dimension of 343, and a single MLP layer for these embeddings. Furthermore, five neighbours were used, and the kernel width was set to a value of 2. Overall, these hyperparameter choices are on the lower end of the ranges explored during testing, reflecting the trade-of between regression accuracy and computational cost for our real-time system.

Table 5.7 reports the performance of the selected DeltaConv configuration on the validation and test sets. The results indicate consistent behaviour across both splits, with an expected modest decrease in regression quality on the held-out test set. Interestingly, the segmentation performance is actually higher for the test set.

Dataset	**AMAE**	**Inference Time (ms)**	**BC**	**RC**	**MSE**	**ARI**	**AMI**
Validation	0.00737	2.711	0.189	0.128	0.0222	0.720	0.726
Test	0.00876	2.762	0.172	0.119	0.0348	0.738	0.734

Table 5.7: DeltaConv Chosen configuration performance

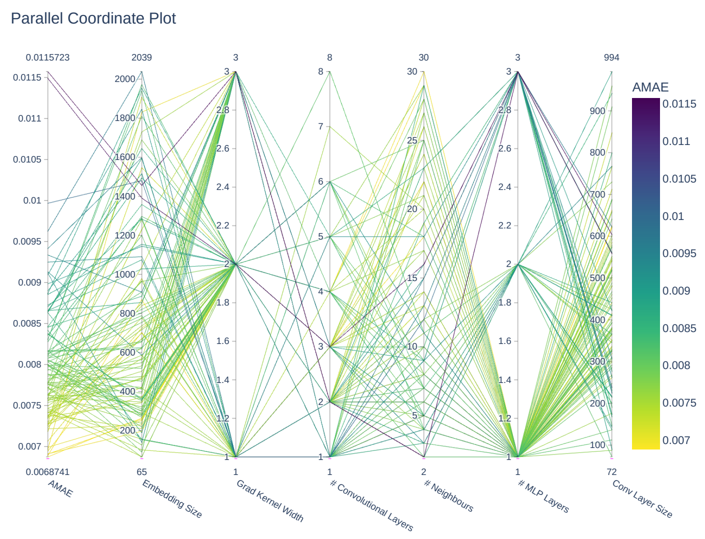


Figure 5.7: A parallel coordinate plot showing the impact of the hyperparameters on the loss for DeltaConv

Figure 5.7 presents a parallel coordinate plot for the relationship between DeltaConv hyperparameters and validation AMAE. although multiple parameters vary simultaneously, certain trends are evident. We observe that configurations with a low number of neighbours are associated with higher AMAE values. In addition, we observe that DeltaConv models with 3 MLP layers include many runs with poor AMAE values.

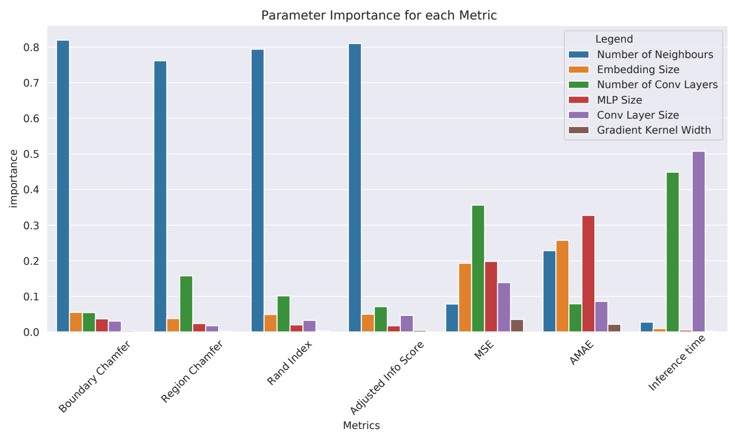


Figure 5.8: Plot showing parameter importance for DeltaConv for each metric

Figure 5.8 reports the fANOVA-estimated [26] importance of the DeltaConv hyperparameters with respect to each evaluated performance metric. Convolutional layer size and the number of convolutional layers are the dominant factors influencing inference time. A clear distinction is observed for parameter importance between metrics that measure regression accuracy (AMAE and MSE) and those that measure segmentation performance (BC, RC, ARI and AMI).

For the regression metrics, all parameters seem to contribute comparable importance, indicating that either multiple parameters jointly influenced the distance field prediction or that the available data was insufficient for fANOVA to isolate dominant factors. In contrast, the number of neighbours is consistently the most influential parameter for segmentation quality across all segmentation metrics.

5.2.3 Comparing MLP and DeltaConv

This subsection compares the performance of the MLP and DeltaConv with respect to regression and segmentation accuracy, as well as computational efficiency. The comparison is performed by analyzing the respective Pareto fronts and by evaluating the performance of the selected configurations identified in the previous subsections.

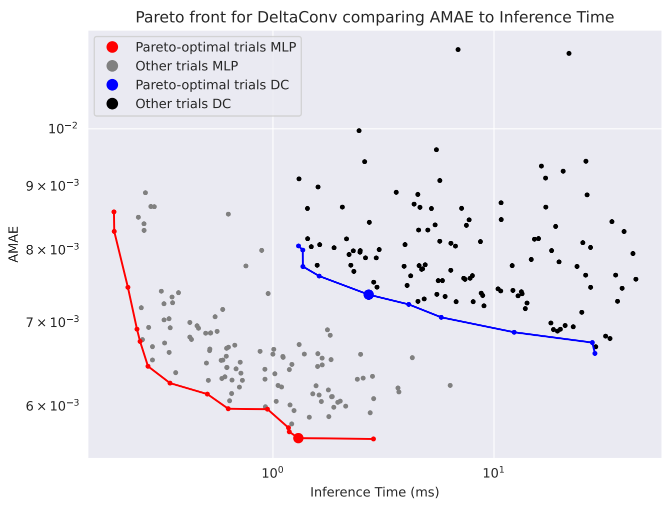


Figure 5.9: Plot showing the Pareto front for both the MLP and DeltaConv plotting inference time (s) against validation loss using logarithmic axis.

Figure 5.9 presents an overlapped view of the Pareto front of both the MLP and DeltaConv. The MLP consistently achieves lower validation AMAE at comparable or lower inference times. While we anticipated the MLP to have a runtime advantage due to its simpler architecture, the observed regression performance of DeltaConv remains consistently lower across the explored hyperparameter ranges for the fixed-mesh setting considered here.

Model	AMAE ↓	Inference Time (ms) ↓	BC ↓	RC ↓	MSE ↓	ARI ↑	AMI ↑
MLP	**0.00676**	**1.460**	**0.0854**	**0.0490**	**0.0244**	**0.838**	**0.805**
DeltaConv	0.00876	2.762	0.172	0.119	0.0348	0.738	0.734

Table 5.8: Pareto Chosen configuration Evaluation Metrics

Table 5.8 compares the selected Pareto-optimal configurations of the MLP and DeltaConv architectures on the held-out test set. The MLP performs better at regressing the distance field and has lower segmentation error across all reported metrics while requiring approximately half the inference time.

Figure 5.10 visualizes the predicted distance fields for both MLP and DeltaConv, while also showing the ground-truth. The displayed examples show that both methods capture the overall fracture boundary structure, with both methods producing the same incorrect fracture in the third column. However, as shown below each image, the MLP produces lower average per-vertex AMAE values in the majority of the cases.

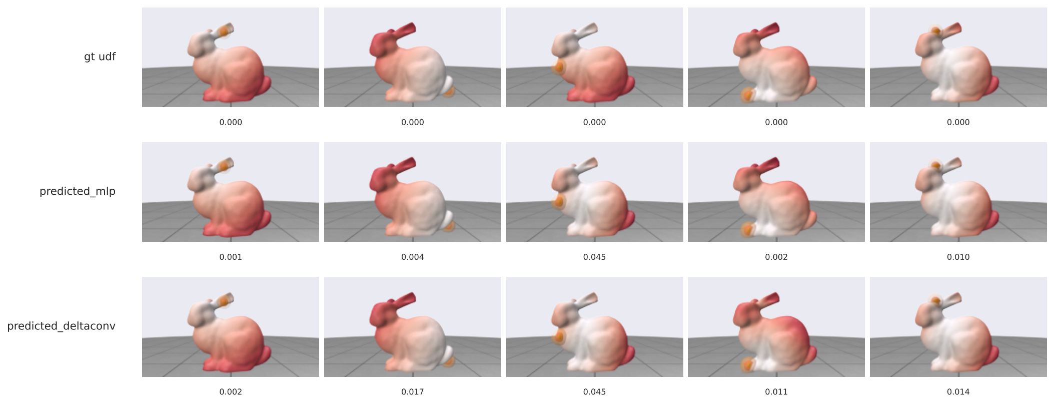


Figure 5.10: UDF prediction comparison between DeltaConv and MLP. Values below each example show average per-vertex AMAE. White indicates vertices near the fracture boundary (low UDF), red indicates vertices farther away (high UDF). Columns use constant impulse information.

5.2.4 Investigating the slow inference speed of DeltaConv

This subsection analyzes the factors that contribute to the higher inference time of the DeltaConv architecture. In addition to increased memory consumption, which limits the feasible model size, DeltaConv exhibits significantly longer runtimes compared to the MLP. To identify computational bottlenecks, we synchronized CUDA execution between individual submodules of the DeltaConv forward pass and recorded their execution times during validation.

Table 5.9: Runtime breakdown (milliseconds) of a DeltaConv model with 3 layers and 500 layer size. Runtime breakdown is per subcomponent

Component	**Default (ms)**	**Optimized (ms)**
DeltaNet — internal breakdown		
k-NN Graph + Matrix Build	4.24	0.10
Value Retrieval	0.15	0.11
DC Layer, Iteration 1	1.90	1.90
DC Layer, Iteration 2	4.60	4.60
DC Layer, Iteration 3	1.70	1.70
Subtotal (DeltaNet)	*12.59*	*8.41*
Concatenation 1	0.07	0.06
Global Embedding MLP	0.22	0.22
Pooling	0.34	0.44
Concatenation 2	0.08	0.08
Output MLP	0.35	0.35
Total (top-level)	**13.65**	**9.56**

Table 5.9 reports a component-wise runtime breakdown for a DeltaConv configuration with three convolutional layers of width 500, comparing the default implementation with an optimized version in which the k-nearest neighbour graph and operator matrices are precomputed.

Pre-computing these components improved performance by approximately 4 ms, primarily by eliminating the repeated k-NN graph construction and matrix creation. The DeltaConv layers themselves still remain the largest contributors to runtime. In the optimized configuration the three convolutional layers account for 88% of the total inference time.

Interestingly, the second iteration of the DC layer takes almost 3 times as much time as the two outer layers. This suggests that these inner layers are more costly due to propagations of vectors between layers. Since the majority of the runtime is spent on the convolutional layers themselves, opportunities for further optimization using pre-processing is limited.

5.3 Comparing segmentation methods

In this section, we evaluate the performance of the segmentation methods when applied to predicted unsigned distance fields. The methods are compared with respect to segmentation accuracy and computational efficiency. Additionally, we compare how the segmentation methods differ predicting unsigned distance fields for both MLP and DeltaConv, whose performance can be seen in Table 5.10.

We compared 5 segmentation methods, Watershed segmentation, Region growing, K-Means, Hierarchical segmentation and Felzenszwalb segmentation. These methods takes as input the ground-truth mesh and the predicted UDF.

Model	AMAE	MSE	Seg. Method	Duration (ms)	RC	BC	ARI	AMI
DeltaConv	0.0087	0.034	Felzenszwalb	80.20	0.244	0.177	0.516	0.555
			Hierarchical	359.4	0.411	0.104	0.749	0.623
			K-Means	68.27	0.262	1.201	0.014	0.094
			Region Growing	363.5	0.056	0.109	0.835	0.814
			Watershed	26.80	0.111	0.157	0.754	0.743
MLP	0.0071	0.024	Felzenszwalb	672.8	0.236	0.164	0.533	0.569
			Hierarchical	357.1	0.395	0.075	0.802	0.670
			K-Means	66.86	0.265	1.191	0.017	0.097
			Region Growing	328.7	**0.034**	**0.057**	**0.892**	**0.866**
			Watershed	**23.66**	0.043	0.080	0.855	0.816

Table 5.10: Performance comparison of models across segmentation methods with durations in milliseconds (three significant digits), with best-performing values highlighted.

