Politecnico di Torino

Corso di Laurea Magistrale in Ingegneria Meccanica
A.a. 2025/2026
Sessione di Laurea Febbraio/Marzo 2026

Politecnico di Torino

Graph Neural Network-Based Prediction of the Effective Stiffness of Hierarchical Spinodoid Mechanical Metamaterials

Relatori:
Prof. Andrea Tridello
Prof. Chao Gao
Prof. Chiara Bertolin

Candidati:
Carlo Alberto Greco

Abstract

Spinodoid mechanical metamaterial has received increasing interests since last decade because of its tunable stochastic design and defect insensitive [1, 2].
However, the mechanical response of spinodoid mechanical metamaterials is strongly correlated with generated stochastic designs, which might offer very poor mechanical performance and very large variation of mechanical properties (e.g., stiffness and anisotropy), thereby limiting wider applications in industry.
To tackle this designing challenge, we exploit the concept of hierarchical design, which has been proved a successful solution in Nature to generative tunable enhanced mechanical behaviors of materials, to improve the mechanical response of spinodoid mechanical metamaterials. Moreover, to effectively explore the huge design space offered by hierarchical spinodoid mechanical metamaterials, artificial intelligence (AI) based approach will also be exploited.
In this work, the hierarchical design principle of spinodoid mechanical metamaterials will be investigated first and the geometrical characterization of 2D and 3D hierarchical spinodoid mechanical metamaterials will be performed. Finite element (FE) simulations will be conducted by commercial FE software Abaqus to obtain the effective stiffness of designed hierarchical spinodoid mechanical metamaterials. The results of FE simulations will form the database for AI-based approach. Graph neural network, which offers great potential in the design of materials, will be constructed as the predictor of effective stiffness of hierarchical spinodoid mechanical metamaterials.
Therefore, this work can form the solid foundation of applying hierarchical spinodoid mechanical metamaterials for wider industrial applications in future.

Sammendrag

Spinodoide mekaniske metamaterialer har fått økende oppmerksomhet det siste tiåret på grunn av sitt justerbare stokastiske design og sin defekt-toleranse [1], [2]. Den mekaniske responsen til spinodoide mekaniske metamaterialer er imidlertid sterkt korrelert med de genererte stokastiske designene, som kan gi svært dårlig mekanisk ytelse og stor variasjon i mekaniske egenskaper (f.eks. stivhet og anisotropi). Dette begrenser deres anvendelse i industrien. For å møte denne designutfordringen benytter vi konseptet med hierarkisk design, som har vist seg å være en vellykket løsning i naturen for å generere justerbare og forbedrede mekaniske egenskaper i materialer, med mål om å forbedre den mekaniske responsen til spinodoide mekaniske metamaterialer.
Videre, for å utforske det enorme designrommet som tilbys av hierarkiske spinodoide mekaniske metamaterialer, vil en tilnærming basert på kunstig intelligens (KI) også bli benyttet. I dette arbeidet vil prinsippet for hierarkisk design av spinodoide mekaniske metamaterialer først bli undersøkt, og den geometriske karakteriseringen av 2D- og 3D-hierarkiske spinodoide mekaniske metamaterialer vil bli utført. Finite element (FE) simuleringer vil bli gjennomført ved hjelp av det kommersielle FE-programmet Abaqus for å bestemme den effektive stivheten til de designede hierarkiske spinodoide mekaniske metamaterialene. Resultatene fra FE-simuleringene vil danne databasen for den KI-baserte tilnærmingen. Et grafnevralt nettverk, som har stort potensial innen materialdesign, vil bli konstruert som en prediktor for den effektive stivheten til hierarkiske spinodoide mekaniske metamaterialer.
Dermed kan dette arbeidet danne et solid grunnlag for fremtidig industriell anvendelse av hierarkiske spinodoide mekaniske metamaterialer.

Preface

This master's thesis was carried out at the Norwegian University of Science and Technology (NTNU), within the Department of Mechanical and Industrial Engineering, as part of the Master's Degree Program in Mechanical Engineering at the Politecnico di Torino. The work was conducted between March and October 2025, under the supervision of Prof. Chao Gao and Prof. Chiara Bertolin, with Prof. Andrea Tridello serving as the academic advisor in Italy.

The research presented in this thesis focuses on the numerical generation, morphological characterization, and mechanical modeling of hierarchical spinodoid metamaterials. The motivation behind this project stems from the increasing scientific and industrial interest in architected materials, and from the need for efficient computational frameworks capable of linking complex microstructural morphologies to their macroscopic mechanical behavior. The work involved the development of a fully automated computational workflow integrating stochastic structure generation, finite element analysis (FEM), and graph-based deep learning. A Message Passing Neural Network (MPNN) was implemented in MATLAB to predict the effective stiffness of hierarchical spinodoid structures directly from their geometrical and topological descriptors. This interdisciplinary approach combined numerical modeling, programming, and artificial intelligence, contributing to the advancement of data-driven methods in materials engineering.

This thesis was carried out in parallel with the work of my colleague and friend Denis Giordana, who shared the same academic background at the Politecnico di Torino and worked under the same supervision team. Although each of us developed an independent thesis, focused respectively on spinodoid and Voronoi structures, the research followed a common workflow and methodological framework. Working alongside Denis made the overall experience particularly stimulating, as it provided opportunities for continuous discussion, exchange of ideas, and mutual support throughout the project.

Working on this thesis has been a deeply enriching experience that allowed me to strengthen my technical and analytical skills while gaining insight into advanced computational design strategies. It has been both challenging and rewarding, requiring patience, persistence, and creativity, and it provided a valuable opportunity to explore the integration of mechanics, numerical modeling, and artificial intelligence in the study of metamaterials.

I would like to express my sincere gratitude to Prof. Chao Gao and Prof. Chiara Bertolin for their continuous guidance, constructive feedback, and inspiring supervision throughout this project. I am also sincerely thankful to Prof. Andrea Tridello for his valuable support and coordination with the Politecnico di Torino.

Table of Contents

List of Figures

1. INTRODUCTION

In recent decades, the rapid development of additive manufacturing technologies and computational design methods has led to the emergence of metamaterials, a class of artificial materials whose macroscopic properties primarily arise from their internal microarchitecture rather than their chemical composition. This distinctive feature enables the realization of mechanical, thermal, and acoustic behaviors that are not found in conventional materials, such as negative elastic moduli, programmable directional stiffness, and highly controllable energy absorption capabilities. Mechanical metamaterials, in particular, have found applications across various fields, from biomedicine to aerospace, due to their ability to combine lightweight design, strength, and structural adaptability. The evolution of 3D printing technologies has further enabled the fabrication of complex and multiscale geometries, translating theoretical models once confined to numerical simulations into physically realizable structures.

Among the various families of metamaterials, spinodoid structures have recently attracted significant interest for their bicontinuous and non-periodic morphology, inspired by spinodal phase-separation processes described by the Cahn–Hilliard model. In this context, the spinodal decomposition is not interpreted as an actual physical phase separation between two materials but rather as a mathematical analogy for generating complex morphological fields.

The generation of spinodoid architectures can be approached mainly through two methodologies.

The first relies on the Cahn–Hilliard model, which describes phase separation in binary systems through the temporal evolution of a concentration field. The numerical solution of this equation produces realistic bicontinuous morphologies from which isosurfaces representing the two phases can be extracted. However, this process is computationally expensive and offers limited control over anisotropy.

To overcome these limitations, a second and more statistical approach based on Gaussian Random Fields (GRF) has been introduced. In this method, the microstructure is generated by constructing a random scalar field resulting from the superposition of a finite number of stationary waves with random amplitudes, phases, and propagation directions. When the propagation directions of these standing waves are uniformly distributed in space, an isotropic morphology is obtained. Conversely, by imposing an angular limit ϑ around the principal axes, the wave vectors become preferentially oriented, and the resulting structure exhibits anisotropic characteristics, such as lamellar, columnar, or cubic configurations. Applying a level-set threshold to the scalar field $\varphi(x)$ yields the bicontinuous surface of the structure corresponding to a prescribed relative density. This method, significantly more efficient than the Cahn–Hilliard simulation, provides direct control over the statistical properties and morphological anisotropy of the structure, making it particularly suitable for parametric studies and inverse design [3, 4, 5].

This bicontinuous configuration ensures a uniform stress distribution, high structural efficiency, and remarkable robustness to manufacturing defects, making spinodoids an ideal platform for the design of advanced mechanical metamaterials.

Kumar et al. [3] formally introduced the concept of spinodoid metamaterials, proposing a parametric approach that allows the internal morphology to be controlled through a small

number of parameters related to the relative density and the spatial correlation of the phase field. Their work demonstrated that spinodoids can outperform lattice structures based on periodic minimal surfaces, such as Gyroid or Primitive, in terms of specific stiffness and mechanical stability.

Liu et al. [5] performed an in-depth analysis of the mechanical behavior of both homogeneous and density-graded spinodoid structures, revealing pronounced post-yield stability and the absence of localized collapse. Experimental results showed a gradual transition toward densification, ensuring excellent energy absorption capacity and a more stable mechanical response compared to truss or TPMS lattices.

Similarly, Vafaeefar et al. [6] confirmed the high energy-absorption efficiency of spinodoids, demonstrating that their bicontinuous architecture promotes energy dissipation under dynamic loading conditions.

Zheng et al. [7] proposed an innovative data-driven topology optimization framework for designing spinodoids with target mechanical properties, combining high-fidelity FEM simulations with machine-learning models to drastically reduce computational cost. In parallel, Wang et al. [4] implemented an inverse-design approach based on artificial neural networks for the design of spinodoid bone scaffolds, tuning the anisotropic elastic response to mimic that of human trabecular tissue.

Similar results were obtained by Liu and Acar [8] using Generative Adversarial Networks (GANs), which were capable of synthesizing two-dimensional spinodoid configurations with pre-defined elastic properties.

Deng et al. [9] further extended this perspective by demonstrating how artificial-intelligence models can guide the design of three-dimensional non-periodic architectures with controllable directional properties.

In recent years, research has increasingly focused on the introduction of multiscale hierarchy in metamaterials, inspired by natural systems such as bone and mollusk shells. The integration of multiple structural levels enables the combination of stiffness, toughness, and energy-dissipation capability. Gu et al. [10] demonstrated that adding a secondary hierarchical level to the lamellar structure of the *Strombus gigas* mollusk shell increases impact resistance by up to 85% compared to single-scale configurations, due to crack-deflection and crack-arrest mechanisms. Zhu et al. [11] transferred this concept to artificial metamaterials by designing bio-inspired hierarchical architectures capable of enhancing energy absorption through the presence of substructures organized across multiple scales.

Within the broader context of graph-based metamaterials, the topological representation of architecture through graph models and the use of Graph Neural Networks (GNN) have enabled accurate structure–property correlations across large datasets of periodic and non-periodic lattices. Meyer et al. [12] demonstrated that GNNs can learn with high accuracy the relationship between structural configuration and mechanical response, achieving drastic reductions in computational time compared to FEM simulations.

Maurizi et al. [13] further extended this paradigm to mesh-to-graph domains, developing a Message Passing Neural Network (MPNN) applied to beam-based and shell-based metamaterials. Trained on high-fidelity FEM data, the model successfully predicted stress and strain fields within the structures with accuracy comparable to full numerical simulations, yet with computational costs several orders of magnitude lower. This work highlighted the potential of GNNs as physically consistent surrogate models for the mechanical analysis of complex, non-periodic metamaterials.

The application of deep learning to metamaterial design has thus transformed the traditional paradigm, from iterative numerical simulations to data-driven predictive modeling.

In this context, GNNs, and in particular MPNNs, represent highly effective tools for learning nonlinear relationships between geometry and mechanical response. These models are trained on FEM-based datasets, where each graph represents a specific architecture, and the nodes and edges encode its geometric and topological features. Through the message-passing process, information is propagated and aggregated among connected nodes, allowing the network to learn complex structural patterns. Once trained, the network acts as a computational surrogate capable of rapidly predicting mechanical properties of unseen architectures, drastically reducing the need for simulations and enabling efficient optimization and inverse-design workflows [12, 13, 4].
Although the state of the art for primary spinodoids is now well established, from Cahn–Hilliard/GRF generation and mechanical characterization to machine-learning-guided inverse design, significant gaps remain concerning hierarchical spinodoids. In particular, systematic morphological and mechanical characterization of multiscale architectures is still limited, while the modeling and prediction of effective stiffness using graph-based neural networks remain largely unexplored. From a methodological standpoint, the complementarity between neural and meta-heuristic approaches has proven crucial: deep neural networks act as differentiable surrogates that accelerate the exploration of the design space, while genetic and Bayesian optimization algorithms provide global search strategies for complex phenomena such as buckling and targeted anisotropy [14, 4].
In this framework, the present thesis aims to fill the existing gaps on hierarchical spinodoids through the development of an integrated pipeline that combines generation, simulation, and machine learning.
The work includes:

- the definition of a multiscale generation and morphological-characterization method for hierarchical architectures, developed for both 3D (Section 2.2.1 – Generation and characterization of 3D structures) and 2D cases (Section 2.2.3 – Transition to the 2D case, Section 2.2.4 – Hierarchical 2D lamellar spinodoids);

- the construction of a FEM-based dataset for evaluating effective stiffness (Section 2.2.2 – FEM analysis, Section 2.2.4.7 – FEM analysis, Section 2.2.4.9 – Dataset definition for neural network development);

- the implementation of a GNN/MPNN model capable of learning structure–property relationships on multi-resolution graphs (Section 2.2.5 – Development of the Message Passing Neural Network).

In doing so, this study systematically extends the consolidated advances achieved on primary spinodoids [7, 3, 5], on deep-learning strategies for metamaterials [15], and on graph-centric GNN approaches [12, 13], transferring them to the hierarchical domain, where the current state of the art remains in its early stages of development.

2. MATERIALS AND METHODS

2.1 MATERIALS

2.1.1 Theoretical Background on spinodoid generation

Spinodoid metamaterials represent a class of metamaterials whose macrostructure is inspired by the characteristic morphology of microstructures generated by spinodal decomposition. The latter is a phase separation process that occurs spontaneously under thermodynamic instability and is driven by concentration gradients. The result is the formation of two distinct, continuously distributed and interconnected phases, separated by an interface. If, through a selective or hypothetical treatment, only one of the two phases is retained, the resulting structure is referred to as a spinodoid.
The phase separation process can be simulated using the Cahn-Hilliard equations, but this is computationally demanding.
A simpler and more effective way to generate a spinodoid (or spinodal-like) structure is by creating a GRF (Gaussian Random Field), a scalar field formed by the superposition of an arbitrary number of static waves with the same wavelength but different propagation directions and phases. This is clearly a purely analytical method, not based on a physical process, but it enables the generation of the desired structure by approximating the phase field description resulting from spinodal decomposition.
The scalar field $\varphi(x)$ is therefore defined as:

$$\varphi(x) = \sqrt{\frac{2}{N}} \sum_{i=1}^{N} \cos(\beta \mathrm{n_i} \cdot \mathrm{x} + \gamma_\mathrm{i})$$

where:

- N is the number of superimposed static waves;
- β is the wave number, defined as $\frac{2\pi}{\lambda}$ where λ is the common wavelength;
- $\mathrm{n_i}$ is the unit propagation vector of the i-th wave in an n-dimensional space;
- x is the position vector in the n-dimensional space at which the scalar field is evaluated;
- γ_i is the phase of the i-th wave.

The propagation unit vectors of the waves are initially chosen randomly throughout the considered n-dimensional space. However, this formulation does not allow for the creation of anisotropic structures due to the lack of control over the propagation directions, which are selected at random.