Table 5.10 reports the segmentation accuracy and runtime for all segmentation methods applied to distance fields produced by both learning models. K-Means performs consistently poorly for both models across all segmentation metrics for both models. Region

Growing achieves the highest segmentation accuracy, followed by Watershed and Hierarchical segmentation.

Hierarchical segmentation performs well for all segmentation metrics, apart from Region Chamfer where it performs even worse than K-Means. Out of the top performing methods, Watershed has a substantially faster runtime compared to Region Growing and Hierarchical. Felzenszwalb is consistently worse than the top 3 methods, while performing better K-Means.

All methods, apart from K-Means, consistently perform better on the MLP than on DeltaConv for all segmentation metrics. The segmentation methods have very similar runtime between MLP and DeltaConv output, apart from Felzenszwalb which is significantly slower on MLP data.

Segmentation Consistency

Table 5.11 reports how consistently our methods meet our segmentation threshold. This data agrees with what we saw before: Hierarchical, Region Growing and Watershed all perform well and the differences in consistency are marginal. Region Growing has both the highest average and consistency in meeting the threshold. Hierarchical has the third highest average, but meets the threshold with a slightly higher consistency than Watershed, which on average performs better.

Method	**Accuracy (ARI)**	**Threshold percentage**
Felzenszwalb	0.533	7 %
Hierarchical	0.802	86 %
K-Means	0.014	0 %
Region Growing	0.892	88 %
Watershed	0.855	85 %

Table 5.11: Comparison between average ARI values and percentage of ARI values meeting our threshold of 0.7 on the bunny mesh test set for each segmentation method

Results for the MLP model

Figure 5.11 shows segmentation outputs for all segmentation methods based on MLP predicted distance fields. We observe that, while looking at columns 1, 2, and 4, Region Growing, Watershed and Hierarchical Segmentation perform best. These methods create segmentations that closely resemble the ground-truth. In column 5, we observe that these methods all incorrectly segment the ears, which are not present in the ground-truth segmentation. However, the MLP incorrectly predicts an ear fracture. Similarly, in column 3, the MLP incorrectly predicts a fracture through the body itself, which yields incorrect segmentations compared to the ground-truth.

K-Means produces incorrect segmentations across all examples. Felzenszwalb produces somewhat plausible segmentations, but always segments the ears, even in cases where the distance field does not predict fractures in these areas, resulting in unstable fracture predictions.

Results for the DeltaConv model

Figure 5.12 shows the same image grid, but instead using DeltaConv model predictions. Region growing, Watershed, and Hierarchical segmentation perform well in columns 1 and 4. Contrary to the MLP, the segmentation models struggle to segment the distance field for column 2, where Region growing and Hierarchical segmentation cannot find a segmentation at all. DeltaConv, like the MLP, also incorrectly predicts a full body fracture in column 3 and an incorrect ear fracture in column 5, which can be seen in the resulting segmentations.

As with the MLP, K-Means does not create any comprehensive fractures, while Felzenszwalb creates comprehensible fractures, but produces a lot of over-segmentation close to the ears.

5.4 Comparing various meshes

Table 5.12 presents the segmentation performance of the evaluated methods for different shapes, including standard deviations for all metrics. Since the MLP previously outperformed DeltaConv, only the MLP results are reported here. Image grids containing non-bunny meshes can be found in Appendix A.

Method	**Bunny**						**Chair**						**Vase**					
AMAE	0.00645						0.01710						0.00956					
	Duration (ms ↓)		BC (↓)		ARI (↑)		Duration (ms ↓)		BC (↓)		ARI (↑)		Duration (ms ↓)		BC (↓)		ARI (↑)	
	Mean	Std	Mean	Std	Mean	Std	Mean	Std	Mean	Std	Mean	Std	Mean	Std	Mean	Std	Mean	Std
Felzenszwalb	667.50	36.3	0.162	0.15	0.531	0.20	19.10	0.25	0.270	0.14	0.240	0.21	9.04	12.6	0.321	0.09	0.002	0.02
Hierarchical	353.10	27.9	0.075	0.11	0.806	0.21	8.60	8.89	**0.090**	0.12	0.670	0.34	2.86	0.21	**0.068**	0.04	**0.432**	0.25
K-Means	64.20	31.5	1.194	0.23	0.018	0.04	5.50	8.43	0.510	0.18	0.120	0.10	4.25	12.5	0.640	0.09	0.003	0.01
Region Growing	325.50	64.5	**0.060**	0.12	**0.893**	0.22	20.30	4.05	0.120	0.11	**0.710**	0.32	11.45	0.83	0.128	0.11	0.224	0.22
Watershed	**24.40**	23.4	0.077	0.11	0.867	0.22	**4.00**	11.3	0.130	0.14	0.620	0.30	**1.58**	0.12	0.137	0.07	0.264	0.17

Table 5.12: Structured comparison of methods across Bunny, Chair, and Vase with segmentation metrics. Best mean values per object are bolded (excluding constant AMAE).

For these experiments, we measured Boundary Chamfer (BC) distance to assess the correctness of the fracture lines and Adjusted Rand Index (ARI) to assess the correctness of the segmentation.

For the bunny mesh, a newly trained model was used, resulting in slightly different but comparable data to earlier experiments. Watershed achieved the lowest runtime on average. Its runtime standard deviation was nearly equal to the mean, indicating that while some runs were faster, a smaller number took considerably longer. Among the methods, watershed still had the lowest runtime standard deviation. For BC, the standard deviations were similar across most methods, with k-means notably higher and Felzenszwalb slightly higher than the others. For ARI, standard deviations were also similar, with k-means showing a lower deviation due to its consistently poor performance.

For the chair mesh, results were similar to the bunny but with some differences. Watershed remained the fastest method, though its runtime was closer to that of k-means and

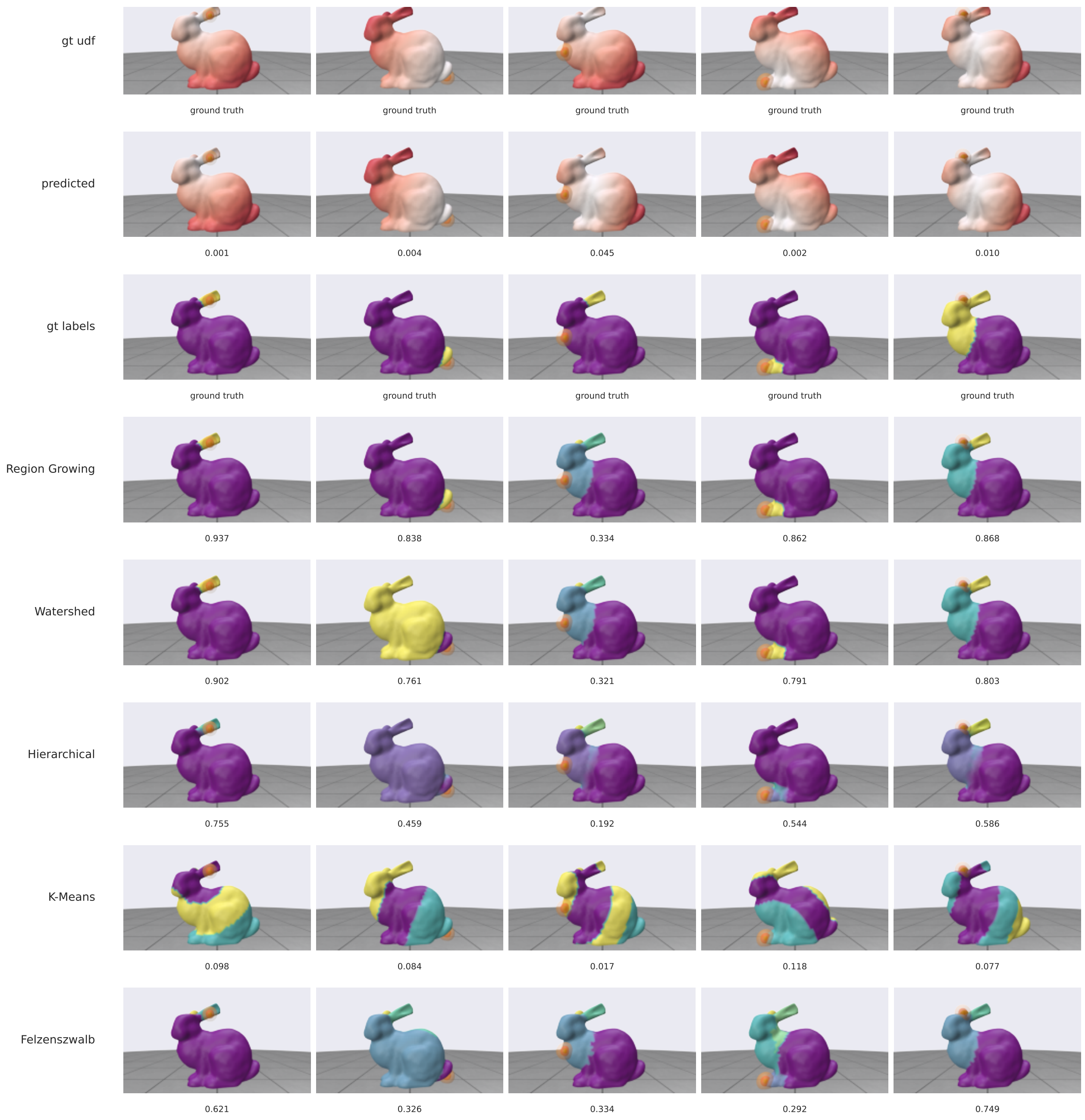


Figure 5.11: Segmentation results with the MLP. Rows (top to bottom): ground-truth unsigned distance field (UDF), predicted UDF, ground-truth segmentation, and results from region growing, watershed, hierarchical, k-means, and Felzenszwalb. Columns: dataset samples. In UDFs, white = near fracture, red = far. Segmentation colors are arbitrary and only distinguish regions. Numbers indicate Rand index for segmentations (higher is better) and AMAE for UDFs (lower is better). Red dots mark impact points.

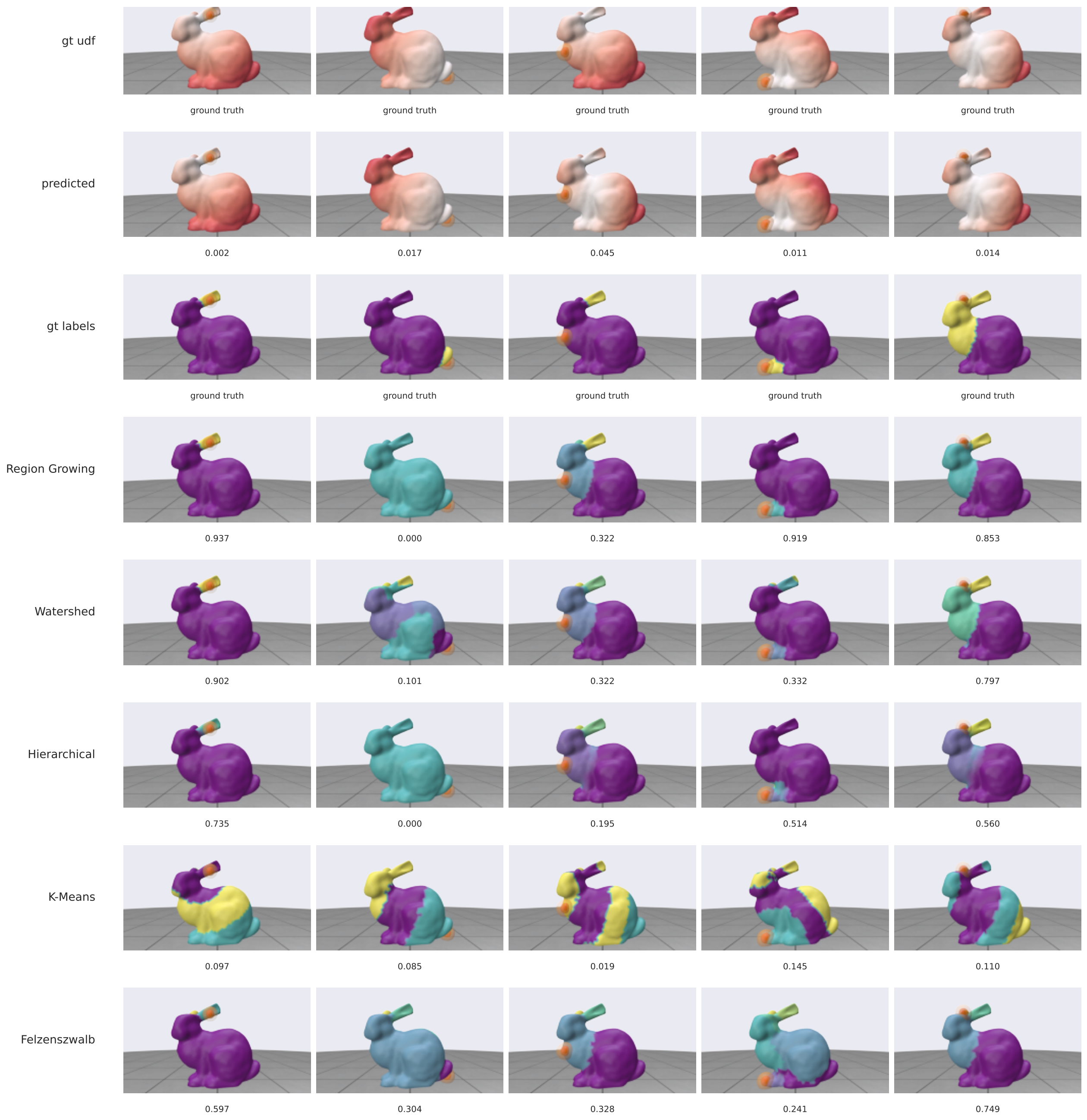


Figure 5.12: Segmentation results with DeltaConv. Rows (top to bottom): ground-truth unsigned distance field (UDF), predicted UDF, ground-truth segmentation, and results from region growing, watershed, hierarchical, k-means, and Felzenszwalb. Columns: dataset samples. In UDFs, white = near fracture, red = far. Segmentation colors are arbitrary and only distinguish regions. Numbers indicate Rand index for segmentations (higher is better) and AMAE for UDFs (lower is better). Red dots mark impact points.

hierarchical clustering. Watershed also had the highest runtime standard deviation for this mesh. In terms of BC, hierarchical outperformed both watershed and region growing. For ARI, region growing achieved the highest performance, with hierarchical ranking second. On this smaller mesh, hierarchical was more competitive with watershed in terms of runtime.

For the vase mesh all segmentation methods performed quite poor, with k-means again performing the worst. Watershed was the fastest method, with a very low runtime standard deviation. For BC, hierarchical outperformed the other methods. For ARI, hierarchical again achieved the best performance. On this mesh, hierarchical provided the overall strongest segmentation results.

5.5 Investigating Dataset Impact

In this section, we examine the influence of the dataset characteristics on the performance of our models. We compare the performance across multiple datasets that differ only in the resolution or size of the underlying mesh, while keeping all other experimental conditions constant. This approach allows us to isolate the effect of dataset scale and assess its contribution to model and segmentation performance.

Table 5.13 presents the performance of each segmentation method on the bunny mesh at varying resolutions. The meshes used were: the original (1×) with approximately 3,300 vertices, a 3× subdivided version with about 9,900 vertices, and a 9× subdivided version with around 29,000 vertices. The higher resolution meshes were generated in Blender using the Subdivide operation, which splits each triangle into three smaller ones while preserving the overall geometry. This changes the mesh density, allowing for a consistent comparison of the segmentation runtime relative to the number of vertices. Vertices were used as the metric indicating mesh size, as our segmentation methods generate a mesh-level labelling.

Segmentation Method	**1× (3.3k vert.)**	**3× (9.9k vert.)**	**9× (29.3k vert.)**
Felzenszwalb	672 ms	5,895 ms	50,666 ms
Hierarchical	357 ms	2,859 ms	26,133 ms
K-Means	67 ms	375 ms	2,631 ms
Region Growing	328 ms	1,671 ms	12,026 ms
Watershed	24 ms	62 ms	214 ms

Table 5.13: Segmentation runtime comparison on the bunny mesh at increasing resolutions. The meshes contain approximately 3.3k, 9.9k, and 29.3k vertices, corresponding to the original (1×), 3× subdivided, and 9× subdivided versions created in Blender.

We observe that Watershed is consistently the fastest segmentation method across all mesh resolutions. For the original mesh (3.3k vertices), it achieves a runtime of 24 ms, increasing to 62 ms for the 3× mesh (9.9k vertices) and 214 ms for the 9× mesh (29.3k vertices). This indicates a near-linear scaling with mesh size. Although Watershed never meets our 17 ms target threshold, it consistently comes closest among all tested methods.

K-Means is the second fastest, as discussed earlier, its segmentation performance is very poor. Region Growing, which previously performed the slowest on the vase and chair meshes, now ranks in the mid-range, with runtimes of 328 ms, 1,671 ms, and 12,026 ms across the three mesh sizes. Hierarchical segmentation performs slower than expected, requiring about 357 ms, 2,859 ms, and 26,133 ms respectively. Finally, Felzenszwalb is by far the slowest, reaching 50,666 ms at the highest resolution.

Overall, while Watershed demonstrates the best runtime performance and scalability, none of the evaluated methods satisfy the real-time constraint of 17 ms.

# Vertices	AMAE	MSE	Seg. Method	Duration (ms)	RC	BC	ARI	AMI
3302	0.0071	0.0242	Felzenszwalb	672.8	0.236	0.164	0.533	0.569
			Hierarchical	357.1	0.395	0.075	0.802	0.670
			K-Means	66.86	0.265	1.191	0.017	0.097
			Region Growing	328.7	**0.034**	**0.057**	**0.892**	**0.866**
			Watershed	**23.66**	0.043	0.080	0.855	0.816
9902	**0.0044**	0.0089	Felzenszwalb	5895	0.276	0.266	0.447	0.518
			Hierarchical	2859	0.371	**0.053**	0.835	0.653
			K-Means	375.1	0.258	1.274	0.023	0.012
			Region Growing	1671	0.345	0.061	0.793	0.694
			Watershed	**62.52**	**0.084**	0.067	**0.857**	**0.834**
29300	**0.0044**	**0.0050**	Felzenszwalb	50666	0.348	0.504	0.247	0.369
			Hierarchical	26132	0.370	**0.045**	**0.838**	0.620
			K-Means	2631	**0.255**	1.254	0.021	0.110
			Region Growing	12026	0.370	0.047	0.814	**0.648**
			Watershed	**214.5**	0.284	0.336	0.333	0.437

Table 5.14: Performance comparison of models across segmentation methods and mesh sizes represented by the number of vertices, with durations in milliseconds (three significant digits).

Table 5.14 shows all performance metrics across the three bunny mesh resolutions. A clear improvement in AMAE is observed when increasing from the 1× to the 3× mesh, after which the performance stabilizes for the 9× version. In contrast, MSE continues to improve consistently across all resolutions.

As previously discussed, Watershed achieves the best runtime performance at all scales. In terms of segmentation quality, the 1× mesh shows Region Growing as the top performer, with Watershed and Hierarchical close behind. For the 3× mesh, Watershed attains the highest RC, ARI, and AMI, while Hierarchical performs best in BC. On the 9× mesh, K-Means unexpectedly reports the lowest RC, while still performing poorly in all other metrics. In contrast Hierarchical achieves the best BC and ARI, and Region Growing yields the best AMI. Watershed shows a noticeable decrease in segmentation accuracy at this resolution.

Chapter 6

Discussion

This chapter presents a discussion of the results obtained in chapter 5 and outlines the limitations of the proposed method. In addition, it highlights several potential directions for future work, including the use of improved datasets and alternative fracture representations.

6.1 Comparing between convolutional models and MLP

Both models successfully learned the impulse-dependent fracture patterns and produced outputs suitable for our segmentation approach. However, the MLP consistently outperformed DeltaConv in both inference time and segmentation accuracy. Therefore, we consider the MLP the more suitable choice for real-time fracture prediction in our setting.

We believe DeltaConv performs poorly in this segmentation task because it is primarily designed to represent and understand geometric variations across different shapes. We tested if this ability would transfer to understanding different segmentations of the same shape under varying impulses, however, this was not the case. We were unable to use DeltaConv's multi-mesh capabilities for multi-impulse prediction on a single mesh.

Finally, we cannot conclude that convolutional networks, in general, are less effective than MLPs for this type of task. Further experiments with additional architectures and configurations would be required to make such a determination. We can however conclude that DeltaConv is less suitable than the MLP for predicting impulse-dependent distance fields in our current setup.

Compared to DeepFracture [79], which relies on autoencoders and requires up to a minute, our surface-based method nearly achieves real-time performance on small meshes, at the cost of not supporting internal fractures. Kim et al. instead directly learn the fracture labels using point-transformers, omitting the distance field and segmentation. Their method generalizes to unseen meshes, but does not work with impulse-dependent fractures. A direct comparison to our method is not possible, as the authors do not report their run-time performance.

Our real-time constraints favoured a MLP, outperforming DeltaConv. Whether autoencoders and point-based transformers offer advantages in a real-time setting is left for future work.

6.2 Comparing segmentation methods

All tested segmentation methods performed better on the output of the MLP model, compared to DeltaConv. While Hierarchical segmentation, region growing, and watershed each had the best performance for different metrics, watershed emerged as the most suitable method for a real-time setting.

Watershed consistently had the lowest runtime while performing comparably to the other methods. Although there are cases where Region Growing and Hierarchical Segmentation had better segmentation accuracy, these were marginal differences relative to Watershed's computational cost. Felzenszwalb and K-Means consistently performed poorly across all evaluated shapes and metrics.

Hierarchical segmentation tended to perform better under the Adjusted Rand Index than the Boundary Chamfer metric. Since ARI is less sensitive to small noisy regions, this suggests that Hierarchical segmentation produces many fragmented small regions. Since it was implemented without a merge step, performance improvements may be possible in this area.

All segmentation methods were evaluated using a single parameter configuration across all shapes, as we prioritized usability over shape-specific tuning, which would be impractical for real users.

DeepFracture employed Watershed segmentation for volumetric (3D) data, and using our results we come to the same conclusion for surface meshes [24, 39].

However, Watershed's runtime increases with mesh size, which constrains our ability to meet strict latency targets. These considerations motivate an alternative design: learning the segmentation directly rather than predicting a distance field followed by post-processing. By eliminating a separate segmentation stage, this strategy could reduce computational costs and allow for the use of more expressive models under the same real-time constraints.

6.3 The impact of the dataset and mesh sizes

Overall, as shown in Table 5.14, Watershed's segmentation quality declines noticeably with higher mesh resolutions, despite maintaining the best runtime performance. Region Growing remains consistent across scales but is significantly slower. Hierarchical segmentation appears to improve with increasing mesh size, suggesting better scalability for detailed meshes, though its runtime becomes unrealistic for interactive or game-engine use. Felzenszwalb segmentation and K-Means consistently produce worse segmentation quality.

Increasing the number of data points and mesh resolutions leads to better predictive accuracy, with both AMAE and MSE scores improving significantly. This suggests that better sampling methods, such as sampling triangle surfaces instead of only vertices, can help improve performance.

6.4 Learning impulse dependent distance field to represent fracture patterns in real-time

To answer our main research question, we developed a framework capable of predicting and segmenting impulse-dependent distance fields in real time for a single mesh. Using a combination of a Multi-Layer Perceptron (MLP) and Watershed segmentation, we successfully met the real-time constraint for the Vase and Chair meshes. However, the framework did not meet this requirement for the Bunny mesh, where we narrowly exceeded the real-time threshold

In terms of accuracy, the framework is able to reproduce fractures similar to the ground truth in up to 88 % of cases, depending on the mesh and segmentation algorithm used. On average, the accuracy threshold was met in two out of three tested meshes, indicating that the approach shows strong potential. However, this also means that at least 12 % of the generated fractures still differ significantly from the ground truth. While these results demonstrate that the framework can effectively encode and reproduce realistic, human-believable fractures at a high success rate, this visual quality comes at a notable computational cost.

Most experiments were conducted on relatively small meshes, yet even in these cases the framework struggled to achieve real-time performance. When tested on larger datasets, the segmentation methods required up to a full minute to process a single mesh, clearly exceeding the limits of real-time applicability. Even the fastest methods failed to meet the established real-time threshold, confirming that segmentation remains the primary computational bottleneck. Potential solutions include employing machine learning–based segmentation algorithms or optimizing existing implementations, though these challenges are expected to persist to some extent. Our findings indicate that distance-field-based fracture prediction is feasible for small meshes, but not for larger meshes in a real-time context.