It is, however, possible to limit and control these directions by constraining them within specific "limiting cones" that define the maximum angle the propagation vector can form with respect to the unit vectors of the coordinate axes (Figure 2.2.4: Limiting cones)
Taking three-dimensional space as an illustrative example, each unit vector $\mathrm{n_i}$ must satisfy the condition:

$$\mathrm{n_i} \sim \cup \left(\{k \in S^2 : \ (|k \cdot \hat{e}_1| > \cos\vartheta_1) \vee (|k \cdot \hat{e}_2| > \cos\vartheta_2) \vee (|k \cdot \hat{e}_3| > \cos\vartheta_3)\}\right)$$

Where $S^2 = \{k \in \mathbb{R}^3 : \|k\| = 1\}$ denotes the set of unit vectors defined by a three-dimensional sphere of unit radius, while $\{\hat{e}_1; \hat{e}_2; \hat{e}_3\}$ are the unit vectors along the three coordinate axes x, y, z. The angles $\{\vartheta_1; \vartheta_2; \vartheta_3\}$ are the limiting angles that the propagation direction vectors of the waves can form with each of the three coordinate axes.
The three conditions are linked by a logical OR variable, since in some cases, depending on the values taken by the angles ϑ, it is not possible for a unit vector to satisfy all three conditions simultaneously. The closer the limiting angles approach $\pi/2$, the higher the probability that randomly selected unit vectors will satisfy one of the three conditions.
The constraint imposed on the wave propagation directions is probabilistic in nature; therefore, the final outcome depends not so much on the specific values of the angles ϑ individually, but rather on the combination generated by them. For example, the choice of values $\vartheta_1 = 10°$, $\vartheta_2 = 5°$, $\vartheta_3 = 90°$ will not result in controlled anisotropy along the x and y axes, because any unit vector will satisfy the third condition, creating an isotropic structure comparable to a combination such as $\vartheta_1 = 90°$, $\vartheta_2 = 90°$, $\vartheta_3 = 90°$.
Once the scalar field has been generated within a domain Ω, a level set φ_0 must be applied to obtain the spinodoid structure.
This will produce a binary 0/1 domain corresponding to a physical structure of voids and solids.

$$\chi(\mathrm{x}) = \begin{cases} 1 \; if \; \varphi(x) \leq \varphi_0 \\ 0 \; if \; \varphi(x) > \varphi_0 \end{cases}$$

The value of φ_0 is computed as the quantile of the normalized variable $\varphi(x)$ corresponding to the desired relative density ρ, that is:

$$\varphi_0 = \sqrt{2}\, erf^{-1}(2\rho - 1)$$

In order to use this threshold formulation, the GRF must also be normalized, i.e. transformed to have zero mean and unit standard deviations as follows:

$$\varphi'(x) = \frac{\varphi(x) - \mu}{\sigma}$$

Where μ and σ are the mean and standard deviation, respectively, of the GRF $\varphi(x)$ values.

2.1.2 Theoretical Background on deep learning and MPNNs

2.1.2.1 Artificial Intelligence and Machine Learning

Artificial Intelligence (AI) encompasses a set of methodologies and tools that enable computer systems to perform tasks typically associated with human intelligence. Among its subfields, Machine Learning (ML) has gained a central role, as it provides algorithms capable of automatically learning from data by identifying patterns, correlations, and complex relationships without the need for explicit programming for each specific task.
The fundamental goal of ML is to construct mathematical models $f_\theta: X \to Y$, parameterized by θ, that can accurately predict an output Y from a given input X.
Here:

- f_θ represents the ML model, i.e., the function mapping each input datum X to an output Y;
- X denotes the input space, representing the data provided to the model (e.g., images, feature vectors, node coordinates);
- Y represents the output space, i.e., the quantity to be predicted (e.g., image class, structural property);

The parameters θ are what the model learns during training, such as weights and biases in neural networks.
The training process aims to find the optimal values of θ that minimize a loss function $L(f_\theta(X), Y)$, ensuring that the model's predictions are as close as possible to the real data.
Classical ML models include *linear and logistic regression*, *decision trees*, *Support Vector Machines (SVMs)*, and ensemble methods such as *Random Forests* and *Gradient Boosting*. While effective in many contexts, these methods often rely heavily on manually engineered features. In practice, this means that relevant characteristics must be precomputed or selected from raw data to adequately represent the problem. For example, in an image dataset, one might compute color or texture histograms, whereas in a structural context, geometric quantities such as lengths, angles, or ratios may be needed.
This manual feature extraction requires substantial domain knowledge, that is, understanding which characteristics are meaningful for the studied phenomenon. Without such insight, the model may fail to capture the essential relationships between inputs and outputs.

2.1.2.2 Deep Learning

To overcome these limitations, Deep Learning (DL) has emerged as a powerful subfield of ML. It is characterized by the use of deep artificial neural networks, composed of multiple layers of nonlinear transformations. Such networks are capable of learning hierarchical data representations, where lower layers extract simple features, while deeper layers combine them into increasingly abstract and complex representations.
This property allows deep networks to automate much of the feature extraction process, reducing the need for manual feature engineering. Classical architectures such as Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) have

demonstrated high effectiveness in tasks involving computer vision, speech recognition, and temporal sequence modeling.
Most data traditionally processed in ML have a regular structure, also known as Euclidean data. Euclidean data can be represented in an ordered Cartesian space, for example:

- images (2D grids of pixels),
- videos (sequences of images),
- temporal data (ordered sequences of values).

For such data, traditional networks can easily apply convolutions (which aggregate information from neighboring pixels or features) or sequential operations (which process ordered time steps).

2.1.2.3 Graph Neural Networks and Message Passing Neural Networks

Conversely, many real-world phenomena produce non-Euclidean data, i.e., data lacking a regular or ordered structure. These are better represented by graphs, consisting of nodes (entities) and edges (relations between entities). Graphs can have arbitrary topology: nodes may have varying numbers of neighbors, and there is no intrinsic ordering among them.
Applying traditional networks to such irregular data is problematic because operations such as standard convolutions or sequential updates require a fixed, ordered structure. On a graph, there are no fixed "neighbor positions" or inherent sequences to process.
Examples of non-Euclidean data include social networks, chemical molecules, transportation networks, engineered structures, and biological neural networks. For these cases, Graph Neural Networks (GNNs) were developed, models capable of learning node-, edge-, or graph-level representations while preserving topological and relational information.
GNNs extend the concept of deep learning to graphs by allowing each node v to update its representation by aggregating information from its neighboring nodes $N(v)$. Each node v is associated with a feature vector or embedding h_v, representing its state, and each edge (u, v) may have attributes e_{uv} describing the relation between nodes.
The network updates node embeddings iteratively through message passing, using the following general formulation:

$$m_v^{(t+1)} = \sum_{u \in N(v)} M_t \left(h_v^{(t)}, h_u^{(t)}, e_{uv} \right),$$

$$h_v^{(t+1)} = U_t(h_v^{(t)}, m_v^{(t+1)}),$$

where M_t and U_t are learnable functions, typically implemented as Multi-Layer Perceptrons (MLPs).
An MLP is a feedforward network composed of multiple layers of artificial neurons, where each layer transforms the input through weights and nonlinear activation functions.
In MPNNs, the weights of these MLPs are shared across all nodes, meaning that the same function with identical parameters is applied to each node. This design ensures consistency, reduces the number of parameters, and allows the model to generalize across graphs with different sizes and topologies.

After T message passing iterations, the final node embeddings are aggregated through a readout function R to obtain a global graph representation:

$$h_G = R(\{h_v^{(T)} \mid v \in G\}),$$

where R can be a sum, mean, max operation, or an attention-based mechanism.
The resulting graph-level embedding can then be used for regression or classification tasks.
MPNNs offer several key advantages:

- Permutation invariance: the model's output does not depend on the order of the nodes, ensuring robustness to node reordering;
- Local and global information propagation: achieved through the iterative message passing mechanism;
- Flexibility: capable of incorporating node and edge attributes, making them suitable for heterogeneous data;
- Generalizability: due to shared weights, the same network can be applied to graphs of varying topologies and sizes.

Section 2.2.5 – Development of the Message Passing Neural Network of this work presents the development of an MPNN designed to learn the relationships between input features of hierarchical spinodoid structures and their uniaxial compressive stiffness. The goal is to predict the effective stiffness directly from structural features, eliminating the need for Finite Element Method (FEM) simulations, within a controllable approximation error.

2.1.3 Code implementation: Matlab

In this thesis work, the MATLAB environment was employed throughout all the main phases of the study.
Specifically, it was used for the generation and characterization of spinodoid structures during the initial stage of the research, and subsequently for the creation of the complete model to be subjected to Finite Element Method (FEM) analysis, through the automatic generation of an .inp file via dedicated MATLAB scripts.
Finally, MATLAB also served as the development environment for the implementation of the Message Passing Neural Network (MPNN).
Several plots, figures, and graphs presented in this thesis were likewise produced within the MATLAB environment.

2.1.4 FEM Analysis: Abaqus

The study involves the application of finite element analysis (FEA) to the generated spinodoid structures in order to evaluate their uniaxial compressive stiffness. The commercial software Abaqus will be employed for this purpose, initially through the graphical user interface and subsequently by directly generating the .inp input file using MATLAB. This approach will enable the automation of model generation and the creation of a dataset containing both the structures and their corresponding stiffness values, without requiring manual interaction with the Abaqus GUI for each case.

2.2 METHODS

Before describing the individual steps in detail, the overall workflow of this study is summarized in the figure below:

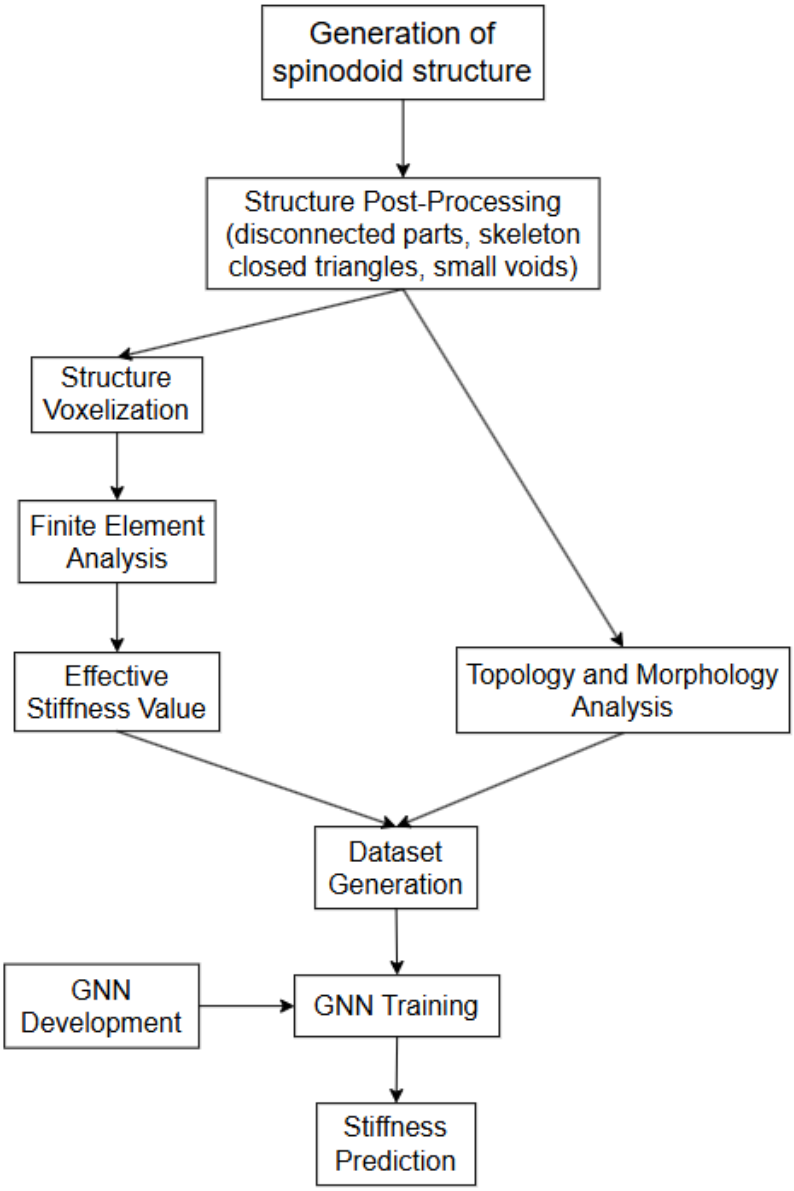


Figure 2.2.1: Overall workflow

2.2.1 Generation and characterization of 3D structures

2.2.1.1 Primary structure generation

The MATLAB code starts by generating the spinodoid structure according to the method described in Section 2.1.1 – Theoretical Background .
Initially, the function spinodoid from the GIBBON toolbox was used for this purpose.
This function requires as input a struct element containing the main generation parameters as well as additional functionalities (such as isocap to close the intersection surfaces at the domain boundaries, and visualization commands like eye(3)).

```
%% SPINODOID STRUCTURE GENERATION
inputStruct = struct();
inputStruct.isocap = true;              % Enables capping to close openings
inputStruct.domainSize = 1;             % Domain size
inputStruct.resolution = 70;            % Resolution for GRF sampling
inputStruct.waveNumber = 9.2 * pi;      % Wave number of the GRF
inputStruct.numWaves = 100;             % Number of waves in the GRF
inputStruct.relativeDensity = 0.3;      % Relative density (between 0.3 and 1)
inputStruct.thetas = [15 15 15];        % Cone angles (in degrees). If set to 90°,
                                          it allows maximum orientation
                                          freedom to the static waves
inputStruct.R = eye(3);                 % Rotation matrix (identity matrix, must
                                          be in SO(3))
inputStruct.ignoreChecks = false;       % Do not ignore parameter checks
% Call the spinodoid function
[F, V, C, GRF, X, Y, Z, levelset] = spinodoid(inputStruct);
% BINARY TRANSLATION WITH 3D GRID
binaryMatrix = GRF <= levelset;
```

It returns the following outputs:

- F: a vector containing the indices of the three vertices of each face in the surface triangulation of the structure;
- V: a vector of the coordinates of the triangulation vertices;
- GRF: a 3D matrix containing the values of the Gaussian random field at each grid point;
- X: a 3D matrix with the x-coordinate value of each grid point;
- Y: a 3D matrix with the y-coordinate value of each grid point;
- Z: a 3D matrix with the z-coordinate value of each grid point;
- Levelset: the level set value computed as described in Section 2.1.1 – Theoretical Background.

Finally, the structure is converted into a binary format for subsequent analyses.

The next step was the implementation of a script for generating the spinodoid structure in MATLAB without relying on auxiliary functions:

```
% 3D SPINODOID GENERATION
domain_size = 1;      % domain size
% linear resolution of the grid along each of the 3 coordinate axes
grid_res = 100;
% side length of the cubic voxel (needed later for voxelized structure generation)
voxelLength = domain_size / (grid_res - 1);
relative_density = 0.3;               % relative density of the structure
% x, y, z-coordinate vector of grid points
x_vec = linspace(0, domain_size, grid_res);
y_vec = linspace(0, domain_size, grid_res);
z_vec = linspace(0, domain_size, grid_res);
% create x, y, z coordinate matrices of the grid
[x_grid,y_grid,z_grid]=ndgrid(x_vec,y_vec,z_vec);
% number of superimposed static waves
n_waves = 100;
wave_lenght = domain_size/4.5;                          % wavelength
wavenumber = 2*pi/wave_lenght;                          % wavenumber
% limiting cones around x, y, z directions
theta1 = 90 * (pi / 180);
theta2 = 90 * (pi / 180);
theta3 = 90 * (pi / 180);

% Generating random waves
% initialize the matrix of wave propagation unit vectors
v = zeros(n_waves, 3);
% initialize the vector of wave phase shifts
phase_shift = zeros(n_waves, 1);
for i = 1:n_waves
    theta = 2*pi*rand;              % random theta angle in spherical coordinates
    phi = asin(2 * rand() - 1);     % random phi angle in spherical coordinates
% calculate 3D unit vector components and ensure at least one limiting condition
is satisfied
    while ~ (abs(cos(phi)*cos(theta))>=cos(theta1) ||
             abs(cos(phi)*sin(theta))>=cos(theta2) || abs(sin(phi))>=cos(theta3))
        theta = 2*pi*rand;
        phi = asin(2 * rand() - 1);
    end
    v(i, :) = [cos(phi)*cos(theta), cos(phi)*sin(theta), sin(phi)];
    phase_shift(i) = 2 * pi * rand;
end

% Compute the GRF field
GRF = zeros(grid_res, grid_res, grid_res);       % initialize the 3D GRF matrix
for i = 1:length(x_vec)
    for j = 1:length(y_vec)
        for k = 1:length(z_vec)
            for w = 1:n_waves
                GRF(i,j,k) = GRF(i,j,k) + cos(wavenumber * v(w, :) * [x_vec(i),
                             y_vec(j), z_vec(k)]' + phase_shift(w));
                             % compute value at each point
            end
        end
    end
end
GRF = GRF * sqrt(2/n_waves);
GRF = (GRF - mean(GRF(:))) / std(GRF(:));              % normalize the GRF
```

```
thresh = sqrt(2) * erfinv(2 * relative_density - 1);   % compute threshold value
domain_bin = GRF < thresh;                             % define the binary domain

% Visualization
figure;
xlabel('X'); ylabel('Y'); zlabel('Z');
surf = isosurface(x_vec, y_vec, z_vec, domain_bin, 0.5); % use the coordinate
                                                           vectors
s = patch(surf, 'FaceColor', [0 0.5 1], 'EdgeColor', 'none', 'FaceAlpha', 0.9);
isonormals(x_vec, y_vec, z_vec, domain_bin, s);
hold on;
cap_struct = isocaps(x_vec, y_vec, z_vec, domain_bin, 0.5,
'capstyle','projected');
patch(cap_struct, 'FaceColor', [0 0.5 1], 'EdgeColor', 'none', 'FaceLighting',
'gouraud');
axis([0 domain_size 0 domain_size 0 domain_size]);
axis equal;
view(3);
grid on;
camlight('headlight');
lighting phong;
material shiny;
title('3D Spinodoid');
```