6.5 Limitations and Future Work

Dataset generation

The dataset and its generation pipeline have several limitations. Most notably, the prefracturing step limits the outcomes to a set of finite candidate crack surfaces. This discretization reduces the diversity of fractures represented in the data and may bias the learned models toward those predefined patterns, creating repeating fracture lines.

A logical improvement is to construct a more realistic dataset using physics-based fracture simulations that better capture material behaviour and crack evolution. Our current generation algorithm is optimized for real-time performance, rather than physical fidelity, limiting its realism. Physics-based simulations would produce richer and more diverse fractures. This would increase the computational cost for generating the dataset, but as a precomputational step this would definitely be worth it. In case new public datasets suitable for our purpose were to be created, they could also be used, though none exist as of now.

Dataset Mapping stage

We employ a simple mapping algorithm that attempts to aligns only the extremes and assumes that this suffices to match the meshes. However, since the simulation mesh is a downsampled version of the original mesh, certain geometric details, such as small extrusions, may be simplified or lost. This can disrupt the correspondence between the ground-truth and simulation meshes. Even minor errors can lead to incorrect labeling and produce unrealistic ground truths. This issue is particularly evident for the vase model, where the inner and outer surfaces of the mesh are in close proximity.

It is important that this limitation is addressed, as in its current form some meshes cannot be reliably used. Ideally, the label information would be generated directly on the ground-truth mesh, removing the need for the mapping stage altogether. This would substantially simplify the pipeline and yield cleaner data for learning. If this is not feasible, the use of more sophisticated and accurate mapping methods is strongly recommended.

Biased dataset

Because our objective is to learn fracture patterns, we exclude all simulated outcomes in which no fracture occurs, which happens quite often when generating a dataset with Breaking Good. Non-fracture cases occur disproportionately when impacts are located at regions not identified by Breaking good as weak. Therefore, this filtering produces an uneven distribution of impact-point–fracture-pattern pairs. In effect, the dataset is biased toward structurally weaker areas, and models trained on it become more likely to predict fractures at those locations, amplifying weaker regions to be even more likely to fracture.

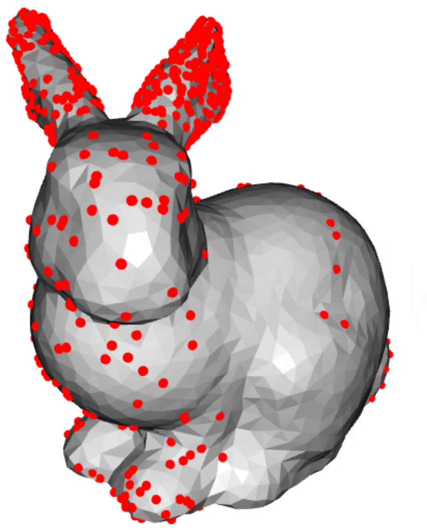


Figure 6.1: Visualization of biased dataset after discarding non-fracture inputs. We show the impact positions of all samples in our dataset.

For example, in the bunny mesh shown in Figure 6.1, the structurally weaker ears have a dense concentration of training datapoints, whereas the body, despite being larger, is structurally stronger and therefore has substantially fewer impact positions. Consequently, there occur a disproportionate amount of ear fracture, as we observed in the example fractures in the previous chapter.

As a consequence, the models regress well-formed fractures on the bunny's ears but can struggle to generate complete fractures elsewhere, such as a mid-body break. This happens partly because the ears are thin and therefore need shorter fracture line to be fully formed, whereas larger fractures, such as full body breaks, require longer unobstructed fracture lines to remain continuously connected. But this also happens because these longer fractures are underrepresented in the training data.

Evaluating Learning Architectures

Contrary to our expectations, the convolutional model did not demonstrate a performance improvement over the MLP, we actually saw the opposite. Due to time constraints, only DeltaConv was evaluated. To better assess whether this observation holds for convolutional architectures in general, it would have been preferable to test additional models, such as the various iterations of PointNet [52, 53, 82].

Further investigation is also warranted into whether architectures beyond convolutional models and MLPs offer performance advantages. Autoencoders, for instance, have been applied successfully in mesh segmentation tasks [83, 44] and in DeepFracture [24]. Exploring their applicability in real-time scenarios would be an interesting direction for future work.

Applicability to real-time

Our approach is promising for interactive settings, but scalability to real-time is limited, particularly on large meshes. The segmentation stage dominates the runtime, which makes regressing and segmenting a distance field a poor fit for real-time scenarios.

To address this, future work should focus on directly learning the fracture labels, omitting the segmentation stage altogether. Other options could be using light-weigh neural segmentation modules or using pre-partitioned meshes (e.g. Voronoi cells) to trade realism for improved performance.

Metrics and validating performance

Evaluating fracture prediction quality is nontrivial. There is no single, concise measure that captures both geometric fidelity and perceived realism. Although our models aim to reproduce exact fracture patterns, multiple plausible outcomes may be acceptable, or even preferable, from a user's perspective.

We experimented with point-cloud similarity metrics to quantify how closely the predicted segmentations match the ground truth. While these measures are convenient, they do not assess whether a result looks physically plausible to human observers. A prediction can be geometrically close yet appear "off," and, conversely, a fracture in a different region can still be acceptable in interactive settings.

To better target crack placement, we also evaluate with a boundary Chamfer metric that measures the distance between the predicted fracture curve and the ground-truth fracture line. This improves sensitivity to local alignment but remains blind to important topological effects: small boundary shifts can produce very different outcomes. For example, in the bunny model, a boundary that detaches both ears can be very similar to one that detaches only a single ear, producing a low error despite a perceptually unrealistic result for the intended scenario. These limitations suggest that, for robust validation, a combination of mesh-aware metrics and human perception is needed.

Refracturing and generalization

The current approach is fixed for a single mesh: we train a separate model for each mesh and do not target cross-mesh generalization. As a result, the method does not support re-fracturing. After an initial break, the geometry and topology change, and the model is not designed to update its predictions for following fractures.

This lack of generalization has practical consequences. Users would need to maintain and switch between multiple mesh-specific models at runtime, which may introduce loading overhead, memory consumption, and code complexity. These factors can undermine the responsiveness required for real-time systems.

Our current approach trains a separate model for a single mesh. A more follow-up goal is a single model that generalizes across meshes with varying topologies, scales, and material properties. Such a model would remove the need to maintain multiple mesh-specific models and would reduce the burden of generating datasets for each new asset, at the possible cost of higher complexity.

This requires mesh encodings that transfer across objects, something DeltaConv actually supports and we expect convolutional models to outperform MLPs on.

Direct Comparisons

The authors of DeepFracture did not provide their dataset for us to compare to, nor did they provide a codebase for us to run. Ideally, we would have run their method with all our metrics, so a direct comparison would have been possible, but this was unfortunately not provided publicly.

Segmentation strategies

The segmentation stage remains the primary bottleneck. On small meshes, current methods are feasible for strict real-time budgets, but on larger meshes they are unlikely to reach real-time speeds. Rewriting the critical parts in a compiled language could make things faster, but it probably won't eliminate the bottleneck entirely.

We identify three directions for reducing the segmentation bottleneck: (i) learn labels directly, removing the separate segmentation stage; (ii) investigate ML-based segmentation methods that can operate at interactive rates; and (iii) use the UDF solely as an internal representation within the network rather than an explicit output requiring post-processing.

Impulse representation

Further work is needed on how the impulse is represented and passed to the system. In the current approach, the impulse is simply appended to all mesh coordinates. Alternatively, the impulse could be passed into the model as a separate vector, which could in turn be preprocessed similarly to how the latent vector in image generation model such as StyleGAN3 [29] is handled. Another possibility is to rotate and translate the mesh in such a way that impulse direction and impact location are implicitly encoded in the coordinates, removing the need for an impulse vector altogether.

Furthermore, depending on the dataset, additional impulse-related information may be available. Providing control over parameters such as impulse strength, as well as incorporating material properties of the interacting objects, could lead to a more comprehensive and physically grounded model.

Chapter 7

Conclusion

This thesis introduced a machine-learning–guided framework for impulse-dependent fracture simulation that operates in real time. We constructed a dedicated dataset to support learning on surface meshes and proposed quantitative metrics to evaluate performance. Central to our approach is a learned unsigned distance field (UDF) defined on the mesh surface; this field is subsequently processed by a segmentation algorithm to produce fracture labels.

Our study demonstrated that the method reliably produces impulse-dependent fracture labellings in 2 out of 3 evaluated scenarios. Notably, a comparatively simple MLP proved more effective than the more complex DeltaConv for learning the surface UDF, outperforming it in both speed and accuracy. Among the segmentation strategies assessed, Watershed delivered the strongest overall performance. Furthermore, we showed that the current bottleneck for the approach to reach real-time performance is the segmentation step.

Overall, we can currently say that 2D impulse-dependent UDF-guided fractures are not suitable for real-time video game environments. However, our findings indicate a meaningful advance toward machine-learning-based, impulse-conditioned segmentation in fracture simulation, with the potential to support faster and more realistic pipelines. We hope this encourages more research into machine learning-based fracture segmentation methodologies and their applicability to real-time scenarios.

Bibliography

[1] R. Adams and L. Bischof. Seeded region growing. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(6):641–647, June 1994. ISSN 1939-3539. doi: 10.1109/34.295913. URL `https://ieeexplore.ieee.org/abstract/document/295913`.

[2] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A Next-generation Hyperparameter Optimization Framework, July 2019. URL `https://arxiv.org/abs/1907.10902v1`.

[3] Pablo Arbelaez, Michael Maire, Charless Fowlkes, and Jitendra Malik. From contours to regions: An empirical evaluation. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 2294–2301, June 2009. doi: 10.1109/CVPR.2009.5206707. URL `https://ieeexplore.ieee.org/abstract/document/5206707`. ISSN: 1063-6919.

[4] Electronic Arts. THE POWER OF PC MEETS THE POTENTIAL OF PORTAL, 2025. URL `https://www.ea.com/games/battlefield/battlefield-6/news/pc-portal-experience`.

[5] Marco Attene, Bianca Falcidieno, and Michela Spagnuolo. Hierarchical mesh segmentation based on fitting primitives. *The Visual Computer*, 22(3):181–193, March 2006. ISSN 1432-2315. doi: 10.1007/s00371-006-0375-x. URL `https://doi.org/10.1007/s00371-006-0375-x`.

[6] H. G. Barrow, J. M. Tenenbaum, R. C. Bolles, and H. C. Wolf. Parametric Correspondence and Chamfer Matching: Two New Techniques for Image Matching. January 1977. URL `https://apps.dtic.mil/sti/html/tr/ADA458355/`. Number: TN153.

[7] S. Beucher and F. Meyer. The Morphological Approach to Segmentation: The Watershed Transformation. In *Mathematical Morphology in Image Processing*. CRC Press, 1992. Num Pages: 49.

[8] Oleksiy Busaryev, Tamal K. Dey, and Huamin Wang. Adaptive fracture simulation of multi-layered thin plates. *ACM Trans. Graph.*, 32(4):52:1–52:6, July 2013. ISSN 0730-0301. doi: 10.1145/2461912.2461920. URL `https://dl.acm.org/doi/10.1145/2461912.2461920`.

[9] Julian Chibane, Aymen Mir, and Gerard Pons-Moll. Neural Unsigned Distance Fields for Implicit Function Learning, October 2020. URL `http://arxiv.org/abs/2010.13938`. arXiv:2010.13938.

[10] Floyd M. Chitalu, Qinghai Miao, Kartic Subr, and Taku Komura. Displacement-Correlated XFEM for Simulating Brittle Fracture. *Computer Graphics Forum*, 39 (2):569–583, 2020. ISSN 1467-8659. doi: 10.1111/cgf.13953. URL `https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13953`. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13953.

[11] Christopher Choy, JunYoung Gwak, and Silvio Savarese. 4D Spatio-Temporal ConvNets: Minkowski Convolutional Neural Networks. pages 3075–3084, 2019. URL `https://openaccess.thecvf.com/content_CVPR_2019/html/Choy_4D_Spatio-Temporal_ConvNets_Minkowski_Convolutional_Neural_Networks_CVPR_2019_paper.html`.