The size of the cubic domain along each axis is set to 1 (domain_size).
An example of a structure that can be obtained is shown below:

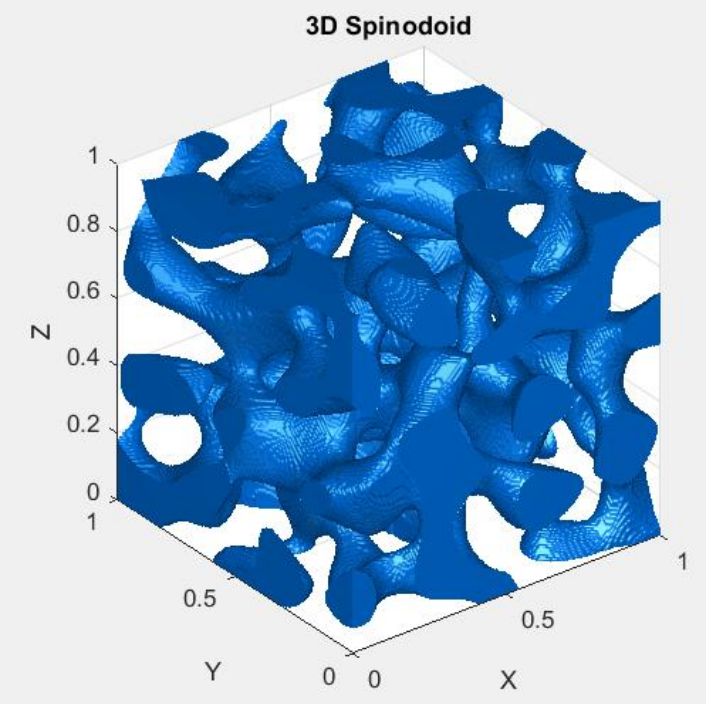


Figure 2.2.2: Example of generated structure

Further examples of achievable structures and the influence of the generation parameters on the structures obtained are presented in Section 3.1.1 – Effect of Generation Parameters on the Obtained Structures.

The unit vector of the propagation direction of each wave is calculated in spherical coordinates following the convention below:

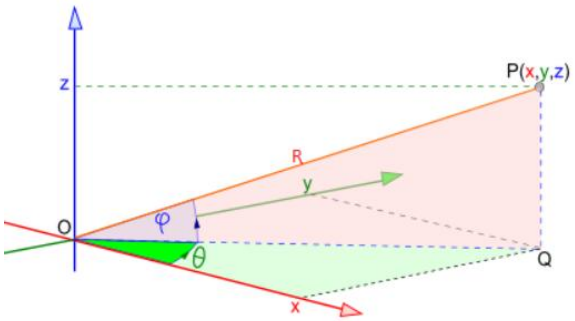


Figure 2.2.3: Convention used for spherical coordinates

Considering that the wave's unit vector has a magnitude of one, its three components are:

- Component along the x-axis: $v_x = \cos\varphi \cos\theta$;
- Component along the y-axis: $v_y = \cos\varphi \sin\theta$;
- Component along the z-axis: $v_z = \sin\varphi$.

The limiting cones extend in both directions along each coordinate axis, as illustrated in the figure below; therefore, to verify whether the unit vectors satisfy the boundary conditions, their corresponding absolute values are used in the script.

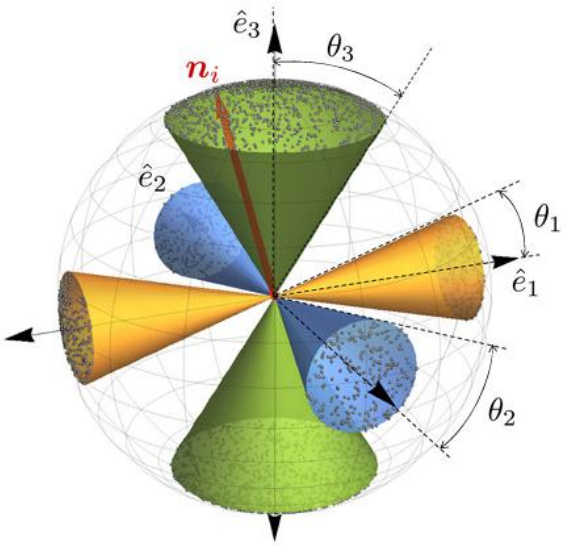


Figure 2.2.4: Limiting cones

Once the parameters of each static wave are obtained, the Gaussian random field (GRF) is calculated at every grid point by summing the contributions of each wave according to the formula already presented in paragraph 2.1.1 – Theoretical Background on spinodoid generation. Subsequently, the GRF is normalized and compared to the threshold variable to obtain the binary domain of the structure.

2.2.1.2 Removing detached parts

As shown in Figure 3.1.5: Structure before and after removing isolated components, the generated structure contains isolated parts, i.e., disconnected from the main body. These do not affect the relevant characteristics or mechanical properties of the spinodoid structure; therefore, lines of code are implemented to remove them:

```
%% keeping only the main structure without detached parts
Connectivity = bwconncomp(domain_bin, 6);              % 6-connectivity in 3D
% Compute number of points in each connected component
numPixels = cellfun(@numel, Connectivity.PixelIdxList);
% Find the index of the largest component
[~, idxLargest] = max(numPixels);
% Create a new binary matrix containing only the main component
domain_bin = false(size(domain_bin));
domain_bin(Connectivity.PixelIdxList{idxLargest}) = true;
```

For this purpose, the function *bwconncomp* is used; it takes two input arguments: the first is a 3D binary matrix, and the second is the type of connectivity required. In 3D, the default connectivity value is 26, meaning that two true-valued points are considered connected even if they are diagonally adjacent within an imaginary cube formed by the points in the 3D grid.
Since the structure will later be voxelized, to avoid cubic voxels being connected only along an edge, a connectivity of 6 is preferred, which considers connections only along the three principal coordinate axes.
The output of the *bwconncomp* function is a struct containing four fields:

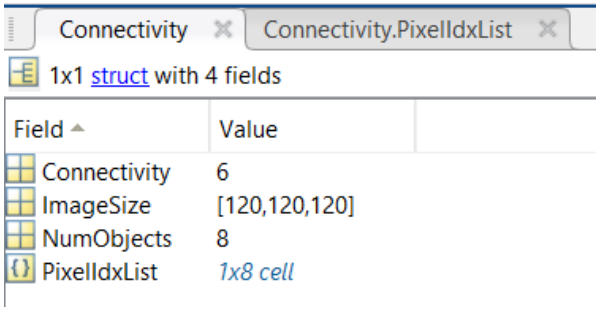


Figure 2.2.5: Output of bwconncomp() function

1. The requested connectivity
2. A vector containing the dimensions of the binary matrix (in this case, with a linear resolution of 120)
3. The number of independent components found throughout the entire domain
4. A cell array containing a vector for each component, composed of the linear indices of the points that make up that component, as shown in the figure below:

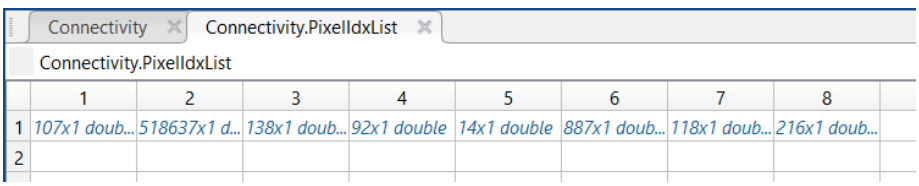


Figure 2.2.6: PixelIdxList variable

At this point, the function *cellfun* is used to obtain a vector containing the number of points comprising each "piece" of the structure.
The index of the largest component (in this case, the one with index 2) is identified, and the binary structure matrix is reconstructed by assigning a true value only to the points belonging to that component. The result is shown in Section 3.1.2 – Structure After Removal of Disconnected Parts.

2.2.1.3 Thinning algorithm (skeletal graph)

To perform a morphological characterization of the structure, the next step was to apply a 3D topological iterative thinning algorithm to obtain a skeletal graph.
This enables the subsequent analysis of properties such as the length of branching segments, node connectivity, and the orientation of various skeletal segments.

```
%% THINNING ALGORITHM
skel = Skeleton3D(domain_bin);  % Apply 3D skeletonization to the binary domain
[idxY, idxX, idxZ] = ind2sub(size(skel), find(skel == 1));  % Get indices of
                                                              skeleton points
skeletonpoints = zeros(length(idxX), 3);  % Initialize array for skeleton point
                                             coordinates
for i = 1:length(idxX)
    skeletonpoints(i, :) = [x_grid(idxX(i), idxY(i), idxZ(i)),
                            y_grid(idxX(i), idxY(i), idxZ(i)),
                            z_grid(idxX(i), idxY(i), idxZ(i))];
                            % Convert voxel indices to physical coordinates
end

% Build connections
delta = 1 / (grid_res - 1);  % Compute voxel spacing based on grid resolution
tol = delta * 0.1;           % Set tolerance for distance comparisons
maxDist = sqrt(3) * delta + tol;  % Maximum allowed distance between connected
                                     skeleton points (diagonal of cube + tol)
connections = [];  % Initialize list of connections
for i = 1:size(skeletonpoints,1)
    for j = i+1:size(skeletonpoints,1)
        if norm(skeletonpoints(i,:) - skeletonpoints(j,:)) <= maxDist
            connections = [connections; i, j];  % Add connection if distance is
                                                   within threshold
        end
    end
end

% Build initial graph
G = graph(connections(:,1), connections(:,2));  % Create undirected graph from
                                                   connections
% Plot original skeleton graph
figure;
plot(G, 'XData', skeletonpoints(:,1), 'YData', skeletonpoints(:,2), 'ZData',
skeletonpoints(:,3),'NodeLabel', {}, 'EdgeColor', 'k', 'LineWidth', 1,
'MarkerSize', 1);
title('Original 3D Skeleton Graph (with closed triangles)');
xlabel('X'); ylabel('Y'); zlabel('Z'); axis equal; grid on;
```

Initially, the function *Skeleton3D* is applied to the three-dimensional binary matrix (*domain_bin*), which describes the volumetric domain of the structure as a 3D grid of points. This matrix has a value of 1 at positions where material is present and 0 elsewhere. The function performs a volumetric reduction operation, called skeletonization or thinning, which transforms the three-dimensional shape into a network of connected points with unit thickness, identifying the geometric "backbone" of the original structure. The result is stored in a three-dimensional binary matrix (*skel*) with the same dimensions as the original domain, where points belonging to the skeleton have a value of 1.
Next, the three-dimensional indices of all voxels composing the skeleton are identified by using the function *find* to extract the linear indices of points with value 1, and then the function *ind2sub* to convert them into discrete coordinates along the three spatial dimensions. These indices are assigned to the variables *idxY*, *idxX*, and *idxZ*, representing the coordinates along the Y, X, and Z dimensions of the matrix, respectively. This coordinate order reflects MATLAB's matrix reading convention (row, column, depth), which corresponds to y, x, and z coordinates.
For each skeleton point, the discrete coordinates are converted into real physical spatial coordinates by using the matrices *x_grid*, *y_grid*, and *z_grid*. The obtained values are stored in the matrix *skeletonpoints*, where each row corresponds to the three-dimensional coordinates of a skeleton point.
To determine which skeleton points are connected to each other, the distance between adjacent points, denoted as *delta*, is first defined. This is calculated based on the grid resolution along each axis (*grid_res*), assuming the physical domain is normalized to a unit length. A numerical tolerance (*tol*), equal to 10% of *delta*, is then defined to compensate for possible floating-point errors. The maximum allowable distance to consider two skeleton points connected is set to *maxDist*, calculated as the length of the voxel cube diagonal (the square root of 3 multiplied by delta) increased by the tolerance.
To build the connectivity network, the code analyzes all possible pairs of points in the skeleton and calculates the Euclidean distance between each pair. If the distance between two points is less than or equal to the maximum defined value *maxDist*, they are considered connected, and their pair of indices is added to the connections list. In this way, the topology of the skeletal network is determined based on the spatial proximity of the points.
Finally, an undirected graph *G* is created using MATLAB's *Graph* function, which takes as input the pairs of connected nodes stored in *connections*.
The skeleton representation is shown in Section 3.1.3 – Structure Skeleton After Thinning Algorithm.

2.2.1.4 Removing small triangulations

Generating connections based on a maximum distance criterion leads to the creation of small, closed triangulations within the skeleton, as shown in the figure below:

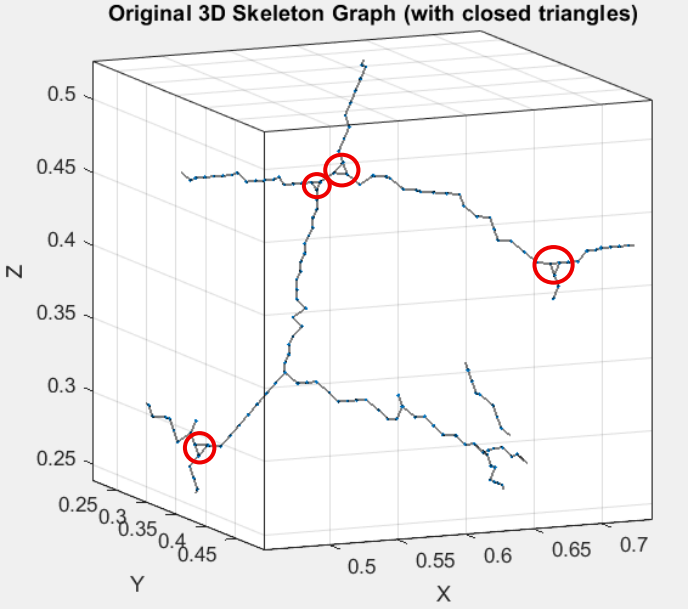


Figure 2.2.7: Closed loop triangles

The following portion of code is dedicated to detecting and removing closed loop triangles in the three-dimensional graph generated from the skeleton of the structure. These triangles are geometric artifacts with no physical relevance in the real structure, caused by the maximum distance criterion used to connect the skeleton points. If not removed, they reduce the accuracy of subsequent analyses on the node degree distribution and the branching node chains.

```
% REMOVING CLOSED LOOP TRIANGLES
% Step 1: Detect all triangles in the graph
A = adjacency(G);  % Get the adjacency matrix of the graph G
triangles = [];    % Initialize an empty list to store detected triangles
n = numnodes(G);   % Get the total number of nodes in the graph
% Loop over all triplets of nodes to find mutually connected triangles
for i = 1:n-2
    for j = i+1:n-1
        if A(i,j)  % Check if there's an edge between nodes i and j
            for k = j+1:n
                % If all three nodes are connected pairwise, it's a triangle
                if A(i,k) && A(j,k)
                    triangles = [triangles; i j k];  % Store the triangle
                end
            end
        end
    end
end
% Step 2: Identify which triangle edges to remove (to break minimal loops)
edgesToRemove = [];  % Initialize list of edges that will be removed
% Loop through each detected triangle
for t = 1:size(triangles,1)
```

```
    tri = triangles(t,:);      % Get node indices of the triangle
    pts = skeletonpoints(tri,:);          % Get the 3D coordinates of the triangle
                                            vertices
    % Compute Euclidean distances between each pair of triangle vertices
    d12 = norm(pts(1,:) - pts(2,:));
    d23 = norm(pts(2,:) - pts(3,:));
    d31 = norm(pts(3,:) - pts(1,:));
    dists = [d12, d23, d31];              % Store the three edge lengths
    % Check if this triangle is a minimal loop (all edges ≤ sqrt(3)*delta + tol)
    if max(dists) <= sqrt(3)*delta + tol
        % Identify the longest edge
        [~, maxIdx] = max(dists);
        if maxIdx == 1
            edgeToRemove = sort([tri(1), tri(2)]);  % First-edge nodes
        elseif maxIdx == 2
            edgeToRemove = sort([tri(2), tri(3)]);  % Second-edge nodes
        elseif maxIdx == 3
            edgeToRemove = sort([tri(3), tri(1)]);  % Third-edge nodes
        end
        % Add the edge to the removal list if it's not already included
        if isempty(edgesToRemove) || ~ismember(edgeToRemove, edgesToRemove,'rows')
            edgesToRemove = [edgesToRemove; edgeToRemove];
        end
    end
end
% Step 3: Actually remove the edges from the graph
for k = 1:size(edgesToRemove,1)
    edgeID = findedge(G, edgesToRemove(k,1), edgesToRemove(k,2));  % Get edge
                                                                     index in G
    if edgeID > 0
        G = rmedge(G, edgeID);  % Remove the edge from the graph
    end
end
% Plot cleaned skeleton graph
figure;
plot(G, 'XData', skeletonpoints(:,1), 'YData', skeletonpoints(:,2), 'ZData',
skeletonpoints(:,3),'NodeLabel', {}, 'EdgeColor', 'k', 'LineWidth', 1,
'MarkerSize', 1);
title('Cleaned 3D Skeleton Graph (without closed triangles)');
xlabel('X'); ylabel('Y'); zlabel('Z'); axis equal; grid on;
```

The first part of the code constructs the adjacency matrix *A* associated with the graph *G*, where each element *A(i,j)* equals 1 if there is a direct connection (an edge) between node *i* and node *j*. After initializing the vector to store triangles and determining the total number of nodes *n*, a triple nested loop over indices *i*, *j*, and *k* is executed to identify all triplets of nodes connected pairwise. If the three nodes *i*, *j*, and *k* are mutually adjacent, they form a closed triangle which is then added to the triangles list.
Next, the code examines each detected triangle to evaluate its geometric validity. The nodes composing it are converted into three-dimensional coordinates using the skeletonpoints array, and for each triangle the three Euclidean distances *d12*, *d23*, and *d31* between pairs of vertices are calculated. If all three side lengths are below a threshold equal to the largest diagonal of the voxel cube (with some tolerance), the triangle is considered a minimal closed loop (a local cycle related to the grid resolution rather than a real structural bifurcation). In this case, the longest edge of the triangle is selected, interpreted as an unnecessary diagonal connection, and added to the list edgesToRemove, avoiding duplicates.

Finally, the actual removal is performed by iterating through the *edgesToRemove* list, locating the edges to delete in graph *G* using the *findedge* function, and removing them with *rmedge*.
The final skeleton without minimal loop triangles is shown in section 3.1.4 – Skeleton of the Structure After Removing Closed Triangulations.

2.2.1.5 First thickness analysis (constant radius reconstruction)

The study of the thickness variation of the structure along its various branches is of fundamental interest.
An initial analysis to assess this aspect was carried out by reconstructing the structure from the skeleton points, under the approximation of a constant circular cross-section with a given radius. This radius is calculated so as to achieve the same relative density as the originally set structure.
The reconstructed spinodoid will then be compared with the original one to understand the deviation of the simplified model from the real structure, which will subsequently lead to a more detailed study of the thickness variation along the branches of the structure.

```
%% BULK STRUCTURE RECONSTRUCTION
%Reconstruction of the main skeleton as a binary matrix with the same dimensions
as the original one
binaryMatrix_cleaned = false(size(domain_bin));
idx = zeros(size(skeletonpoints, 1), 1);
for i = 1:length(idx)
    d = sqrt((x_grid(:) - skeletonpoints(i,1)).^2 + (y_grid(:) -
skeletonpoints(i,2)).^2 + (z_grid(:) - skeletonpoints(i,3)).^2);
    [~, idx(i)] = min(d); % linear idx of each point of the skeleton in the
                            domain_bin matrix
End
% matrix idx of each point of the skeleton in the domain_bin matrix
[iy, ix, iz] = ind2sub(size(domain_bin), idx);
for k = 1:length(ix)
    binaryMatrix_cleaned(ix(k), iy(k), iz(k)) = true;
end
% Morphological dilation loop
targetVolume = round(relative_density * numel(domain_bin)); % must be an integer
                                                              (number of true points)
tolerance = 0.01 * targetVolume;
expansionRadius = 4;  % initialize expansion radius
binaryMatrix_loop = binaryMatrix_cleaned; % initialize matrix with only the points
                                            of the skeleton
currentVolume = nnz(binaryMatrix_loop);   % initialize volume
while abs(currentVolume - targetVolume) >= tolerance
    expansionRadius = expansionRadius * (1.02 * (currentVolume < targetVolume) +
                      0.98 * (currentVolume > targetVolume));
    % increasing or decreasing of 2% the expansion radius until reaching the
    target volume (with a tolerance)
    [Xs, Ys, Zs] = ndgrid(-expansionRadius:expansionRadius);
    % Xs,Ys,Zs are 3D matrix of numbers from -expansionRadius to expansionRadius
    in each dimension
    sphereMask = (Xs.^2 + Ys.^2 + Zs.^2) <= expansionRadius^2; %creating spherical
                                                                binary object
    binaryMatrix_loop = imdilate(binaryMatrix_cleaned, sphereMask); %dilating each
    point of the skeleton with spherical dimensions saved in spheremask
```

```
    currentVolume = nnz(binaryMatrix_loop); % computing final volume at each
                                            iteration
end
% compute the section radius in physical coordinates
section_radius = (expansionRadius-1) * voxelLength;
disp(['The constant section radius of the reconstructed structure is: ',
num2str(section_radius)]);
% compute the final volume fraction
FinalVolumeFrac = nnz(binaryMatrix_loop) / numel(domain_bin);
disp(['Final Volume Fraction of the riconstructed structure: ',
num2str(FinalVolumeFrac)]);
```

Up to the previous step, the skeleton points were stored within the *skeletonpoints* matrix. However, the current objective is to represent the skeleton (and subsequently the reconstructed structure) within a matrix having the same dimensions as the original domain. This will allow for the association with the physical coordinate vectors *x_vec*, *y_vec*, and *z_vec*, and enable a direct comparison with the actual structure.
To achieve this, a 3D binary matrix is initialized with the same dimensions as *domain_bin*, where all entries are initially set to false.
Using a *for* loop and the physical coordinates, each skeleton point is mapped to its corresponding linear index in the 3D grid, which is then converted to matrix index using *ind2sub*.
These points are then set to true in the *binaryMatrix_cleaned*, resulting in a binary representation of the skeleton within the original domain.
At this stage, the target volume, defined as the integer number of true points to be reached, is initialized. This value is computed by multiplying the desired relative density by the total number of points in the domain.
To ensure rapid convergence of the upcoming while loop for skeleton dilation, a tolerance of 1% with respect to the target volume is imposed.
An initial trial value for the dilation radius is set, and the matrix *binaryMatrix_loop* is also initialized. This matrix is initially equal to *binaryMatrix_cleaned* (i.e., it contains only the skeleton), and at each iteration it will also include the points added by spherical dilation.
The while loop increases or decreases the dilation radius by 2%, depending on whether the resulting volume is greater or smaller than the target volume (within the specified tolerance).
Three-dimensional matrices *Xs*, *Ys*, and *Zs* are then generated, each containing the respective coordinates of a new point grid that spans from -expansionradius to +expansionradius in each direction, with unit steps.
From this cubic grid, a binary matrix of the same dimensions is derived, containing true values arranged in such a way as to form a sphere centered in the matrix, with a radius (i.e., number of true points along the radius) equal to *expansionradius*.
Finally, the binary structure *binaryMatrix_loop* is reconstructed by applying the *imdilate* function. This function expands each true point in the initial matrix *binaryMatrix_cleaned* radially, by the number of points defined in the spherical mask *sphereMask*.
Each iteration concludes with the update of the current volume, which will be used to assess convergence in the next iteration. Once convergence is achieved, the result is an approximated reconstruction of the target structure. The result is shown in Section 3.1.5 – Reconstructed Structure Using Constant Radius Approximation.
The radius value, initially expressed in grid points, is then converted into physical coordinates, and the relative density (or volume fraction) of the resulting structure is subsequently computed.

It is also possible to assess the difference between the original structure and the approximate reconstruction with constant radius by highlighting, using different colors, the regions that have been added or removed.

```
%% VISUALIZATION OF ADDED/REMOVED PARTS AFTER THE RECONSTRUCTION
addedParts   = binaryMatrix_loop & ~domain_bin;
removedParts = domain_bin & ~binaryMatrix_loop;
volumreconstruct_error = (nnz(addedParts)+nnz(removedParts))/nnz(domain_bin)*100;
fprintf('Percentage volumetric error of the constant radius structure is: %.2f%%\n
', volumreconstruct_error);
figure; hold on;
patch(isosurface(y_grid,x_grid,z_grid,double(addedParts),.5),
'FaceColor','g','EdgeColor','none','FaceAlpha',.7);
patch(isosurface(y_grid,x_grid,z_grid,double(removedParts),.5),
'FaceColor','r','EdgeColor','none','FaceAlpha',.7);
title('Comparison of Added (green) and Removed (red) Parts After Reconstruction');
xlabel('X'); ylabel('Y'); zlabel('Z');
axis equal; view(3); camlight('headlight'); lighting gouraud; grid on; hold off;
```

AddedParts and *RemovedParts* are two matrices with the same dimensions as *domain_bin*, containing respectively the points that are true in the reconstructed structure and false in the original one, and vice versa.
By plotting both matrices using the *isosurface* and *patch* commands (to close the structure at the domain boundaries), the image shown in the figure below is obtained.

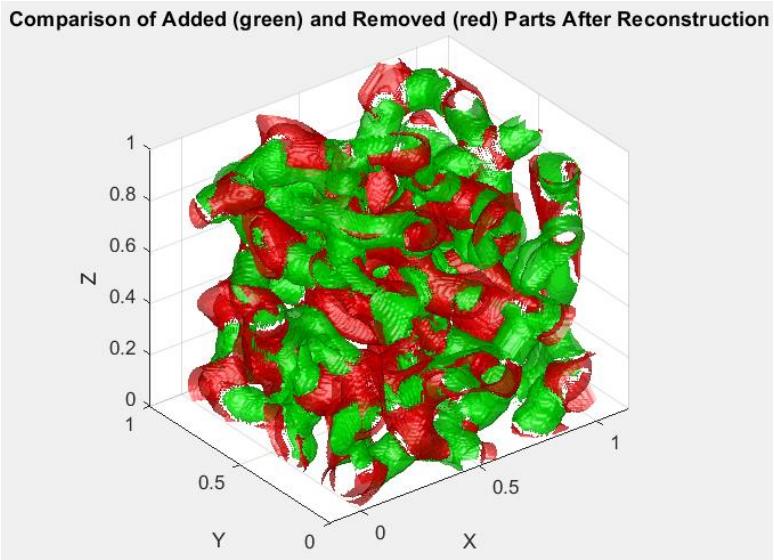


Figure 2.2.8: Added and removed parts after reconstruction

It is possible to analytically compute the volumetric error resulting from this type of reconstruction (as shown in the code above).
For several structures, this value can exceed 40%; therefore, a more robust algorithm will be implemented to provide a more accurate description of the thickness variation along the different branches of the structure (2.2.1.6.5 – Thickness analysis).
It should be noted that the study of certain morphological features based on a skeleton representation is not meaningful for every type of spinodoid structure. For instance, in the case of a lamellar structure, such analyses lose their significance.
This type of analysis is instead effective for isotropic, cubic, and also columnar spinodoids.

2.2.1.6 Geometric and morphological characterization

In the following, a few lines of code will be presented to geometrically and topologically characterize the structures, taking an isotropic structure as a reference. It is left to the reader's discretion to assess whether these parameters may also be applicable to the other types of structures considered.

2.2.1.6.1 Node degree

Thanks to the graph variable, which contains the connections between nodes forming each segment, it is possible to compute the degree of each node (number of edges or segments incident to it).
This analysis provides insight into features such as the number of branching points or terminal nodes in the structure and will serve as initial information for subsequent characterizations.

```
% 1)NODE DEGREE
NodeDegree = degree(G);
maxNodeDegree = max(NodeDegree);
nodesdegree_vec = zeros(maxNodeDegree, 1);
for i = 1:length(NodeDegree)
    nodesdegree_vec(NodeDegree(i)) = nodesdegree_vec(NodeDegree(i)) + 1;
end
```

For this purpose, the function *degree* is used, which returns a vector (*NodeDegree*) where the i-th element corresponds to the degree of the i-th node of the matrix *skeletonpoints*. Subsequently, the vector *nodesdegree_vec* is generated, which contains in its i-th position the number of nodes of degree i. Finally, the result is plotted in a bar chart, shown in Section 3.1.6.1 – Node Degree Distribution.

2.2.1.6.2 Total skeleton lenght

The total length of the skeleton can also be evaluated by leveraging the *connections* variable contained in the *graph* object, along with the coordinates of each skeleton point stored in *skeletonpoints:*

```
% 2)TOTAL SKELETON LENGTH
totalLength = 0;
for i = 1:size(connections, 1)
    node1 = connections(i, 1);
    node2 = connections(i, 2);
    totalLength = totalLength + norm(skeletonpoints(node1, :) -
skeletonpoints(node2, :));
end
disp(['The total skeleton length is: ', num2str(totalLength)]);
```

2.2.1.6.3 Branch chain analysis

The procedure adopted to identify and measure branch chains within the skeletal structure stored in the graph variable *G* is now described. A branch chain is defined as a connected

sequence of elementary segments (between two adjacent grid points) that links either two branching nodes, i.e., nodes with a degree greater than or equal to three, or a terminal node and a branching node. These chains constitute the main structural units of the skeletal network of the metamaterial and are fundamental for geometric analysis. Along these branch chains, the subsequent characterization of the structural cross-sectional variation will be conducted (Section 2.2.1.6.5 – Thickness analysis).

```
% 3)BRANCH SEGMENT LENGTH CALCULATION
% Calculate lengths of branch chain connecting branch nodes (degree >= 3)
branchNodes = find(NodeDegree >= 3); % Identify branch nodes (with degree >= 3)
chainLengths = [];  % Array to store chain lengths
chainNodes = {};    % Cell object to store the nodes of each chain
visitedPairs = false(numnodes(G), numnodes(G));  % squared logical matrix of
numnodes(G)xnumnodes(G) dimension, to track if a pair of nodes has already been
processed
for i = 1:length(branchNodes)
    startNode = branchNodes(i);
    nbrs = neighbors(G, startNode); % returns a vector of nodes directly connected
                                      to startNode in the graph G
    for j = 1:length(nbrs)
        if visitedPairs(startNode, nbrs(j)) || visitedPairs(nbrs(j), startNode)
            continue;  % if the segment has already been processed
        end
        % if the selected neighbor is a branchnode, a branch chain is generated
        if NodeDegree(nbrs(j)) >= 3
            segLength = norm(skeletonpoints(startNode, :) -
            skeletonpoints(nbrs(j), :));
            chainLengths(end+1) = segLength; % add the segment lenght in
                                               chainLengths vector
            visitedPairs(startNode, nbrs(j)) = true; % mark connection of visited
                                                       points
            visitedPairs(nbrs(j), startNode) = true;
            chainNodes{end+1} = [startNode, nbrs(j)]; % add in chainNodes{} cell
            the vector containing index of nodes of the branch segment
        else
            % if the first neighbor isn't a branch nodes continue on the chain
            currentNode = nbrs(j);
            prevNode = startNode;
            segLength = norm(skeletonpoints(startNode, :) -
                        skeletonpoints(currentNode, :)); % update the chain length
            segmentNodesChain = [startNode, currentNode];  % adding the two first
                                                             nodes of the chain
            while true
                visitedPairs(prevNode, currentNode) = true;
                visitedPairs(currentNode, prevNode) = true;
                nbrs2 = neighbors(G, currentNode); % Returns a vector of nodes
                directly connected to currentNode in the graph G
                nbrs2(nbrs2 == prevNode) = []; % Removing prevNode from the
                neighbors vector to not go back in the chain
                % Filter already visited connections
                unvisited = nbrs2(~visitedPairs(currentNode, nbrs2) &
                ~visitedPairs(nbrs2, currentNode));
                if isempty(unvisited)
                    break;  % No unvisited neighbor, end chain
                end
                nextNode = unvisited(1);  % Pick the first unvisited neighbor
                segLength = segLength + norm(skeletonpoints(currentNode, :) -
                            skeletonpoints(nextNode, :)); %update the chain length
```

```
                segmentNodesChain = [segmentNodesChain, nextNode]; % adding
                nextNode to the chain
                if NodeDegree(nextNode) >= 3
                    visitedPairs(currentNode, nextNode) = true;
                    visitedPairs(nextNode, currentNode) = true;
                    break; % if nextNode is a branch node a branch chain is
                             generated
                else
                    prevNode = currentNode; %update prevNode
                    currentNode = nextNode; %update currentNode
                end
            end
            chainLengths(end+1) = segLength; % adding the final chain length to
                                             the chainLengths vector
            chainNodes{end+1} = segmentNodesChain;  % add in chiainNodes{} cell
            the vector containing nodes indexes of the final branch chain
        end
    end
end
```

The analysis begins with the identification of all branching nodes. The function *find* was used to obtain, in the vector *branchNodes*, the indices of nodes for which the *NodeDegree* vector has a value greater than or equal to 3. Subsequently, three fundamental structures are initialized: the vector *chainLengths*, in which the lengths of the different branch chains will be stored; the cell array *chainNodes*, which will contain, for each chain, an ordered list (vector) of the nodes indexes that compose it; and finally the boolean matrix *visitedPairs*, used to record connections between node pairs that have already been processed, in order to avoid duplications.