[12] Mark Claypool, Kajal Claypool, and Feissal Damaa. The effects of frame rate and resolution on users playing first person shooter games. page 607101, San Jose, CA, January 2006. doi: 10.1117/12.648609. URL `http://proceedings.spiedigitallibrary.org/proceeding.aspx?doi=10.1117/12.648609`.

[13] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2): 182–197, April 2002. ISSN 1941-0026. doi: 10.1109/4235.996017. URL `https://ieeexplore.ieee.org/document/996017`.

[14] Gouri Dhatt, Emmanuel Lefrançois, and Gilbert Touzot. *Finite Element Method*. John Wiley & Sons, December 2012. ISBN 978-1-118-56970-2. Google-Books-ID: wvE1ClcHq5IC.

[15] M.-P. Dubuisson and A.K. Jain. A modified Hausdorff distance for object matching. In *Proceedings of 12th International Conference on Pattern Recognition*, volume 1, pages 566–568 vol.1, October 1994. doi: 10.1109/ICPR.1994.576361. URL `https://ieeexplore.ieee.org/abstract/document/576361`.

[16] Linxu Fan, Floyd M. Chitalu, and Taku Komura. Simulating Brittle Fracture with Material Points. *ACM Trans. Graph.*, 41(5):177:1–177:20, 2022. ISSN 0730-0301. doi: 10.1145/3522573. URL `https://dl.acm.org/doi/10.1145/3522573`.

[17] Pedro F. Felzenszwalb and Daniel P. Huttenlocher. Efficient Graph-Based Image Segmentation. *International Journal of Computer Vision*, 59(2):167–181, September 2004. ISSN 1573-1405. doi: 10.1023/B:VISI.0000022288.19776.77. URL `https://doi.org/10.1023/B:VISI.0000022288.19776.77`.

[18] Loeiz Glondu, Maud Marchal, and Georges Dumont. Real-Time Simulation of Brittle Fracture Using Modal Analysis. *IEEE Transactions on Visualization and Computer Graphics*, 19(2):201–209, February 2013. ISSN 1941-0506. doi: 10.1109/TVCG.2012.121. URL `https://ieeexplore.ieee.org/abstract/document/6197190?casa_token=PAO5avGGPRwAAAAA:BdFctVgLo9_myene0IKZUVMIsAAOiqHciYEMF2U-r07MR1BvqG7Bh9Unrk4LdsvzFJg6Wub3`. Conference Name: IEEE Transactions on Visualization and Computer Graphics.

[19] David Hahn and Chris Wojtan. High-resolution brittle fracture simulation with boundary elements. *ACM Trans. Graph.*, 34(4):151:1–151:12, July 2015. ISSN 0730-0301. doi: 10.1145/2766896. URL `https://dl.acm.org/doi/10.1145/2766896`.

[20] David Hahn and Chris Wojtan. Fast approximations for boundary element based brittle fracture simulation. *ACM Transactions on Graphics*, 35(4):1–11, July 2016. ISSN 0730-0301, 1557-7368. doi: 10.1145/2897824.2925902. URL `https://dl.acm.org/doi/10.1145/2897824.2925902`.

[21] W. S. Hall. Boundary Element Method. In W. S. Hall, editor, *The Boundary Element Method*, pages 61–83. Springer Netherlands, Dordrecht, 1994. ISBN 978-94-011-0784-6. doi: 10.1007/978-94-011-0784-6_3. URL `https://doi.org/10.1007/978-94-011-0784-6_3`.

[22] Rana Hanocka, Amir Hertz, Noa Fish, Raja Giryes, Shachar Fleishman, and Daniel Cohen-Or. MeshCNN: A Network with an Edge, February 2019. URL `http://arxiv.org/abs/1809.05910`. arXiv:1809.05910.

[23] Robert M. Haralick, Stanley R. Sternberg, and Xinhua Zhuang. Image Analysis Using Mathematical Morphology. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-9(4):532–550, July 1987. ISSN 1939-3539. doi: 10.1109/TPAMI.1987.4767941. URL `https://ieeexplore.ieee.org/abstract/document/4767941`.

[24] Yuhang Huang and Takashi Kanai. DeepFracture: A Generative Approach for Predicting Brittle Fractures, October 2023. URL `http://arxiv.org/abs/2310.13344`. arXiv:2310.13344 [cs].

[25] Lawrence Hubert and Phipps Arabie. Comparing partitions. *Journal of Classification*, 2(1):193–218, December 1985. ISSN 1432-1343. doi: 10.1007/BF01908075. URL `https://doi.org/10.1007/BF01908075`.

[26] F. Hutter, H. Hoos, and K. Leyton-Brown. An efficient approach for assessing hyperparameter importance. In *Proceedings of International Conference on Machine Learning 2014 (ICML 2014)*, page 754–762, June 2014.

[27] Benjamin F. Janzen and Robert J. Teather. Is 60 FPS better than 30? the impact of frame rate and latency on moving target selection. In *CHI '14 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '14, pages 1477–1482, New York, NY, USA, April 2014. Association for Computing Machinery. ISBN 978-1-4503-2474-8. doi: 10.1145/2559206.2581214. URL `https://doi.org/10.1145/2559206.2581214`.

[28] Christian Jung and Claudia Redenbach. Crack modeling via minimum-weight surfaces in 3d Voronoi diagrams. *Journal of Mathematics in Industry*, 13(1):10, November 2023. ISSN 2190-5983. doi: 10.1186/s13362-023-00138-1. URL `https://doi.org/10.1186/s13362-023-00138-1`.

[29] Tero Karras, Miika Aittala, Samuli Laine, Erik Härkönen, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. Alias-Free Generative Adversarial Networks, October 2021. URL `http://arxiv.org/abs/2106.12423`. arXiv:2106.12423 [cs].

[30] Seunghwan Kim, Sunha Park, and Seungkyu Lee. Neural Clustering for Prefractured Mesh Generation in Real-time Object Destruction. In *SIGGRAPH Asia 2024 Posters*, pages 1–2, Tokyo Japan, December 2024. ACM. ISBN 9798400711381. doi: 10.1145/3681756.3697973. URL `https://dl.acm.org/doi/10.1145/3681756.3697973`.

[31] Dan Koschier, Sebastian Lipponer, and Jan Bender. Adaptive tetrahedral meshes for brittle fracture simulation. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '14, pages 57–66, Goslar, DEU, July 2015. Eurographics Association.

[32] Yu-Kun Lai, Shi-Min Hu, Ralph R. Martin, and Paul L. Rosin. Fast mesh segmentation using random walks. In *Proceedings of the 2008 ACM symposium on Solid and physical modeling*, SPM '08, pages 183–191, New York, NY, USA, June 2008. Association for Computing Machinery. ISBN 978-1-60558-106-4. doi: 10.1145/1364901.1364927. URL `https://doi.org/10.1145/1364901.1364927`.

[33] Nikolas Lamb, Cameron Palmer, Benjamin Molloy, Sean Banerjee, and Natasha Kholgade Banerjee. Fantastic Breaks: A Dataset of Paired 3D Scans of Real-World Broken Objects and Their Complete Counterparts, May 2023. URL `http://arxiv.org/abs/2303.14152`. arXiv:2303.14152 [cs].

[34] Guillaume Lavoué and Christian Wolf. Markov Random Fields for Improving 3D Mesh Analysis and Segmentation. In *Eurographics 2008 Workshop on 3D Object Retrieval*, pages 1–8, Crete, Greece, April 2008. URL `https://hal.science/hal-01500680`.

[35] J. A. Levine, A. W. Bargteil, C. Corsi, J. Tessendorf, and R. Geist. A peridynamic perspective on spring-mass fracture. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '14, pages 47–55, Goslar, DEU, July 2015. Eurographics Association.

[36] Rong Liu and Hao Zhang. Segmentation of 3D meshes through spectral clustering. In *12th Pacific Conference on Computer Graphics and Applications, 2004. PG 2004. Proceedings.*, pages 298–305, October 2004. doi: 10.1109/PCCGA.2004.1348360. URL `https://ieeexplore.ieee.org/abstract/document/1348360`. ISSN: 1550-4085.

[37] Shengmei Liu, Atsuo Kuwahara, James J Scovell, and Mark Claypool. The Effects of Frame Rate Variation on Game Player Quality of Experience. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, pages 1–10, Hamburg Germany, April 2023. ACM. ISBN 978-1-4503-9421-5. doi: 10.1145/3544548.3580665. URL `https://dl.acm.org/doi/10.1145/3544548.3580665`.

[38] S. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, March 1982. ISSN 1557-9654. doi: 10.1109/TIT.1982.1056489. URL `https://ieeexplore.ieee.org/abstract/document/1056489`.

[39] A.P. Mangan and R.T. Whitaker. Partitioning 3D surface meshes using watershed segmentation. *IEEE Transactions on Visualization and Computer Graphics*, 5(4):308–321, October 1999. ISSN 1941-0506. doi: 10.1109/2945.817348. URL `https://ieeexplore.ieee.org/abstract/document/817348`.

[40] H. G. Maschke and M. Kuna. A review of boundary and finite element methods in fracture mechanics. *Theoretical and Applied Fracture Mechanics*, 4(3): 181–189, December 1985. ISSN 0167-8442. doi: 10.1016/0167-8442(85)90003-5. URL `https://www.sciencedirect.com/science/article/pii/0167844285900035`.

[41] Daniel Maturana and Sebastian Scherer. VoxNet: A 3D Convolutional Neural Network for real-time object recognition. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 922–928, Hamburg, Germany, September 2015. IEEE. ISBN 978-1-4799-9994-1. doi: 10.1109/IROS.2015.7353481. URL `http://ieeexplore.ieee.org/document/7353481/`.

[42] Francesco Milano, Antonio Loquercio, Antoni Rosinol, Davide Scaramuzza, and Luca Carlone. Primal-Dual Mesh Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, volume 33, pages 952–963. Curran Associates, Inc., 2020. URL `https://proceedings.neurips.cc/paper/2020/hash/0a656cc19f3f5b41530182a9e03982a4-Abstract.html`.

[43] Matthias Müller, Nuttapong Chentanez, and Tae-Yong Kim. Real time dynamic fracture with volumetric approximate convex decompositions. *ACM Transactions on Graphics*, 32(4):115:1–115:10, July 2013. ISSN 0730-0301. doi: 10.1145/2461912.2461934. URL `https://doi.org/10.1145/2461912.2461934`.

[44] Danish Nazir, Muhammad Zeshan Afzal, Alain Pagani, Marcus Liwicki, and Didier Stricker. Contrastive Learning for 3D Point Clouds Classification and Shape Completion. *Sensors*, 21(21):7392, January 2021. ISSN 1424-8220. doi: 10.3390/s21217392. URL `https://www.mdpi.com/1424-8220/21/21/7392`. Number: 21 Publisher: Multidisciplinary Digital Publishing Institute.

[45] J. C. Newman. The merging of fatigue and fracture mechanics concepts: a historical perspective. *Progress in Aerospace Sciences*, 34(5):347–390, July 1998. ISSN 0376-0421. doi: 10.1016/S0376-0421(98)00006-2. URL `https://www.sciencedirect.com/science/article/pii/S0376042198000062`.

[46] Alan Norton, Greg Turk, Bob Bacon, John Gerth, and Paula Sweeney. Animation of fracture by physical modeling. *The Visual Computer*, 7(4):210–219, July 1991. ISSN 1432-2315. doi: 10.1007/BF01900837. URL `https://doi.org/10.1007/BF01900837`.

[47] James F. O'Brien and Jessica K. Hodgins. Graphical modeling and animation of brittle fracture. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques - SIGGRAPH '99*, pages 137–146, Not Known, 1999. ACM Press. ISBN 978-0-201-48560-8. doi: 10.1145/311535.311550. URL `http://portal.acm.org/citation.cfm?doid=311535.311550`.

[48] James F. O'Brien, Adam W. Bargteil, and Jessica K. Hodgins. Graphical modeling and animation of ductile fracture. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '02, pages 291–294, New York, NY, USA, July 2002. Association for Computing Machinery. ISBN 978-1-58113-521-3. doi: 10.1145/566570.566579. URL `https://dl.acm.org/doi/10.1145/566570.566579`.