For each node in *branchNodes*, the code examines its immediate neighbors, obtained through the function *neighbors(G, startNode)*, which returns the nodes directly connected to *startNode* in the graph *G*. For each adjacent node, it is checked whether the connection pair has already been explored (via *visitedPairs*): if so, the loop continues with the next neighbor.

If the adjacent node also has a degree greater than or equal to 3, it constitutes a branch chain consisting of a single segment. Its length is calculated as the Euclidean distance between the spatial coordinates of the two nodes, stored in *skeletonpoints*, and saved in *chainLengths*. The indices of the two end nodes are stored in the corresponding entry of *chainNodes*.

If, instead, the adjacent node has degree 2, a loop is initiated to iteratively traverse the nodes along the chain until a second branching node is reached. The variable *currentNode* keeps track of the node currently under examination, while *prevNode* stores the previous node in the sequence to prevent backtracking. At each iteration, the total chain length is updated (by accumulating the distance between consecutive nodes), and the list of traversed nodes, stored in *segmentNodesChain*, is extended.

The list of neighbors of the current node is filtered to remove the previous node and to exclude already visited connections. If no unexplored neighbors remain, the chain is terminated. Otherwise, the traversal continues along the first available neighbor. When a node with degree ≥ 3 or a terminal node is finally reached, the chain is considered complete: the collected data are stored in *chainLengths* (total length) and *chainNodes* (sequence of nodes forming the chain).

This procedure ensures a complete analysis of the network, recording each chain connecting branching or terminal nodes through intermediate degree-2 nodes only once, without duplications or infinite loops.

For greater clarity, the results are then plotted in a bar chart, showing the percentage of segments for each length range of 0.1 units:

```
% Calculate the total number of branch segments
branchChain_number = length(chainLengths);
% Define the bin edges for the histogram
edges = 0:0.1:max(chainLengths);
% Compute the frequency of segments within each bin
[counts, ~] = histcounts(chainLengths, edges);
% Calculate the relative frequency (percentage)
relativeFreq = (counts / branchChain_number) * 100;
% Create the bar plot
figure;
bar(edges(1:end-1), relativeFreq, 'histc');
% Label the axes and the title
xlabel('Segment Length');
ylabel('Percentage');
title('Percentage Distribution of Segment Lengths');
grid on;
```

The result is shown in Section 3.1.6.2 – Branch Chain Length Distribution.
Furthermore, as an additional check of the method's correctness, the total skeleton length obtained previously was compared with the sum of the branch chain lengths:

```
%check
Totalskeletonlenght_check=sum(chainLengths);
inaccuracy=(totalLength-Totalskeletonlenght_check)/totalLength;
```

The two values are identical in the MATLAB workspace, and the value of the variable inaccuracy is on the order of 10^{-12}.

2.2.1.6.4 Segments orientation

After identifying and computing the lengths of the branch chains, the study proceeds with the analysis of their orientations in three-dimensional space within the structure. The following code is intended to analyze the distribution of segment orientations along the branch chains in terms of azimuthal and elevational angles:

```
% 4) DISTRIBUTION OF SEGMENTS' ORIENTATION
window_size = 4;   % Number of points between endpoints of the orientation vector
all_azimuths = [];                % Initialize array for storing azimuth angles
all_elevations = [];              % Initialize array for storing elevation angles
for c = 1:length(chainNodes)        % Loop through all chains
    chain = chainNodes{c};          % Get current chain (list of point indices)
    coords = skeletonpoints(chain, :);    % Retrieve 3D coordinates of the chain
                                            points
    % Skip chains that are too short to compute orientation
    if size(coords,1) < window_size
        continue
    end
    % Slide a window along the chain to compute orientation vectors
    for i = 1:(size(coords,1) - window_size + 1)
        p1 = coords(i, :);                      % Starting point of the segment
        p2 = coords(i + window_size - 1, :);    % Ending point of the segment
        vec = p2 - p1;                      % Orientation vector between p1 and p2
```

```
        % Compute azimuth: angle in XY plane, symmetrized to [0, 180)
        azimuth = atan2(vec(2), vec(1));         % [-pi, pi]
        azimuth = abs(mod(azimuth, 2*pi) - pi);  % [0, 180)
        azimuth_deg = rad2deg(azimuth);
        % Compute elevation: angle from XY plane to the vector, in range [0, π/2]
        elevation = atan2(abs(vec(3)), norm(vec(1:2))); % [0, pi/2]
        elevation_deg = rad2deg(elevation);             % [0, 90]
        % Store computed angles
        all_azimuths(end+1) = azimuth_deg;
        all_elevations(end+1) = elevation_deg;
    end
end
% Plot probability density functions (PDFs) of azimuth and elevation angles
figure;
subplot(1,2,1);
histogram(all_azimuths, 50, 'Normalization', 'pdf');   % PDF of azimuth angles
xlabel('Azimuth [°]');
ylabel('Probability density');
title('Azimuthal PDF');
subplot(1,2,2);
histogram(all_elevations, 50, 'Normalization', 'pdf'); % PDF of elevation angles
xlabel('Elevation [°]');
ylabel('Probability density');
title('Elevation PDF');
```

The starting data is the variable *chainNodes*, a cell array containing, for each branch chain, the vector of indices of the nodes that compose it. These indices refer to the rows of *skeletonpoints*, an Nx3 matrix that contains the three-dimensional spatial coordinates (x, y, z) of all nodes belonging to the structure's skeleton. Each element of *chainNodes* therefore represents a continuous chain of connected nodes, that is, a branch of the skeleton graph delimited by branching nodes (degree ≥ 3) and/or terminal nodes.
To analyze the distribution of local orientations along these chains, "extended" segments composed of multiple consecutive grid points are considered, here chosen with a window size of 4. Since the initial grid is discrete and cubic, analyzing segments of only two consecutive points would result in a limited and quantized number of orientations (few possible directions of the vectors between adjacent points on the grid). By extending the segment to multiple points (4 nodes in this case), a smoother and more meaningful estimate of the local orientation is obtained, overcoming the quantization imposed by spatial discretization.
The code then iterates over all chains contained in chainNodes. For each chain, it extracts the 3D coordinates of the corresponding nodes from skeletonpoints. Chains that are too short (fewer than 4 points) are skipped, as it is not possible to calculate a segment of the desired length.
For each sufficiently long chain, a sliding window of size 4 is applied: for each possible position along the chain, the starting point *p1* and the ending point *p2* of the window are identified, and the oriented vector *vec* = *p2* - *p1* is calculated. This vector approximates the local orientation of the branch in that portion of the chain.
At this stage, two angles are calculated to describe the three-dimensional orientation of the vector vec:

- Azimuthal (azimuth): defined as the angle in the XY plane relative to the X-axis, calculated using the function *atan2(vec(2), vec(1))*, which returns values in the range $[-\pi;\pi]$. To introduce directional symmetry, i.e., so that a vector and its

opposite are considered to have the same orientation (without imposing a direction, which in this case would have no physical meaning), the azimuth is normalized to the range [0°, 180°] using the operation *abs(mod(azimuth,* $2 \cdot \pi$*) -* π*)*. This symmetrizes the angle by considering a semicircular domain, thereby avoiding double representations of opposite orientations.

- Elevation: defined as the angle between the vector and the XY plane, calculated as *atan2(abs(vec(3)), norm(vec(1:2)))*, that is, the arctangent of the absolute value of the Z component divided by the norm of the X and Y components. The absolute value ensures that the angle is positive, without distinguishing between upward or downward directions. The elevation is thus in the range from 0 to 90 degrees.

The azimuthal and elevational angles calculated for all segments are stored in the arrays *all_azimuths* and *all_elevations*. After processing all chains, these angles, originally in radians, are converted to degrees using *rad2deg* for a more intuitive angular interpretation. Finally, to visualize the statistical distribution of the orientations, two histograms normalized to probability density (pdf) are generated.
The results are shown in Section 3.1.6.3 – Probability Density Distribution of Skeleton Branch Orientation.

2.2.1.6.5 Thickness analysis

A more detailed analysis of the local thickness profile along selected chains is presented below. The local thickness is calculated as the radius of the largest inscribed sphere within the structure at the points of each analyzed chain. Although the code potentially allows analysis of the entire structure, the following focuses on a limited number of branches, selected according to specific criteria, in order to obtain clearer and more meaningful results and to simplify their presentation through plots and figures.

```
% 5) NORMALIZED RADIUS PROFILE ALONG BRANCHES
n_samples = 4;  % Number of branch chains to sample for analysis
min_nodes = round(0.2*grid_res);  % Minimum number of nodes required in a branch
                                    chain to be considered
margin_points = round(0.1 * grid_res);  % Margin (in number of points) to exclude
                                          chains near the domain border
% Filter chains that are long enough
initialCount = numel(chainNodes);  % Count initial number of chains
long_chainnodes = chainNodes(cellfun(@(x) numel(x) > min_nodes, chainNodes));
% Keep only chains with more than min_nodes
filteredCount = numel(long_chainnodes);  % Count how many chains passed the filter
disp(['Initial chains: ', num2str(initialCount), ', after node count filter: ',
num2str(filteredCount)]);
assert(filteredCount > 0, 'No chain with >30 nodes.'); % Stop if no chains meets
                                                         the length requirement
% Exclude segments near the domain border
domain_matrix_size = size(domain_bin);  % Get the size of the 3D binary domain
is_internal = @(points) all(points > margin_points & points <= (domain_matrix_size
              - margin_points), 2);  % Define function to check if a point is
                                       within margin
non_border_chains = {};  % Initialize output cell array
for i = 1:numel(long_chainnodes)
```

```
    chain = long_chainnodes{i};  % Get the current chain
    coords = skeletonpoints(chain, :);  % Get 3D coordinates of the chain
    points = round(coords/domain_size * (grid_res - 1)) + 1;  % Convert normalized
    coordinates to points indices
    if all(is_internal(points))  % Keep only chains fully inside the internal
    domain
        non_border_chains{end+1} = chain;
    end
end
disp(['Long non-border chains: ', num2str(numel(non_border_chains)), ' out of ',
num2str(numel(long_chainnodes))]);
assert(~isempty(non_border_chains), 'No internal chain found.');  % Stop if no
                                                                  internal chains found
% Randomly select a subset of chains to visualize and analyze
selectedIndices = randperm(numel(non_border_chains), min(n_samples,
numel(non_border_chains)));
long_chainnodes = non_border_chains(selectedIndices);  % Select the random chains
% Initialize container to store 3D coordinates of selected chains
chainnodes_coords = cell(1, numel(long_chainnodes));
% Visualize selected segments
figure;
axis equal;
hold on;
colors = lines(numel(long_chainnodes));  % Generate distinguishable colors for
                                           each chain
for i = 1:numel(long_chainnodes)
    chain = long_chainnodes{i};
    coords = skeletonpoints(chain, :);  % Get 3D coordinates
    chainnodes_coords{i} = coords;  % Store coordinates
    plot3(coords(:,1), coords(:,2), coords(:,3), 'Color', colors(i,:),'LineWidth',
    2);  % Plot line
    scatter3(coords(1,1), coords(1,2), coords(1,3), 30, 'filled',
    'MarkerFaceColor', colors(i,:));  % Mark starting point
end
p = patch(surf, 'FaceColor', [0.7, 0.7, 0.7],'EdgeColor', 'none','FaceAlpha',0.2);
% Plot transparent surface
xlabel('X');
ylabel('Y');
zlabel('Z');
grid on;
title('Selected Chains with Different Colors (Internal Only)');
hold off;
view(3)
% Compute Euclidean distance transform of the binary domain
distance_map = bwdist(~domain_bin);  % Distance of each point to the nearest
                                       background point
% Initialize containers to store radius and points indices for each chain
chain_radii = cell(1, numel(long_chainnodes));
segment_subs  = cell(1, numel(long_chainnodes));
for i = 1:numel(long_chainnodes)
    points_indices = round(chainnodes_coords{i}/domain_size .*(grid_res - 1)) + 1;
    % Convert coordinates to points indices
    segment_subs{i} = points_indices;  % Store points coordinates
    x = points_indices(:,1);  % Extract x-coordinates
    y = points_indices(:,2);  % Extract y-coordinates
    z = points_indices(:,3);  % Extract z-coordinates
    idx_linear = sub2ind(size(domain_bin), y, x, z);  % Convert to linear indices
    chain_radii{i} = distance_map(idx_linear);  % Get local radius from distance
                                                  map
```

```
end
% Plot normalized radius profile along each chain
figure;
hold on;
for i = 1:numel(chain_radii)
    coords_points = segment_subs{i};  % Get points coordinates
    dists = sqrt(sum(diff(coords_points).^2, 2));  % Compute Euclidean distances
                                                     between consecutive points
    cum_length = [0; cumsum(dists)];  % Cumulative length along the chain
    norm_length = cum_length / cum_length(end);  % Normalize cumulative length
                                                   from 0 to 1
    radii = double(chain_radii{i});   % Convert radius to double
    norm_radii = radii / max(radii);  % Normalize radius from 0 to 1
    plot(norm_length, norm_radii, 'Color', colors(i,:), 'LineWidth', 2, ...
        'DisplayName', ['Segment ', num2str(i)]);  % Plot normalized profile
end
xlabel('Normalized Position along Chain');
ylabel('Normalized Radius (bwdist)');
title('Normalized Radius Profiles along Chains');
legend show;
grid on;
```

Initially, the variables *n_samples*, indicating the maximum number of chains to be analyzed, and *min_nodes*, representing the minimum number of points a chain must contain to be considered significant (here set to 20% of the linear resolution along each axis of the grid), are defined. An internal margin, defined as 10% of the grid's linear resolution, is introduced to exclude chains near the domain boundaries. In fact, in these regions the structure may be partially truncated by the domain boundaries: if the skeleton is too close to a boundary and the cross-section is large, part of the structure's volume can be cut off. An example is shown in Figure 2.2.9: Effect of Domain Boundary Truncation on the Structure and Its Cross-Section. Consequently, the cross-section values at points where the structure is truncated do not accurately reflect the true cross-sectional profile, as they are affected by this boundary effect.

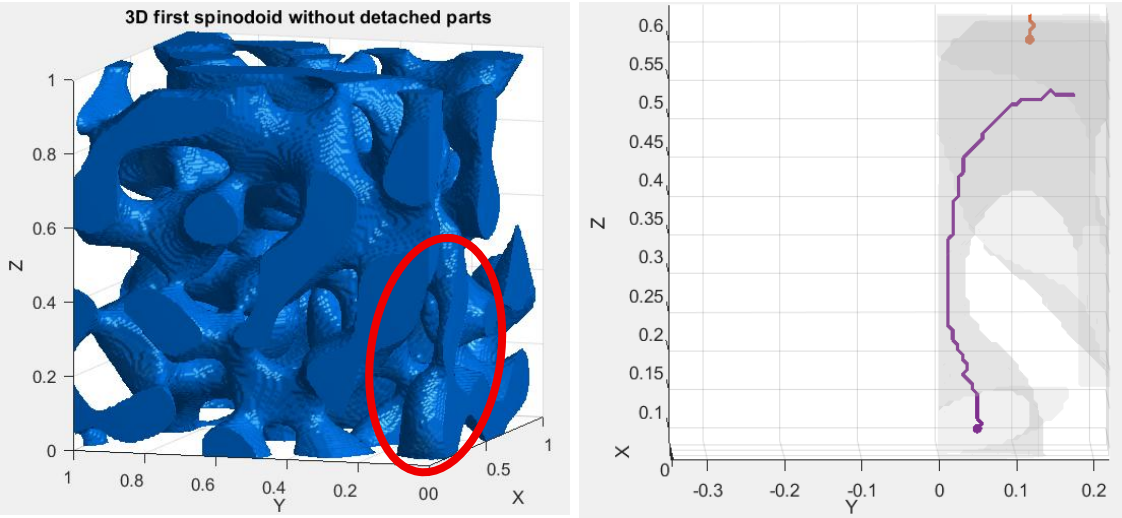


Figure 2.2.9: Effect of Domain Boundary Truncation on the Structure and Its Cross-Section

The first filtering step consists of counting the initial chains (*initialCount*) and retaining only those that exceed the minimum number of points (*long_chainnodes*). An assertion (*assert*) interrupts execution if no chain meets the length criterion.
Subsequently, chains extending too close to the domain boundaries are excluded. To this end, a function *is_internal* is defined to verify whether all points of a chain lie within the specified limits. For each long chain, the normalized coordinates of the points (*skeletonpoints*) are converted into integer indices relative to the three-dimensional grid, and the funtion *is_internal* is applied. Only chains satisfying both requirements (minimum length and distance from the domain boundary) are retained in *non_border_chains*.
Once the valid chains are filtered, a random subset (up to a maximum of *n_samples*) is selected using *randperm*. The selected chains are saved, and their coordinates are stored in *chainnodes_coords*.
For visualization, a new figure is opened, and each chain is plotted with a distinct color generated by the *lines* function. Each chain is traced with a line, and its starting point is highlighted with a filled marker. Simultaneously, the outer surface of the spinodoid structure is visualized as a transparent object to provide spatial context.
Next, the function *bwdist* is used to calculate the minimum distance of each point within the binary domain to the nearest zero value (i.e., the external boundary). For the skeleton points of each chain, this value represents the local radius of the largest inscribed sphere within the structure.
For each selected chain, the physical coordinates of the points are transformed into grid indices (*points_indices*) and then converted into linear indices (*sub2ind*) compatible with the three-dimensional distance map (*distance_map*). These indices are used to obtain the corresponding minimal radius values, which are stored in *chain_radii*.
Finally, in a new figure, the normalized radius profile along each chain is plotted. The Euclidean distance between consecutive points (*dists*) and the cumulative length along the chain (*cum_length*) are calculated. The cumulative length is normalized between 0 and 1 (*norm_length*). The local radii are also normalized relative to the maximum value of the respective chain (*norm_radii*). Each profile is plotted using the same color assigned to the corresponding chain in the previous plot. The x-axis represents the normalized position along the chain, while the y-axis represents the normalized radius derived from the *bwdist* map.
The results are presented in Section 3.1.6.4 – Cross-Section Profile of the Structure Along the Skeleton.

2.2.1.7 Hierarchical structures

Finally, the code for generating a hierarchical spinodoid structure based on the previously analyzed primary structure is implemented.
The procedure involves creating a new GRF and, subsequently, a new binary field (after applying the threshold), in a manner entirely analogous to that of the primary structure, with the difference that this time the new GRF is considered only within the domain defined by the original spinodoid.

```
%% HIERARCHY
%Generation of a new GRF
relative_density_2 = 0.7;
n_waves_2 = 100;
wave_lenght_2 = domain_size/20;
wavenumber_2 = 2*pi/wave_lenght_2;
```

```
theta1_2 = 15 * (pi / 180);
theta2_2 = 15 * (pi / 180);
theta3_2 = 15 * (pi / 180);
% Generate random waves
v_2 = zeros(n_waves_2, 3);
phase_shift_2 = zeros(n_waves_2, 1);
for i = 1:n_waves_2
    theta = 2*pi*rand;
    phi = asin(2 * rand() - 1);
    while ~ (abs(cos(phi)*cos(theta))>=cos(theta1_2) ||
          abs(cos(phi)*sin(theta))>=cos(theta2_2) || abs(sin(phi))>=cos(theta3_2))
        theta = 2*pi*rand;
        phi = asin(2 * rand() - 1);
    end
    v_2(i, :) = [cos(phi)*cos(theta), cos(phi)*sin(theta), sin(phi)];
    phase_shift_2(i) = 2 * pi * rand;
end
% Compute the GRF field
GRF_2 = zeros(grid_res, grid_res, grid_res);
for i = 1:length(x_vec)
    for j = 1:length(y_vec)
        for k = 1:length(z_vec)
            for w = 1:n_waves_2
                GRF_2(i,j,k) = GRF_2(i,j,k) + cos(wavenumber_2 * v_2(w, :) *
                              [x_vec(i), y_vec(j), z_vec(k)]' + phase_shift_2(w));
            end
        end
    end
end
GRF_2 = GRF_2 * sqrt(2/n_waves_2);
GRF_2 = (GRF_2 - mean(GRF_2(:))) / std(GRF_2(:));
thresh_2 = sqrt(2) * erfinv(2 * relative_density_2 - 1);
domain_bin_2 = (GRF_2 < thresh_2) & (domain_bin == 1);
```

In the last line, the previously described criterion for generating the binary field is translated into code.
Similarly to the primary structure, isolated parts of the structure that are not mechanically significant are removed in the same manner.

```
%% KEEPING ONLY THE MAIN STRUCTURE WITHOUT ISOLATED PARTS
Connectivity_2 = bwconncomp(domain_bin_2, 6); % 6-connectivity in 3D
% Compute number of points in each connected component
numPixels_2 = cellfun(@numel, Connectivity_2.PixelIdxList);
% Find the index of the largest component
[~, idxLargest_2] = max(numPixels_2);
% Create a new binary matrix containing only the main component
domain_bin_2 = false(size(domain_bin_2));
domain_bin_2(Connectivity_2.PixelIdxList{idxLargest_2}) = true;
```

An example of a structure that can be obtained is shown below:

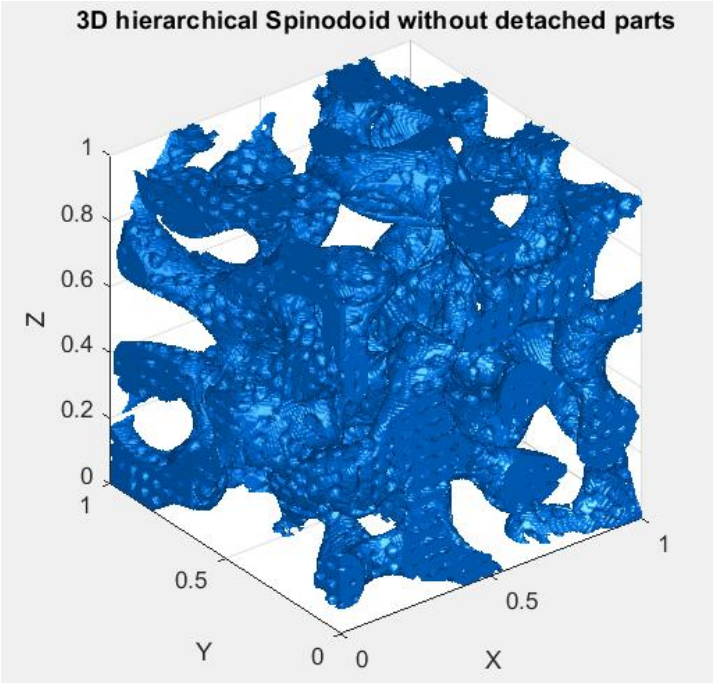


Figure 2.2.10: Example of a Hierarchical Spinodoid Structure

Other examples of obtainable structures are shown in Section 3.1.7 – Examples of Obtainable Hierarchical Structures.

2.2.1.8 Preliminary voxelization test

The study of hierarchical spinodoid structures in this thesis focuses on the mechanical property of uniaxial compressive stiffness. To determine this property, it is necessary to convert the structure generated in MATLAB into a format compatible with import into a finite element analysis software such as Abaqus. Since the structure is currently defined as a binary matrix of points, a voxelized reconstruction has been adopted, using cubic voxels whose origins correspond to the active points (value 1) of the binary grid.
To become familiar with the voxelization process, the first step was to write, in MATLAB, a Python file interpretable by Abaqus, capable of importing the primary structure as a union of cubic voxels.

```
% Function to generate an Abaqus input script to create a voxel-based structure
function export_voxel_grid_to_abaqus_instances(BW, voxel_size, output_file)
    % Get the size of the 3D binary grid
    [Nx, Ny, Nz] = size(BW);
    % Open the output file for writing
    fid = fopen(output_file, 'w');
    % Abaqus script header: import necessary modules and initialize model
    fprintf(fid, 'from abaqus import *\n');
    fprintf(fid, 'from abaqusConstants import *\n');
    fprintf(fid, 'import part, material, section, assembly\n\n');
    fprintf(fid, 'mdb.Model(name="VoxelModel")\n');
    fprintf(fid, 'model = mdb.models["VoxelModel"]\n');
    % Create a sketch of a square representing the voxel base
    fprintf(fid, 's = model.ConstrainedSketch(name="__profile__",
                    sheetSize=200.0)\n');
    fprintf(fid, 'size = %.4f\n', voxel_size);
```

```
    fprintf(fid, 's.rectangle(point1=(0.0, 0.0), point2=(size, size))\n');
    % Create a 3D cube from the sketch to be used as the voxel unit
    fprintf(fid, 'part = model.Part(name="VoxelCube", dimensionality=THREE_D,
                          type=DEFORMABLE_BODY)\n');
    fprintf(fid, 'part.BaseSolidExtrude(sketch=s, depth=size)\n\n');
    % Initialize assembly
    fprintf(fid, 'a = model.rootAssembly\n');
    % Counter for naming each voxel instance
    count = 0;
    % Loop through the binary voxel grid
    for x = 1:Nx
        for y = 1:Ny
            for z = 1:Nz
                if BW(x, y, z)
                    % Compute the voxel's spatial position
                    count = count + 1;
                    x0 = (x-1) * voxel_size;
                    y0 = (y-1) * voxel_size;
                    z0 = (z-1) * voxel_size;
                    % Create a new instance of the cube and place it at the
                      correct position
                    fprintf(fid, 'a.Instance(name="CubeInst_%d", part=part,
                    dependent=ON)\n', count);
                    fprintf(fid, 'a.translate(instanceList=("CubeInst_%d",),
                    vector=(%.4f, %.4f, %.4f))\n',count, x0, y0, z0);
                end
            end
        end
    end
    % Merge all cube instances into a single solid part
    fprintf(fid, '\nall_instances = [i.name for i in a.instances.values()]\n');
    fprintf(fid, 'a.InstanceFromBooleanMerge(name="MergedPart",
    instances=[a.instances[i] for i in all_instances],\n');
    fprintf(fid, '    keepIntersections=OFF, originalInstances=DELETE,
    domain=GEOMETRY)\n');
    % Print confirmation message
    fprintf(fid, '\nprint("Created %d cubes and merged them into a single
    Part.")\n', count);
    % Close the output file
    fclose(fid);
end
```

A function is then created that takes as input the three-dimensional binary matrix *BW*, whose dimensions are *Nx*, *Ny*, and *Nz*, obtained using the size function. Each active point corresponds to the origin of an elementary cube in the final geometry. The parameter *voxel_size* defines the side length of the cube, that is, the uniform spatial dimension of each voxel to be generated. Finally, *output_file* specifies the name of the text file that will be created to contain the Python code compatible with Abaqus.
The execution begins by opening a file for writing (*fopen*) named according to *output_file*. Within this file, the Python code is written, which, once executed in Abaqus, will create the desired geometry. The first part of the script is dedicated to importing the fundamental Abaqus modules required for defining the geometry, materials, and assembly. Subsequently, a new model named *VoxelModel* is created.
In Abaqus, it is not possible to create three-dimensional sketches or perform extrusions with variable depth or offsets relative to surfaces. Therefore, the only method to construct a voxelized structure is to define a two-dimensional square sketch, which is then extruded

to the desired depth to generate a cube. This cube is defined as a reference part, from which instances are subsequently created and translated to the correct positions within the assembly.
To this end, a sketch is created as a two-dimensional square profile (*ConstrainedSketch*). The side length of the square is defined by *size*, which corresponds to the value of the *voxel_size* parameter. This square represents the base of the cube to be extruded. From this sketch, a three-dimensional solid is generated with an extrusion depth equal to *voxel_size*, constituting the reference voxel.
The assembly where the voxel instances will be placed is initialized (model.rootAssembly). The variable *count* is an integer counter used to assign a unique name to each instance (cube) that will be created and positioned in three-dimensional space.
A triple nested loop iterates over all coordinates of the matrix *BW*. For each triple (x, y, z), if the corresponding value BW(x, y, z) is true, it indicates the presence of a point belonging to the structure at that grid position, and a cube must be generated with its origin positioned at the spatial coordinates (x-1)**voxel_size*, (y-1)**voxel_size*, (z-1)**voxel_size* (the subtraction of 1 converts the discrete index to a real spatial coordinate).
For each identified voxel, a Python instruction *a.Instance* is written to the file to create a new instance of the reference cube within the assembly, with a progressive name. Subsequently, the displacement of the instance is specified using the command *a.translate*, applying a translation vector equal to the spatial coordinates of the reference point. This correctly positions each cube to faithfully represent the voxelized structure derived from the initial grid.
At the end of the loop, once all instances have been generated and positioned, a Python script is added to the file to create a new entity named *MergedPart*, obtained through a Boolean merge operation (*InstanceFromBooleanMerge*). This operation combines all individual cubes into a single solid, removing intersections and eliminating the original instances.
The result obtained in Abaqus is shown in Section 3.1.8 – Voxelized Representation of the Structure Through Python File.
However, this methodology presents several limitations and will be replaced by an alternative technique for importing the voxelized structure, as discussed in Section 4.1.2 – Constraints of Voxelization via Python File.