[49] Jeong Joon Park, Peter Florence, Julian Straub, Richard Newcombe, and Steven Lovegrove. DeepSDF: Learning Continuous Signed Distance Functions for Shape Representation, January 2019. URL `http://arxiv.org/abs/1901.05103`. arXiv:1901.05103.

[50] Eric G. Parker and James F. O'Brien. Real-time deformation and fracture in a game environment. In *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '09, pages 165–175, New York, NY, USA, August 2009. Association for Computing Machinery. ISBN 978-1-60558-610-6. doi: 10.1145/1599470.1599492. URL `https://dl.acm.org/doi/10.1145/1599470.1599492`.

[51] Mark Pauly, Richard Keiser, Bart Adams, Philip Dutré, Markus Gross, and Leonidas J. Guibas. Meshless animation of fracturing solids. *ACM Trans. Graph.*, 24(3):957–964, July 2005. ISSN 0730-0301. doi: 10.1145/1073204.1073296. URL `https://dl.acm.org/doi/10.1145/1073204.1073296`.

[52] Charles R. Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation, April 2017. URL http://arxiv.org/abs/1612.00593. arXiv:1612.00593 [cs].

[53] Charles R. Qi, Li Yi, Hao Su, and Leonidas J. Guibas. PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space, June 2017. URL http://arxiv.org/abs/1706.02413. arXiv:1706.02413 [cs].

[54] Guocheng Qian, Yuchen Li, Houwen Peng, Jinjie Mai, Hasan Abed Al Kader Hammoud, Mohamed Elhoseiny, and Bernard Ghanem. PointNeXt: Revisiting PointNet++ with Improved Training and Scaling Strategies, October 2022. URL http://arxiv.org/abs/2206.04670. arXiv:2206.04670 [cs].

[55] Saty Raghavachary. Fracture generation on polygonal meshes using Voronoi polygons. In *ACM SIGGRAPH 2002 conference abstracts and applications*, SIGGRAPH '02, page 187, New York, NY, USA, July 2002. Association for Computing Machinery. ISBN 978-1-58113-525-1. doi: 10.1145/1242073.1242200. URL https://dl.acm.org/doi/10.1145/1242073.1242200.

[56] Soumik Rakshit. Efficient Graph-Based Image Segmentation, September 2020. URL https://soumik12345.github.io/geekyrakshit-blog/algebra/computervision/convolution/maths/python/2020/09/17/efficient-graph-based-image-segmentation.html.

[57] William M. Rand. Objective Criteria for the Evaluation of Clustering Methods. *Journal of the American Statistical Association*, 66(336):846–850, December 1971. ISSN 0162-1459. doi: 10.1080/01621459.1971.10482356. URL https://www.tandfonline.com/doi/abs/10.1080/01621459.1971.10482356. Publisher: ASA Website _eprint: https://www.tandfonline.com/doi/pdf/10.1080/01621459.1971.10482356.

[58] Gernot Riegler, Ali Osman Ulusoy, and Andreas Geiger. OctNet: Learning Deep 3D Representations at High Resolutions, April 2017. URL http://arxiv.org/abs/1611.05009. arXiv:1611.05009 [cs].

[59] Rui S. V. Rodrigues, José F. M. Morgado, and Abel J. P. Gomes. Part-Based Mesh Segmentation: A Survey. *Computer Graphics Forum*, 37(6):235–274, 2018. ISSN 1467-8659. doi: 10.1111/cgf.13323. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13323. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13323.

[60] Stanley Theodore Rolfe and John M. Barsom. *Fracture and Fatigue Control in Structures: Applications of Fracture Mechanics*. ASTM International, 1977. Google-Books-ID: t3N8CtDxfzYC.

[61] Sara C. Schvartzman and Miguel A. Otaduy. Fracture animation based on high-dimensional Voronoi diagrams. In *Proceedings of the 18th meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '14, pages 15–22,

New York, NY, USA, 2014. Association for Computing Machinery. ISBN 978-1-4503-2717-6. doi: 10.1145/2556700.2556713. URL https://dl.acm.org/doi/10.1145/2556700.2556713.

[62] Silvia Sellán, Yun-Chun Chen, Ziyi Wu, Animesh Garg, and Alec Jacobson. Breaking Bad: A Dataset for Geometric Fracture and Reassembly. *Advances in Neural Information Processing Systems*, 35:38885–38898, December 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/fe18f2090bf1e0fd5a1ded5bdd7ca351-Abstract-Datasets_and_Benchmarks.html.

[63] Silvia Sellán, Jack Luong, Leticia Mattos Da Silva, Aravind Ramakrishnan, Yuchuan Yang, and Alec Jacobson. Breaking Good: Fracture Modes for Realtime Destruction, July 2022. URL http://arxiv.org/abs/2111.05249. arXiv:2111.05249 [cs].

[64] Ariel Shamir. A survey on Mesh Segmentation Techniques. *Computer Graphics Forum*, 27(6):1539–1556, 2008. ISSN 1467-8659. doi: 10.1111/j.1467-8659.2007.01103.x. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2007.01103.x. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2007.01103.x.

[65] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, July 1948. ISSN 0005-8580. doi: 10.1002/j.1538-7305.1948.tb01338.x. URL https://ieeexplore.ieee.org/abstract/document/6773024.

[66] Lior Shapira, Ariel Shamir, and Daniel Cohen-Or. Consistent mesh partitioning and skeletonisation using the shape diameter function. *The Visual Computer*, 24(4):249–259, April 2008. ISSN 1432-2315. doi: 10.1007/s00371-007-0197-5. URL https://doi.org/10.1007/s00371-007-0197-5.

[67] Vincent Sitzmann, Eric R. Chan, Richard Tucker, Noah Snavely, and Gordon Wetzstein. MetaSDF: Meta-learning Signed Distance Functions, June 2020. URL http://arxiv.org/abs/2006.09662. arXiv:2006.09662.

[68] Alexander Strehl and Joydeep Ghosh. Cluster Ensembles — A Knowledge Reuse Framework for Combining Multiple Partitions. *Journal of Machine Learning Research*, 3(Dec):583–617, 2002. ISSN ISSN 1533-7928. URL https://www.jmlr.org/papers/v3/strehl02a.html.

[69] Jonathan Su, Craig Schroeder, and Ronald Fedkiw. Energy stability and fracture for frame rate rigid body simulations. In *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '09, pages 155–164, New York, NY, USA, August 2009. Association for Computing Machinery. ISBN 978-1-60558-610-6. doi: 10.1145/1599470.1599491. URL https://dl.acm.org/doi/10.1145/1599470.1599491.

[70] N. Sukumar, N. Moës, B. Moran, and T. Belytschko. Extended finite element method for three-dimensional crack modelling. *International Journal for Numerical Methods in Engineering*, 48(11):1549–1570, 2000. ISSN 1097-0207. doi: 10.1002/1097-0207(20000820)48:11⟨1549::AID-NME955⟩3.0.CO;2-A. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/1097-0207%2820000820%2948%3A11%3C1549%3A%3AAID-NME955%3E3.0.CO%3B2-A. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/1097-0207%2820000820%2948%3A11%3C1549%3A%3AAID-NME955%3E3.0.CO%3B2-A.

[71] Y. M. Tang, A. F. Zhou, and K. C. Hui. Comparison between FEM and BEM for Real-time Simulation. *Computer-Aided Design and Applications*, 2(1-4):421–430, January 2005. ISSN 1686-4360. doi: 10.1080/16864360.2005.10738391. URL http://www.cad-journal.net/files/vol_2/Vol2Nos1-4.html.

[72] Greg Turk and Marc Levoy. The stanford 3d scanning repository. http://graphics.stanford.edu/data/3Dscanrep/, 1994.

[73] Nguyen Xuan Vinh, Julien Epps, and James Bailey. Information theoretic measures for clusterings comparison: is a correction for chance necessary? In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 1073–1080, New York, NY, USA, June 2009. Association for Computing Machinery. ISBN 978-1-60558-516-1. doi: 10.1145/1553374.1553511. URL https://doi.org/10.1145/1553374.1553511.

[74] Stephanie Wang, Mengyuan Ding, Theodore F. Gast, Leyi Zhu, Steven Gagniere, Chenfanfu Jiang, and Joseph M. Teran. Simulation and Visualization of Ductile Fracture with the Material Point Method. *Proc. ACM Comput. Graph. Interact. Tech.*, 2 (2):18:1–18:20, July 2019. doi: 10.1145/3340259. URL https://dl.acm.org/doi/10.1145/3340259.

[75] Yinan Wang, Diane Oyen, Weihong (Grace) Guo, Anishi Mehta, Cory Braker Scott, Nishant Panda, M. Giselle Fernández-Godino, Gowri Srinivasan, and Xiaowei Yue. StressNet - Deep learning to predict stress with fracture propagation in brittle materials. *npj Materials Degradation*, 5(1):1–10, February 2021. ISSN 2397-2106. doi: 10.1038/s41529-021-00151-y. URL https://www.nature.com/articles/s41529-021-00151-y. Publisher: Nature Publishing Group.

[76] Yue Wang, Yongbin Sun, Ziwei Liu, Sanjay E. Sarma, Michael M. Bronstein, and Justin M. Solomon. Dynamic Graph CNN for Learning on Point Clouds, June 2019. URL http://arxiv.org/abs/1801.07829. arXiv:1801.07829 [cs].

[77] Joe H. Ward Jr. Hierarchical Grouping to Optimize an Objective Function. *Journal of the American Statistical Association*, 58(301):236–244, March 1963. ISSN 0162-1459. doi: 10.1080/01621459.1963.10500845. URL https://www.tandfonline.com/doi/abs/10.1080/

01621459.1963.10500845. Publisher: ASA Website _eprint: https://www.tandfonline.com/doi/pdf/10.1080/01621459.1963.10500845.

[78] Joachim Weickert. Anisotropic Diffusion in Image Processing. 1998.

[79] Ruben Wiersma, Ahmad Nasikun, Elmar Eisemann, and Klaus Hildebrandt. DeltaConv: Anisotropic Operators for Geometric Deep Learning on Point Clouds. *ACM Transactions on Graphics*, 41(4):1–10, July 2022. ISSN 0730-0301, 1557-7368. doi: 10.1145/3528223.3530166. URL http://arxiv.org/abs/2111.08799. arXiv:2111.08799 [cs].

[80] Jiajun Wu, Chengkai Zhang, Tianfan Xue, Bill Freeman, and Josh Tenenbaum. Learning a Probabilistic Latent Space of Object Shapes via 3D Generative-Adversarial Modeling. In *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016. URL https://proceedings.neurips.cc/paper_files/paper/2016/hash/44f683a84163b3523afe57c2e008bc8c-Abstract.html.

[81] Zhirong Wu, Shuran Song, Aditya Khosla, Fisher Yu, Linguang Zhang, Xiaoou Tang, and Jianxiong Xiao. 3D ShapeNets: A Deep Representation for Volumetric Shapes, April 2015. URL http://arxiv.org/abs/1406.5670. arXiv:1406.5670 [cs].

[82] Saining Xie, Jiatao Gu, Demi Guo, Charles R. Qi, Leonidas Guibas, and Or Litany. PointContrast: Unsupervised Pre-training for 3D Point Cloud Understanding. In Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm, editors, *Computer Vision – ECCV 2020*, pages 574–591, Cham, 2020. Springer International Publishing. ISBN 978-3-030-58580-8. doi: 10.1007/978-3-030-58580-8_34.

[83] Yaoqing Yang, Chen Feng, Yiru Shen, and Dong Tian. FoldingNet: Point Cloud Autoencoder via Deep Grid Deformation, April 2018. URL http://arxiv.org/abs/1712.07262. arXiv:1712.07262 [cs] version: 2.

[84] Yunjin Lee, Seungyong Lee, and A. Shamir. Intelligent mesh scissoring using 3D snakes. In *12th Pacific Conference on Computer Graphics and Applications, 2004. PG 2004. Proceedings.*, pages 279–287, Seoul, Korea, 2004. IEEE. ISBN 978-0-7695-2234-0. doi: 10.1109/PCCGA.2004.1348358. URL http://ieeexplore.ieee.org/document/1348358/.