2.2.2 FEM analysis (3D)

The MATLAB code to generate the complete .inp file of the FEM model, as introduced in Section 4.1.2 – Constraints of Voxelization via Python File, is reported below. In addition to the main structure, it also includes two plates (upper and lower) to which loads and constraints for uniaxial compression are applied. Prior to the actual writing of the .inp file, the first part of the code performs the voxelization and saves the data on the nodes and elements required for the creation of an orphan mesh in Abaqus. All subsequent steps for defining the model will be explained step by step.

```
%% CREATING VOXELIZED STRUCTURE
function export_voxels_to_inp_with_plates_as_one_part(BW, voxelLength, filename)
[Nx, Ny, Nz] = size(BW); % Get the size of the 3D binary domain
nodeGrid = zeros(Nx+1, Ny+1, Nz+1); % 3D grid to store node IDs
allNodes = []; % List of all nodes
allElements = []; % List of all elements
```

```
nodeID = 1; % Node counter
elementID = 1; % Element counter
offsets = [0 0 0; 1 0 0; 1 1 0; 0 1 0; 0 0 1; 1 0 1; 1 1 1; 0 1 1]; % Offsets for
                                                                    8 cube corners
% Spinodoid elements and nodes
for i = 1:Nx
    for j = 1:Ny
        for k = 1:Nz
            if BW(j,i,k) % Check if point is true
                elemNodeIDs = zeros(1,8); % Store node IDs for current element
                for v = 1:8
                    ii = i + offsets(v,1);
                    jj = j + offsets(v,2);
                    kk = k + offsets(v,3);
                    if nodeGrid(ii,jj,kk) == 0 % If node does not exist yet
                        x = (i-1 + offsets(v,1)) * voxelLength;
                        y = (j-1 + offsets(v,2)) * voxelLength;
                        z = (k-1 + offsets(v,3)) * voxelLength;
                        allNodes(end+1,:) = [nodeID, x, y, z]; % Add new node
                        nodeGrid(ii,jj,kk) = nodeID; % Store node ID in grid
                        elemNodeIDs(v) = nodeID; % Assign node to element
                        nodeID = nodeID + 1;
                    else
                        elemNodeIDs(v) = nodeGrid(ii,jj,kk); % Reuse existing node
                    end
                end
                allElements(end+1,:) = [elementID, elemNodeIDs]; % Add element
                                                                   and node IDs
                elementID = elementID + 1;
            end
        end
    end
end

% Plates
Lx = Nx * voxelLength; % Domain length in x
Ly = Ny * voxelLength; % Domain length in y
Lz = Nz * voxelLength; % Domain height
topZ = Lz;             % Top plate z-position
bottomZ = -voxelLength; % Bottom plate z-position
[topNodes, topElems, nodeID, elementID] = create_plate_reuse_nodes(0, 0, topZ, Lx,
Ly, voxelLength, nodeID, elementID, allNodes); % Top plate
allNodes = [allNodes; topNodes];
allElements = [allElements; topElems];
[bottomNodes, bottomElems, nodeID, elementID] = create_plate_reuse_nodes(0, 0,
bottomZ, Lx, Ly, voxelLength, nodeID, elementID, allNodes); % Bottom plate
allNodes = [allNodes; bottomNodes];
allElements = [allElements; bottomElems];

% WRITING INP FILE
filepath = fullfile('C:\Users\greco\Politecnico Di Torino Studenti Dropbox\Carlo
Alberto Greco\Erasmus NTNU master thesis\.inp files', filename);
fid = fopen(filepath, 'w');
fprintf(fid, '*HEADING\n');
fprintf(fid, '** Unified voxel structure with top and bottom rigid plates\n\n');
fprintf(fid, '*PART, NAME=SPINODOID_ALL\n');
fprintf(fid, '*NODE\n');
fprintf(fid, '%d, %.6f, %.6f, %.6f\n', allNodes'); % Write all node coordinates
fprintf(fid, '*ELEMENT, TYPE=C3D8, ELSET=AllElements\n');
```

```
fprintf(fid, '%d, %d, %d, %d, %d, %d, %d, %d, %d\n', allElements'); % Write all
                                                                      elements
fprintf(fid, '** Section: Section-1\n');
% Define solid section
fprintf(fid, '*Solid Section, elset=AllElements, material=Material-1\n');
fprintf(fid, '*END PART\n\n');
% ASSEMBLY
fprintf(fid, '**ASSEMBLY\n');
fprintf(fid, '*ASSEMBLY, NAME=Assembly\n');
fprintf(fid, '*INSTANCE, NAME=SPINODOID_ALL_INST, PART=SPINODOID_ALL\n');
fprintf(fid, '*END INSTANCE\n');
% Reference Points
topRP_ID = nodeID + 1; % ID for top reference point
bottomRP_ID = nodeID + 2; % ID for bottom reference point
xCenter = Lx / 2;
yCenter = Ly / 2;
zTop = topZ + voxelLength;
zBottom = bottomZ;
fprintf(fid, '*Node\n');
% Top reference node
fprintf(fid, '%d, %.6f, %.6f, %.6f\n', topRP_ID, xCenter, yCenter, zTop);
fprintf(fid, '*Node\n');
% Bottom reference node
fprintf(fid, '%d, %.6f, %.6f, %.6f\n', bottomRP_ID, xCenter, yCenter, zBottom);
fprintf(fid, '*Nset, nset=RP_TOP\n%d\n', topRP_ID); % Define top node set
fprintf(fid, '*Nset, nset=RP_BOTTOM\n%d\n', bottomRP_ID); % Define bottom node set
%CREATING SET FOR UPPER AND LOWER PLATES
topPlateElemIDs = topElems(:,1); % Get top plate element IDs
bottomPlateElemIDs = bottomElems(:,1); % Get bottom plate element IDs
% Elset + Surface UPPER PLATE
fprintf(fid, '*Elset, elset=TOPPLATE_S2_ELEMS, internal,
instance=SPINODOID_ALL_INST\n');
for i = 1:length(topPlateElemIDs)
    fprintf(fid, '%d', topPlateElemIDs(i));
    if mod(i,16) == 0 || i == length(topPlateElemIDs)
        fprintf(fid, '\n');
    else
        fprintf(fid, ', ');
    end
end
fprintf(fid, '*Surface, type=ELEMENT, name=TOP_SURFACE, internal\n');
fprintf(fid, 'TOPPLATE_S2_ELEMS, S2\n');
% Elset + Surface BOTTOMPLATE
fprintf(fid, '*Elset, elset=BOTTOMPLATE_S1_ELEMS, internal,
instance=SPINODOID_ALL_INST\n');
for i = 1:length(bottomPlateElemIDs)
    fprintf(fid, '%d', bottomPlateElemIDs(i));
    if mod(i,16) == 0 || i == length(bottomPlateElemIDs)
        fprintf(fid, '\n');
    else
        fprintf(fid, ', ');
    end
end
fprintf(fid, '*Surface, type=ELEMENT, name=BOTTOM_SURFACE, internal\n');
fprintf(fid, 'BOTTOMPLATE_S1_ELEMS, S1\n');
% Coupling
fprintf(fid, '**Coupling\n');
fprintf(fid, '*Coupling, constraint name=Top_Coupling, ref node=RP_TOP,
surface=TOP_SURFACE\n');
```

```
fprintf(fid, '*Kinematic\n');
fprintf(fid, '*Coupling, constraint name=Bottom_Coupling, ref node=RP_BOTTOM,
surface=BOTTOM_SURFACE\n');
fprintf(fid, '*Kinematic\n');
fprintf(fid, '*END ASSEMBLY\n');
% Material
fprintf(fid,'**MATERIAL\n');
fprintf(fid,'*Material, name=Material-1\n');
fprintf(fid,'*Elastic\n');
fprintf(fid,'1000., 0.3\n'); % Elastic modulus and Poisson's ratio
fprintf(fid,'*Plastic\n');
fprintf(fid,'10., 0.\n'); % Yield stress and plastic strain
% Step
fprintf(fid,'**STEP\n');
fprintf(fid,'*Step, name=Step-1, nlgeom=YES, inc=200\n');
fprintf(fid,'*Static\n');
fprintf(fid,'0.01, 1., 1e-05, 0.08\n');
% CONSTRAINTS
fprintf(fid,'** BOUNDARY CONDITIONS\n');
% Upper plate
fprintf(fid,'**Name: BC-1 Type: Displacement/Rotation\n');
fprintf(fid,'*Boundary\n');
fprintf(fid,'RP_TOP,1,1\n'); % Fix x displacement
fprintf(fid,'RP_TOP,2,2\n'); % Fix y displacement
fprintf(fid,'RP_TOP,3,3,-0.01\n'); % Apply z displacement
fprintf(fid,'RP_TOP,4,4\n');
fprintf(fid,'RP_TOP,5,5\n');
fprintf(fid,'RP_TOP,6,6\n');
% Lower plate
fprintf(fid,'**Name: BC-1 Type: Symmetry/Antisymmetry/Encastre\n');
fprintf(fid,'*Boundary\n');
fprintf(fid,'RP_BOTTOM, ENCASTRE\n'); % Fix all DOFs
% OUTPUT
fprintf(fid,'*Restart, write, frequency=0\n');
fprintf(fid,'*Output, field, variable=PRESELECT\n'); % Standard field output
% Force/Displacement of upper plate
fprintf(fid,'*Output, history\n');
fprintf(fid,'*node output, nset=RP_TOP\n');
fprintf(fid,'RF3, U3\n'); % Output vertical reaction force and displacement
fprintf(fid,'*end step\n');
fclose(fid);
end

function [nodes, elements, nextNodeID, nextElementID] = create_plate_reuse_nodes
(x0, y0, z0, Lx, Ly, voxelLength, startNodeID, startElementID, existingNodes)
    nodes = []; % List of new nodes to be created
    elements = []; % List of new elements to be created
    nextNodeID = startNodeID; % Node ID counter
    nextElementID = startElementID; % Element ID counter
    % Number of elements along x and y (single layer in z)
    Nx = Lx / voxelLength;
    Ny = Ly / voxelLength;
    % Grid to store node IDs of the bottom face
    nodeGrid = zeros(Nx+1, Ny+1);
    % Create bottom nodes with node reuse from existingNodes
    for i = 1:Nx+1
        for j = 1:Ny+1
            x = x0 + (i-1)*voxelLength;
            y = y0 + (j-1)*voxelLength;
```

```
            z = z0;
            % Check if node already exists at (x, y, z)
            idx = find(existingNodes(:,2)==x & existingNodes(:,3)==y &
            existingNodes(:,4)==z, 1);
            if isempty(idx)
                % Add new node
                nodes(end+1,:) = [nextNodeID, x, y, z];
                nodeGrid(i,j) = nextNodeID;
                nextNodeID = nextNodeID + 1;
            else
                % Reuse existing node ID
                nodeGrid(i,j) = existingNodes(idx,1);
            end
        end
    end
    % Create elements
    for i = 1:Nx
        for j = 1:Ny
            % Bottom face nodes (reused from grid)
            n1 = nodeGrid(i,j);
            n2 = nodeGrid(i+1,j);
            n3 = nodeGrid(i+1,j+1);
            n4 = nodeGrid(i,j+1);
            % Coordinates of the top face nodes
            coords_top = [ ...
                x0+(i-1)*voxelLength, y0+(j-1)*voxelLength, z0+voxelLength; % n5
                x0+(i)*voxelLength,   y0+(j-1)*voxelLength, z0+voxelLength; % n6
                x0+(i)*voxelLength,   y0+(j)*voxelLength,   z0+voxelLength; % n7
                x0+(i-1)*voxelLength, y0+(j)*voxelLength,   z0+voxelLength];% n8
            n_top = zeros(1,4); % Top face node IDs
            % Create/reuse top face nodes
            for k = 1:4
                x = coords_top(k,1); y = coords_top(k,2); z = coords_top(k,3);
                % Check if top node already exists
                idx = find(existingNodes(:,2)==x & existingNodes(:,3)==y &
                      existingNodes(:,4)==z, 1);
                if isempty(idx)
                    % Add new top node
                    n_top(k) = nextNodeID;
                    nodes(end+1,:) = [nextNodeID, x, y, z];
                    nextNodeID = nextNodeID + 1;
                else
                    % Reuse existing top node
                    n_top(k) = existingNodes(idx,1);
                end
            end
            % Assign top face nodes
            n5 = n_top(1); n6 = n_top(2); n7 = n_top(3); n8 = n_top(4);
            % Create cubic element with 8 nodes
            elements(end+1,:) = [nextElementID, n1, n2, n3, n4, n5, n6, n7, n8];
            nextElementID = nextElementID + 1;
        end
    end
end
```

The function *export_voxels_to_inp_with_plates_as_one_part (BW, voxelLength, filename)* takes three input arguments: *BW*, the three-dimensional binary matrix that represents the

structure; *voxelLength*, which specifies the side length of the elemental cube; and *filename*, which denotes the name assigned to the .inp file to be generated.
Once the domain dimensions corresponding to the number of points along the x, y and z directions *([Nx, Ny, Nz] = size(BW))* are extracted, the variable *nodeGrid* is initialized, a three-dimensional grid of size *(Nx+1)×(Ny+1)×(Nz+1)* that will serve as a map to assign each mesh node a unique identifying index. The variables *allNodes* and *allElements* are initialized as empty matrices. *allNodes* will contain, for each node, its identifying index and its three physical coordinates (x, y, z); *allElements*, for each element, will contain its identifying index and the indices of the eight nodes that compose it. The counters *nodeID* and *elementID* are initialized to 1 and are incremented as new nodes and elements are generated.
Since a voxel is created for every grid point with value 1, the variable *offsets* is defined as an 8×3 matrix that contains the displacements of the eight vertex positions of a cube with respect to the local origin of each voxel. These offsets are used to compute the physical coordinates and to assign consistent identifiers to the nodes of each cube in the mesh.
Three nested loops over i, j and k scan the entire binary domain; whenever a point BW(j,i,k) equals 1, i.e., when it belongs to the structure, a cubic voxel is generated. For each voxel, an array *elemNodeIDs* is assembled collecting the eight nodal identifiers. For each cube vertex (denoted by *v*), the absolute indices ii, jj, kk of the node in the global grid are computed. If the node has not yet been considered (that is, *nodeGrid(ii,jj,kk) == 0*), the real spatial coordinates of the node are computed by multiplying the grid coordinates by the voxel length. The node is then appended to *allNodes*, registered in the nodeGrid map and its ID stored in *elemNodeIDs*. If the node already exists, its ID is simply reused. Finally, the cubic element is defined by *elementID* and its eight nodes and is stored in *allElements*.
After completing the voxelization of the spinodoid, the voxelized creation of the upper and lower plates is performed. These plates consist of a single layer of cubic voxels and are generated so as to form a single entity with the structure, sharing nodes and contacting faces.
The physical dimensions of the domain along the three axes (*Lx = Nx × voxelLength, Ly = Ny × voxelLength and Lz = Nz × voxelLength*) are first computed; these values serve to position the plates in 3D space and to determine their extent in the x–y plane. In particular, the coordinates *topZ* and *bottomZ* denote, for both plates, the z-coordinate of their lower face, that is, the plane from which the cube elements of the mesh are generated. To ensure that both plates are in contact with the structure as described, the top plate is generated with its lower face coincident with z = Lz (the height of the spinodoid structure), while the bottom plate starts at z = -voxelLength, so that the cubes (each of height voxelLength) contact the base of the spinodoid structure with their upper face.
The auxiliary function *create_plate_reuse_nodes* is responsible for generating a single voxelized planar plate and for storing its node and element data. Among its arguments it receives the origin vertex coordinates *x0, y0, z0* and the extents in the x–y plane Lx and Ly. In our case the values of x0 and y0 are both equal to zero because the plates originate at the x–y plane origin; the z0 coordinate is chosen as described above.
The nodes and elements of the plate are generated analogously to those of the main structure: for each cube, the eight vertices are computed according to the offsets, and if a node has not yet been considered (i.e., is not already present in the variable *allNodes*), its absolute position is calculated and stored with its unique index. The function returns, for the nodes, the identifying index and the coordinates (x, y, z); for the elements, the identifying index and the indices of the eight nodes composing each element. The function also outputs the updated *nodeID* and *elementID* variables.

The nodes and elements of the plates are finally concatenated to those of the main structure in the variables *allNodes* and *allElements*.
The data saved for nodes and elements are precisely those that must be specified in the .inp file for Abaqus to reconstruct the structure as a mesh of C3D8 elements.
The actual writing of the .inp file then follows: the save path is constructed via *fullfile*(...) and a file is opened with *fopen*. After the header and the declaration of the part, the information on the nodes (*nodeID, x, y, z*) and on the elements *([elementID, elemNodeIDs]*) is written. A solid section is assigned to the entire set and the part is closed.
In the assembly section, an instance of the complete part (spinodoid + plates) must then be defined. Subsequently, two reference points (*topRP_ID* and *bottomRP_ID*) are added, positioned respectively at the center of the top face of the top plate and at the center of the bottom face of the bottom plate. These reference points are added to the nodal listing and included in two node sets named *RP_TOP* and *RP_BOTTOM*.
Two element sets are then created containing the elements of the upper and lower plates (respectively *TOPPLATE_S2_ELEMS* and *BOTTOMPLATE_S1_ELEMS*). From these, the surfaces of interest *TOP_SURFACE* and *BOTTOM_SURFACE* are derived, which, as noted above, are respectively the top face (with outward normal pointing toward the positive z-axis) of the top plate and the bottom face (with outward normal pointing toward the negative z-axis) of the bottom plate.
They are identified using the S2 and S1 commands, which specify, for the plate elements, the top and bottom faces respectively (with respect to the z-axis).

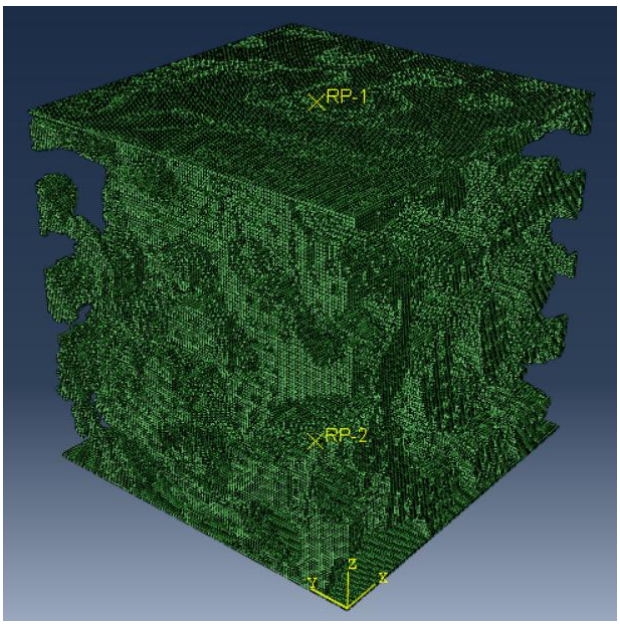


Figure 2.2.11: Assembly with reference points

The next step involves defining the kinematic coupling constraints between each reference point and its corresponding plate surface. These constraints are kinematic in nature, meaning that they enforce the surface to undergo the same rigid-body motion as the reference point.