[85] Youyi Zheng and Chiew-Lan Tai. Mesh Decomposition with Cross-Boundary Brushes. *Computer Graphics Forum*, 29(2):527–535, 2010. ISSN 1467-8659. doi: 10.1111/j.1467-8659.2009.01622.x. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2009.01622.x. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2009.01622.x.

[86] Yufeng Zhu, Robert Bridson, and Chen Greif. Simulating rigid body fracture with surface meshes. *ACM Transactions on Graphics*, 34(4):1–11, July 2015. ISSN 0730-0301, 1557-7368. doi: 10.1145/2766942. URL https://dl.acm.org/doi/10.1145/2766942.

Appendix A

Additional Fracturing Images

This chapter contains additional fracturing image grids for different meshes, similar to Figure 5.10, 5.11 and 5.12 in chapter 5.

A.1 Chair

Figure A.1 shows the distance field prediction of DeltaConv and MLP for the chair mesh.

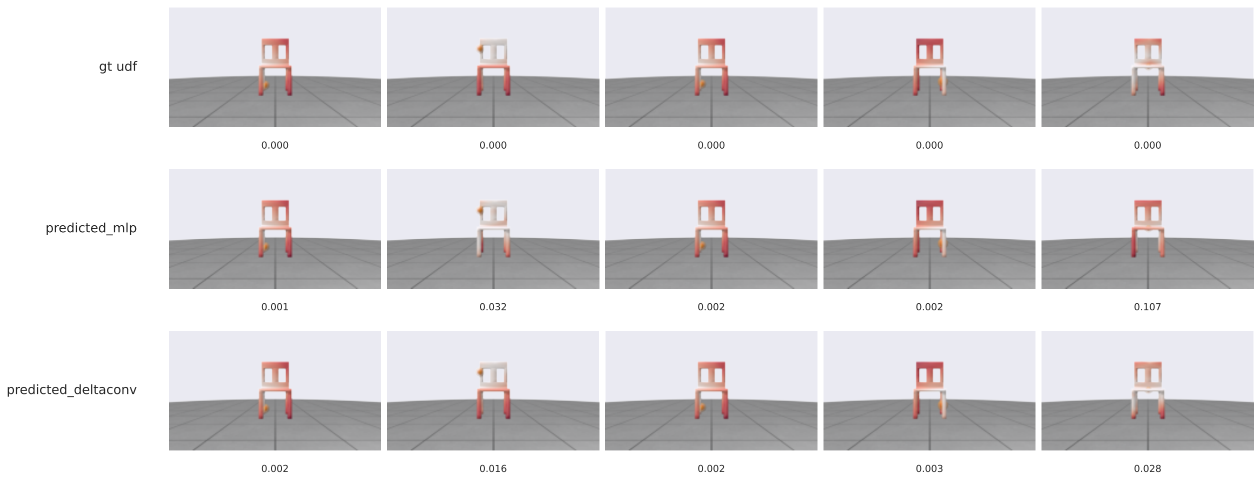


Figure A.1: UDF prediction comparison between DeltaConv and MLP. Values below each example show average per-vertex AMAE. White indicates vertices near the fracture boundary (low UDF), red indicates vertices farther away (high UDF).

Model	AMAE	MSE	Seg. Method	Duration (s)	RC	BC	ARI	AMI
DeltaConv	0.0174	0.0195	Felzenszwalb	10.37	0.207	0.263	0.245	0.424
			Hierarchical	8.15	0.205	0.074	0.705	0.621
			K-Means	6.51	0.136	0.495	0.137	0.242
			Region Growing	24.45	0.126	0.110	0.713	0.702
			Watershed	3.14	0.106	0.127	0.609	0.650
MLP	0.0169	0.0210	Felzenszwalb	20.82	0.209	0.272	0.250	0.429
			Hierarchical	7.85	0.214	0.076	0.731	0.659
			K-Means	5.62	0.136	0.490	0.134	0.245
			Region Growing	22.69	0.128	0.120	0.715	0.705
			Watershed	2.85	0.079	0.081	0.720	0.719

Table A.1: Performance comparison of DeltaConv and MLP models across segmentation methods. Duration in seconds, best-performing values can be highlighted as needed.

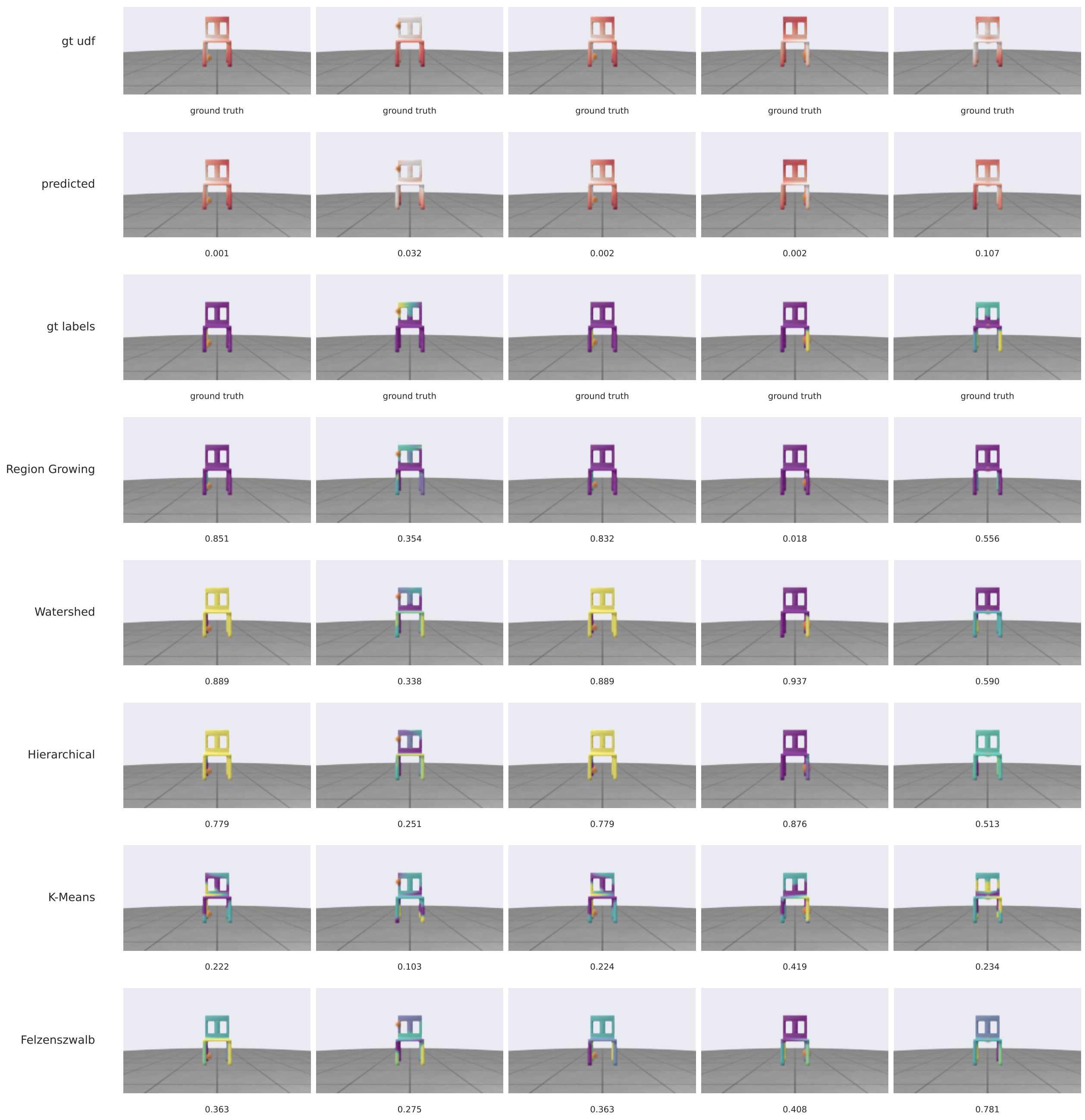


Figure A.2: Segmentation results with the MLP. Rows (top to bottom): ground-truth unsigned distance field (UDF), predicted UDF, ground-truth segmentation, and results from region growing, watershed, hierarchical, k-means, and Felzenszwalb. Columns: dataset samples. In UDFs, white = near fracture, red = far. Segmentation colors are arbitrary and only distinguish regions. Numbers indicate Rand index for segmentations (higher is better) and AMAE for UDFs (lower is better). Red dots mark impact points.

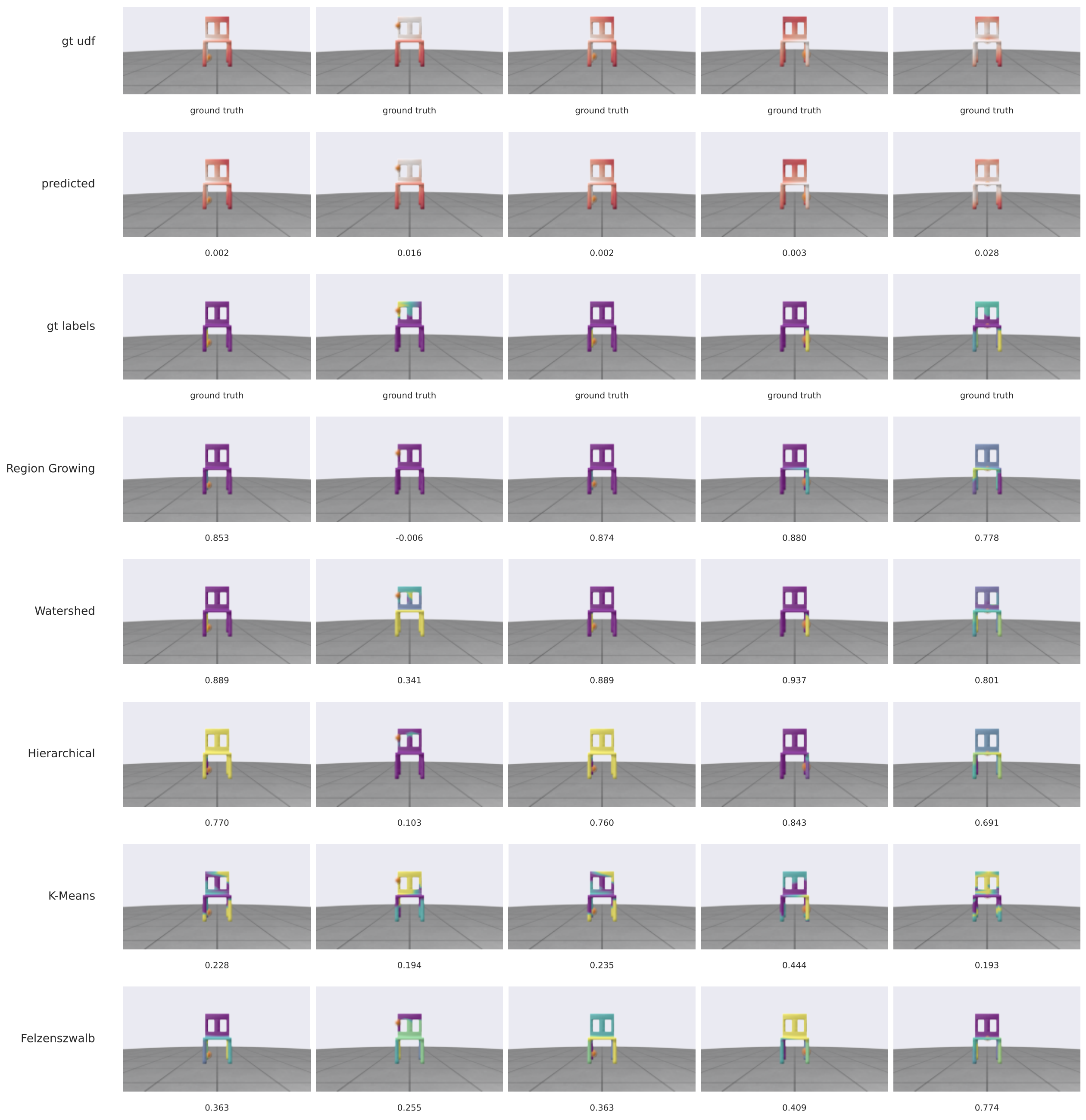


Figure A.3: Segmentation results with DeltaConv. Rows (top to bottom): ground-truth unsigned distance field (UDF), predicted UDF, ground-truth segmentation, and results from region growing, watershed, hierarchical, k-means, and Felzenszwalb. Columns: dataset samples. In UDFs, white = near fracture, red = far. Segmentation colors are arbitrary and only distinguish regions. Numbers indicate Rand index for segmentations (higher is better) and AMAE for UDFs (lower is better). Red dots mark impact points.

A.2 Vase

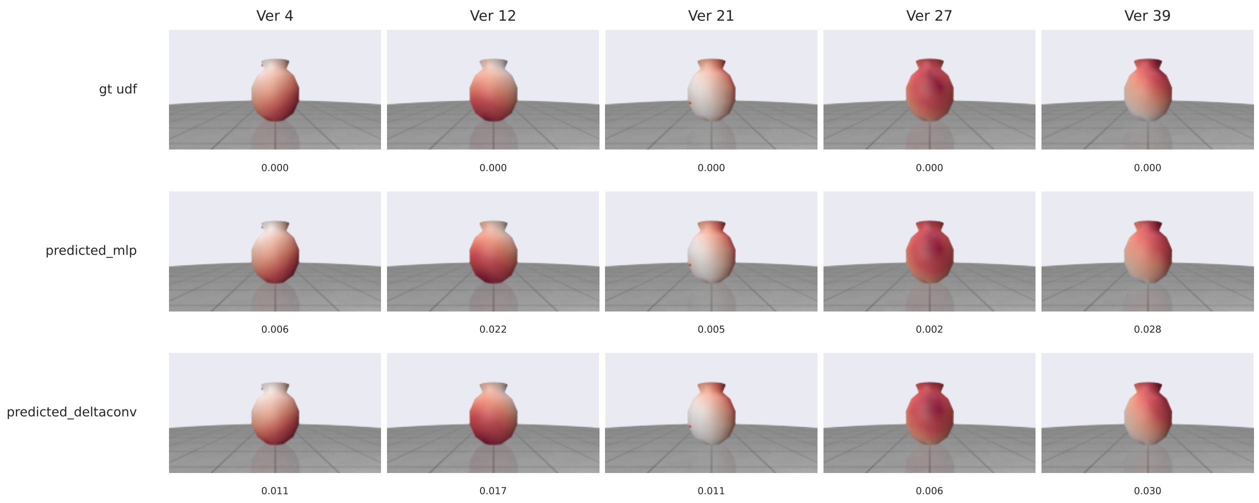


Figure A.4: UDF prediction comparison between DeltaConv and MLP. Values below each example show average per-vertex AMAE. White indicates vertices near the fracture boundary (low UDF), red indicates vertices farther away (high UDF).

Model	AMAE	MSE	Seg. Method	Duration (s)	RC	BC	ARI	AMI
DeltaConv	0.0122	0.0081	Felzenszwalb	6.63	0.029	0.289	-0.007	0.004
	0.0117	0.0062	Hierarchical	3.02	0.272	0.068	0.457	0.379
	0.0117	0.0062	K-Means	3.46	0.213	0.641	0.003	0.042
	0.0120	0.0066	Region Growing	12.38	0.205	0.148	0.228	0.136
	0.0109	0.0040	Watershed	1.85	0.060	0.091	0.508	0.384
MLP	0.0097	0.0063	Felzenszwalb	7.60	0.034	0.313	-0.001	-0.000
	0.0093	0.0050	Hierarchical	2.88	0.257	0.068	0.435	0.366
	0.0093	0.0050	K-Means	4.19	0.218	0.637	0.005	0.043
	0.0094	0.0058	Region Growing	11.68	0.168	0.118	0.282	0.175
	0.0092	0.0032	Watershed	1.71	0.087	0.118	0.297	0.153

Table A.2: Performance comparison of DeltaConv and MLP models across segmentation methods using values from the screenshot. Duration in seconds.

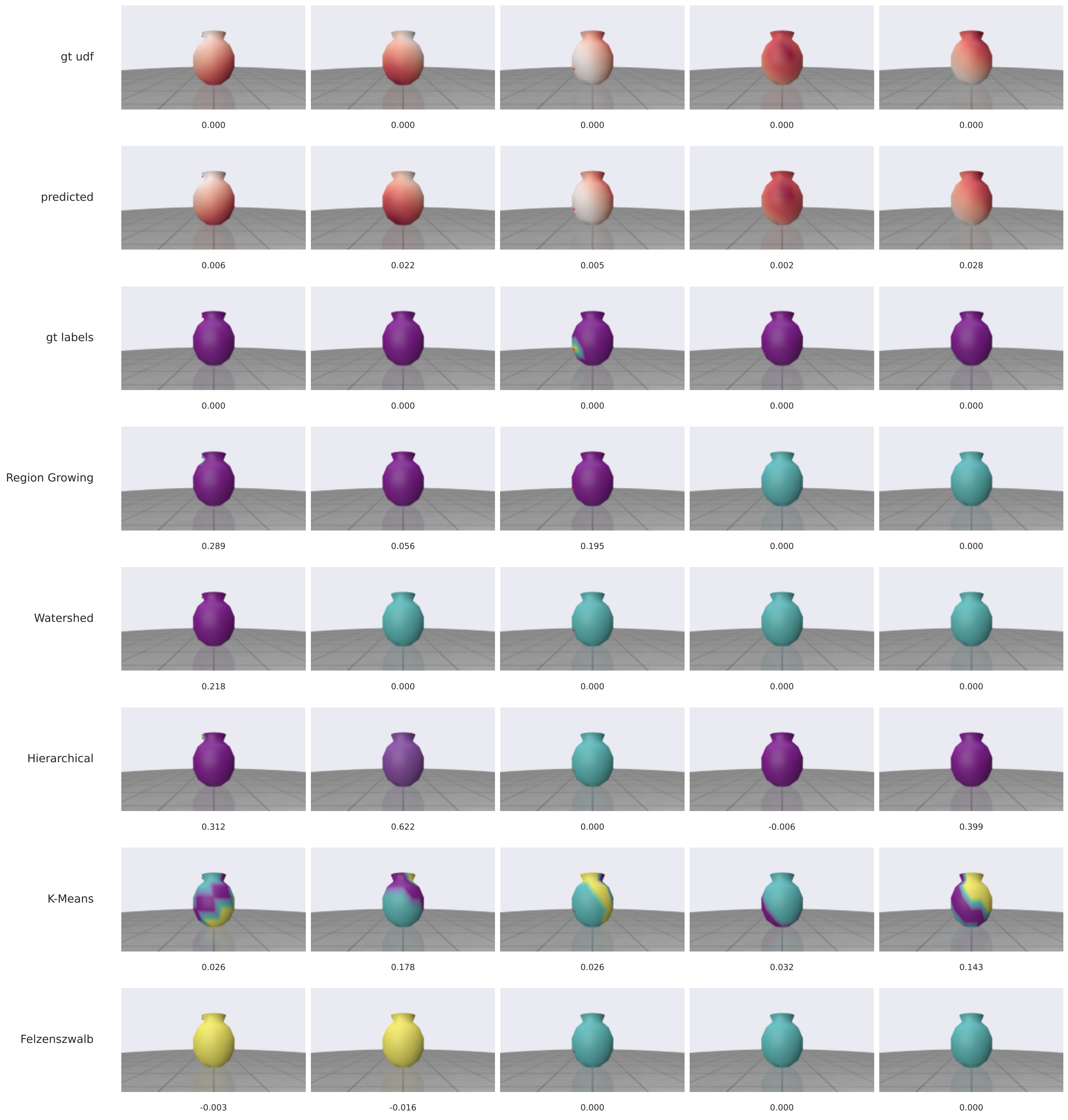


Figure A.5: Segmentation results with the MLP. Rows (top to bottom): ground-truth unsigned distance field (UDF), predicted UDF, ground-truth segmentation, and results from region growing, watershed, hierarchical, k-means, and Felzenszwalb. Columns: dataset samples. In UDFs, white = near fracture, red = far. Segmentation colors are arbitrary and only distinguish regions. Numbers indicate Rand index for segmentations (higher is better) and AMAE for UDFs (lower is better). Red dots mark impact points.

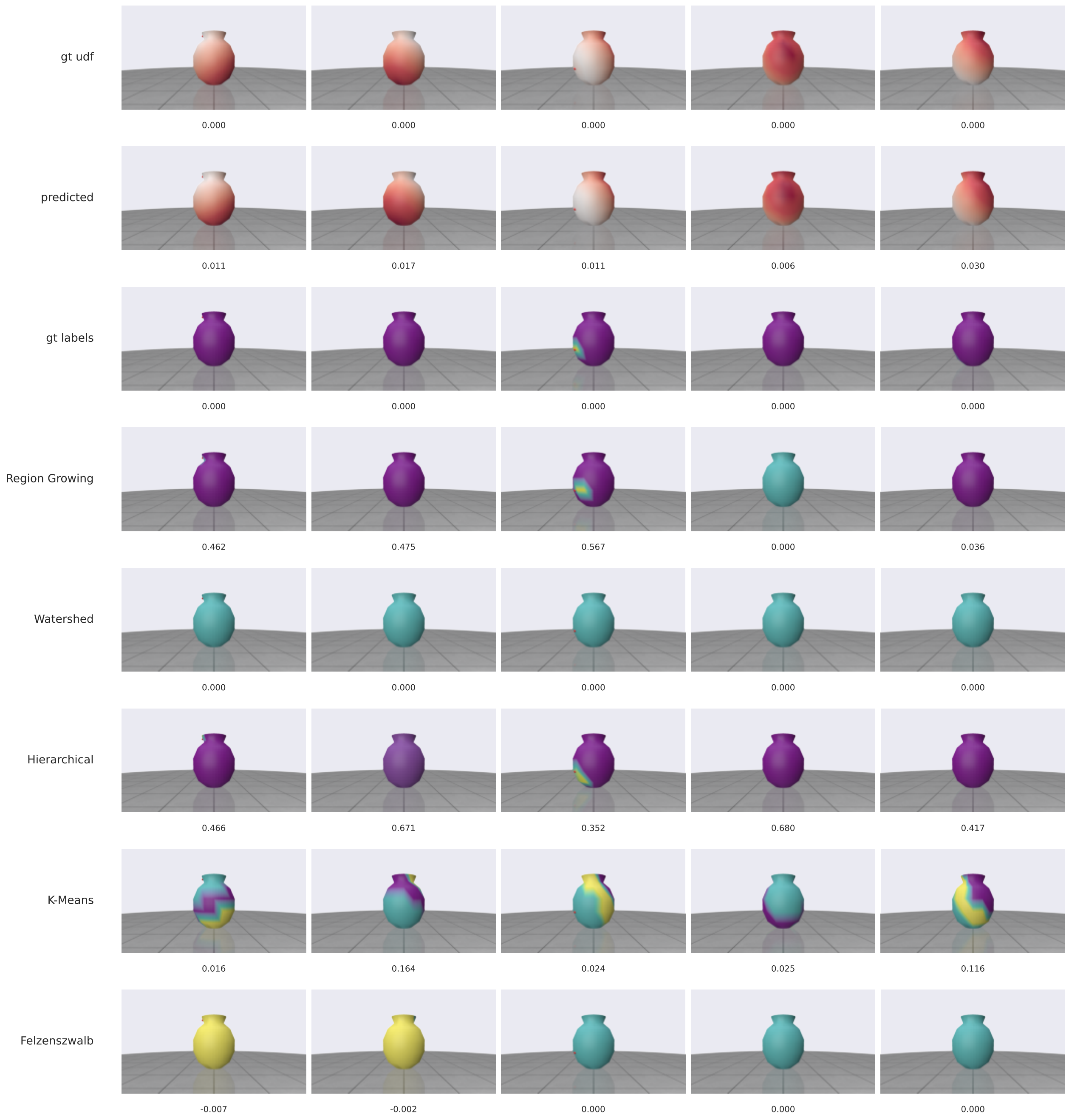


Figure A.6: Segmentation results with DeltaConv. Rows (top to bottom): ground-truth unsigned distance field (UDF), predicted UDF, ground-truth segmentation, and results from region growing, watershed, hierarchical, k-means, and Felzenszwalb. Columns: dataset samples. In UDFs, white = near fracture, red = far. Segmentation colors are arbitrary and only distinguish regions. Numbers indicate Rand index for segmentations (higher is better) and AMAE for UDFs (lower is better). Red dots mark impact points.