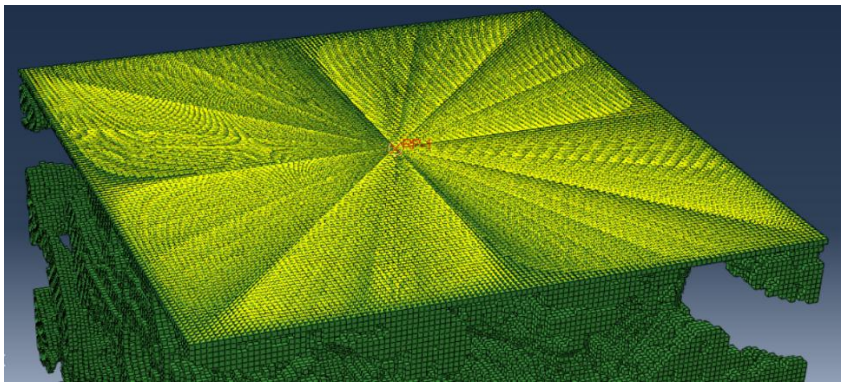


Figure 2.2.12: Kinematic coupling for upper plate (same for lower plate)

After closing the assembly, an elastoplastic material is defined, with a Young's modulus of 1000 MPa and a Poisson's ratio of 0.3. The plastic law assumes a yield stress of 10 MPa and an initial plastic strain value equal to zero.
This is followed by the definition of the analysis step, set as a nonlinear static step with a maximum of 200 increments. The NLGEOM command is set to ON to account for possible nonlinearities arising from large deformations. The minimum and maximum increment values are empirically chosen to ensure the highest possible stability and computational efficiency in completing the analysis.
In the boundary condition section, constraints are applied to the reference points: the upper reference point (and consequently the entire upper plate) is constrained in the x and y degrees of freedom and subjected to an imposed vertical displacement of -0.01 (compression).

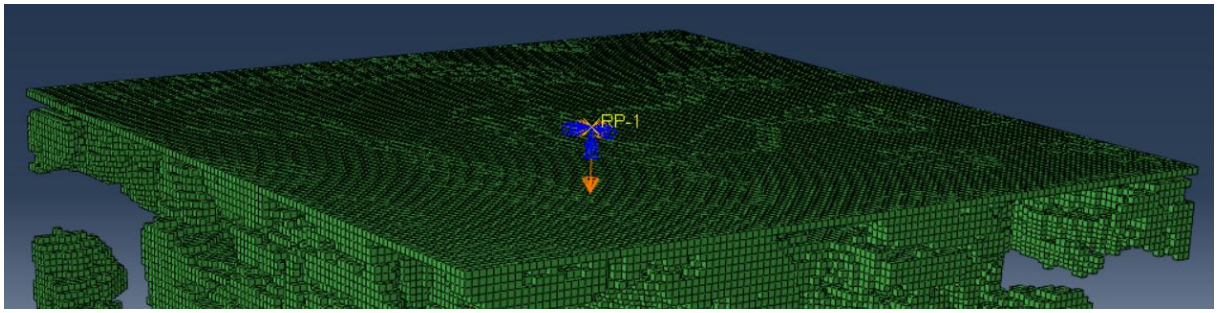


Figure 2.2.13: Boundary conditions, upper plate

The lower reference point (and thus the entire lower plate) is fully restrained using the ENCASTRE constraint.

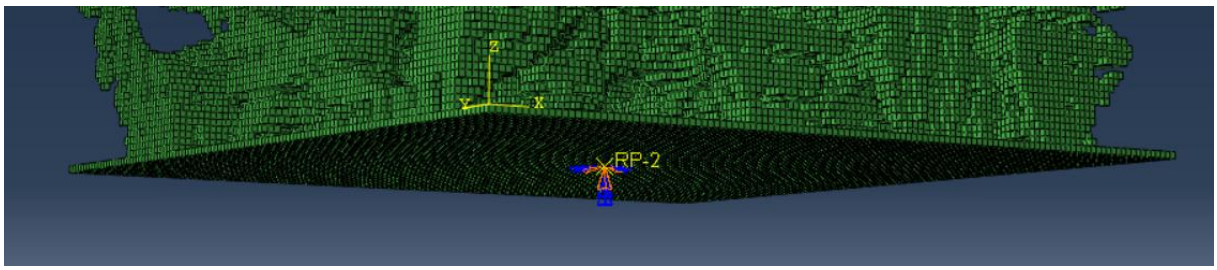


Figure 2.2.14: Boundary conditions, lower plate

Finally, the output options are specified: the predefined field output is activated, and the history output is defined to record the vertical reaction and the vertical displacement of the upper plate (RF3 and U3).
The file is then closed using *fclose(fid).*
The results of the FEM analysis in Abaqus are presented in Section 3.2 – FEM Analysis Results.

2.2.3 Transition to the 2D case

The generation and FEM analysis times of the 3D structures, however, prove to be excessively high to be compatible with the time constraints and computational resources available within the scope of the present thesis work. Hierarchical structures require higher resolution to meaningfully and accurately reproduce their morphological features, which, by definition, occur at smaller scales compared to primary structures and are therefore more affected by the discretization of the binary grid. For these reasons, the study will continue by focusing on two-dimensional hierarchical spinodoid structures, which allow the use of higher resolutions while maintaining a good level of accuracy and a computational load compatible with the temporal and hardware constraints imposed by the thesis framework. It should be emphasized, however, that everything presented hereafter can be similarly applied and generalized to three-dimensional structures.
For the 2D case, the structure generation is carried out in an entirely analogous manner to the 3D case, but the GRF (and consequently the binary domain) is generated in the two-dimensional space as follows:

```
%% 2D SPINODOID GENERATION
clc;
clear;
domain_size = 1;
grid_res = 200;
voxelLength=domain_size/(grid_res-1);
relative_density = 0.75;
x_vec = linspace(0, domain_size, grid_res);
y_vec = linspace(0, domain_size, grid_res);
[x_grid,y_grid]=ndgrid(x_vec,y_vec);
theta1 = 0 * (pi / 180);
theta2 = 10 * (pi / 180);
% Generation of random waves
n_waves = 300;
wavelength = domain_size/4.5;
wavenumber = 2 * pi / wavelength;
v = zeros(n_waves, 2);
phase_shift = zeros(n_waves, 1);
for i = 1:n_waves
    theta = 2 * pi * rand;
    while ~ ( cos(theta)>=cos(theta1) || sin(theta)>= cos(theta2) )
        theta = 2 * pi * rand;
    end
    v(i, :) = [cos(theta), sin(theta)];
    phase_shift(i) = 2 * pi * rand;
end
% Computation of the GRF field
GRF = zeros(grid_res);
for i = 1:length(x_vec)
    for j = 1:length(y_vec)
```

```
        GRF(i, j) = 0;
        for w = 1:n_waves
            GRF(i, j) = GRF(i, j) + cos(wavenumber * v(w, :) * [x_vec(i),
y_vec(j)]' + phase_shift(w));
        end
    end
end
% Normalize the GRF and apply threshold
GRF=sqrt(2/n_waves)*GRF;
GRF = (GRF - mean(GRF(:))) / std(GRF(:));
threshold = sqrt(2) * erfinv(2 * relative_density - 1);
% Apply threshold to get binary domain (solid vs void)
domain_bin = GRF < threshold;
% 2D visualization of the spinodoid (solid vs void)
figure;
contourf(x_vec, y_vec, domain_bin, [0.5 0.5], 'LineColor', 'none');   % Creates a
2D image
colormap([1 1 1; 0 0 1]);    % Colors: white for void, blue for solid
axis equal;
title('2D Spinodoid');
xlabel('x');
ylabel('y');

%% REMOVING ISOLATED PARTS FROM PRIMARY STRUCTURE
% Find all connected components
CC = bwconncomp(domain_bin,4);
% Compute the size of each connected component
numPixels = cellfun(@numel, CC.PixelIdxList);
% Find the index of the largest connected component
[~, idx_max] = max(numPixels);
% Create a new binary domain containing only the main component
domain_bin = false(size(domain_bin));
domain_bin(CC.PixelIdxList{idx_max}) = true;
% 2D visualization of the spinodoid (solid vs void)
figure;
contourf(x_vec, y_vec, domain_bin, [0.5 0.5], 'LineColor', 'none');   % Creates a
2D image
colormap([1 1 1; 0 0 1]);    % Colors: white for void, blue for solid
axis equal;
title('2D Spinodoid without isolated parts');
xlabel('x');
ylabel('y');
% Compute relative density
finalrelativedensity = sum(domain_bin(:)) / numel(domain_bin);
fprintf('Relative density of the first structure is %.2f\n',finalrelativedensity)

%% HIERARCHY
relative_density_2 = 0.8;
x_vec = linspace(0, domain_size, grid_res);
y_vec = linspace(0, domain_size, grid_res);
theta1_2 = 90 * (pi / 180);
theta2_2 = 90 * (pi / 180);
% Generation of random waves
n_waves_2 = 300;
wavelength_2 = domain_size/30;
wavenumber_2 = 2 * pi / wavelength_2;
v_2 = zeros(n_waves_2, 2);
phase_shift_2 = zeros(n_waves_2, 1);
for i = 1:n_waves_2
```

```
    theta = 2 * pi * rand;
    while ~ ( cos(theta)>=cos(theta1_2) || sin(theta)>= cos(theta2_2) )
        theta = 2 * pi * rand;
    end
    v_2(i, :) = [cos(theta), sin(theta)];
    phase_shift_2(i) = 2 * pi * rand;
end
% Computation of the GRF field
GRF_2 = zeros(grid_res);
for i = 1:length(x_vec)
    for j = 1:length(y_vec)
        GRF_2(i, j) = 0;
        for w = 1:n_waves_2
            GRF_2(i, j) = GRF_2(i, j) + cos(wavenumber_2 * v_2(w, :) * [x_vec(i),
y_vec(j)]' + phase_shift_2(w));
        end
    end
end
% Normalize the GRF and apply threshold
GRF_2=sqrt(2/n_waves_2)*GRF_2;
GRF_2 = (GRF_2 - mean(GRF_2(:))) / std(GRF_2(:));
threshold_2 = sqrt(2) * erfinv(2 * relative_density_2 - 1);
% Apply threshold and mask with first structure to obtain hierarchical spinodoid
domain_bin_2 = GRF_2 < threshold_2 & (domain_bin == 1);
% 2D visualization of the hierarchical spinodoid (solid vs void)
figure;
contourf(x_vec, y_vec, domain_bin_2, [0.5 0.5], 'LineColor', 'none');   % Creates
a 2D image
colormap([1 1 1; 0 0 1]);    % Colors: white for void, blue for solid
axis equal;
title('2D Hierarchical Spinodoid');
xlabel('x');
ylabel('y');

%% REMOVING ISOLATED PARTS FROM HIERARCHICAL STRUCTURE
% Find all connected components
CC = bwconncomp(domain_bin_2,4);
% Compute the size of each connected component
numPixels = cellfun(@numel, CC.PixelIdxList);
% Find the index of the largest connected component
[~, idx_max] = max(numPixels);
% Create a new binary domain containing only the main component
domain_bin_2 = false(size(domain_bin_2));
domain_bin_2(CC.PixelIdxList{idx_max}) = true;
% 2D visualization of the hierarchical spinodoid (solid vs void)
figure;
contourf(x_vec, y_vec, domain_bin_2, [0.5 0.5], 'LineColor', 'none');   % Creates
a 2D image
colormap([1 1 1; 0 0 1]);    % Colors: white for void, blue for solid
axis equal;
title('2D Hierarchical Spinodoid without isolated parts');
xlabel('x');
ylabel('y');
% Compute relative density
finalrelativedensity_2 = sum(domain_bin_2(:)) / numel(domain_bin_2);
fprintf('Relative density of the hierarchical structure is
%.2f\n',finalrelativedensity_2)
```

In the 2D case, two limiting angles are defined for both the primary and secondary structures, around the x- and y-axes. Each of these no longer defines a limiting cone as in the 3D case, but rather an angular sector (2D), still defined with the reference axis as its bisector and bidirectional, meaning that it extends in both the positive and negative directions of each axis.
As already seen in the nomenclature of the 3D case, ϑ_1 and ϑ_2 represent the limiting angle values around the x- and y-axes of the primary structure, respectively.
ϑ_{1_2} and ϑ_{2_2} are their counterparts for the secondary structure.
In this case as well, the isolated parts that would not contribute to the mechanical properties of the structure are removed, and the effective relative density is recalculated.
In 2D, the function *bwconncomp* takes a connectivity value of 4. In this case, the default value would be 8, but 4 is used to prevent two consecutive 2D voxels from being considered connected if they are in contact at only a single corner (a similar reasoning was also applied in the case of the 3D structures in Section 2.2.1.2 – Removing detached parts).
An example of an obtainable structure, generated by combining through the ϑ angle values a cubic primary structure with a lamellar secondary one, is shown in the figure below:

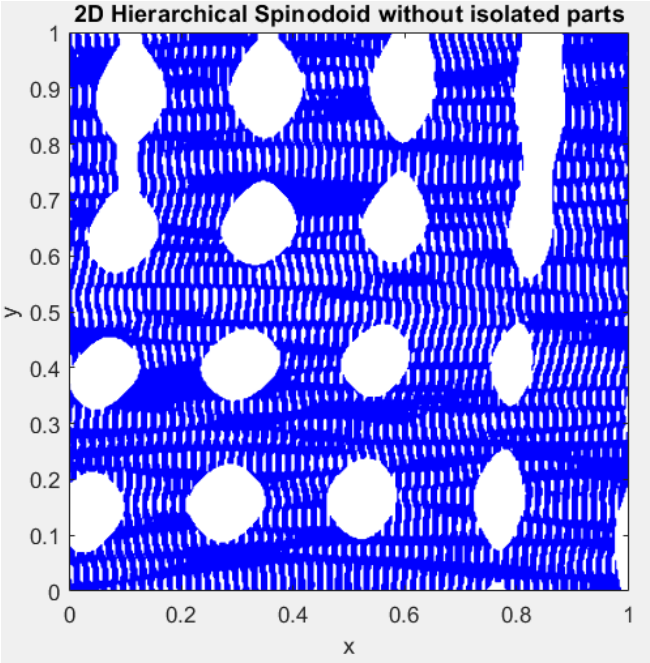


Figure 2.2.15: Example of 2D hierarchical spinodoid ($\boldsymbol{\vartheta_1 = 10}$; $\boldsymbol{\vartheta_2 = 10}$; $\boldsymbol{\vartheta_{1\_1} = 0}$; $\boldsymbol{\vartheta_{1\_2} = 20}$)

Other examples of obtainable structures are shown in Section 3.3 – Examples of Obtainable 2D Hierarchical Structures.

2.2.4 Hierarchical 2D lamellar spinodoids

The continuation of this thesis work will focus on the study of a specific type of hierarchical 2D spinodoids, characterized by both a primary and secondary lamellar structure. These structures will be generated, characterized, and subsequently analyzed through FEM simulations in order to determine their corresponding effective stiffness under uniaxial compression.

Based on these analyses, a dataset will be constructed in which, for each structure, several morphological parameters will be recorded along with the calculated stiffness value. This dataset will serve as the basis for training a neural network, with the aim of learning the correlations between the morphological features of the structures and their corresponding stiffness, thereby enabling the prediction of the stiffness of new structures.
The general workflow of the study is summarized in the flowchart shown in Figure 2.2.1: Overall workflow.

2.2.4.1 New implementation of propagation direction of static wave unit vectors

In the specific case of hierarchical 2D lamellar spinodoid structures, the definition of the unit vector indicating the propagation direction of each wave has been implemented differently from the previously described methodology. The earlier approach did not allow for precise control of the propagation direction of the static waves and, consequently, of the orientation of the generated lamellae, particularly when the limiting angles were large (as the waves could propagate in any direction within the limiting cone).
The new implementation is based on the introduction of a single angle ϑ, ranging from 0° to 180° from the x-axis, which uniquely identifies the propagation direction of the waves. The unit vector associated with each wave is constrained to lie within an angular sector defined by $\vartheta \pm \Delta\vartheta$, where the width of the variation range $\Delta\vartheta$ is set, in this study, to 13° in order to ensure better structural continuity:

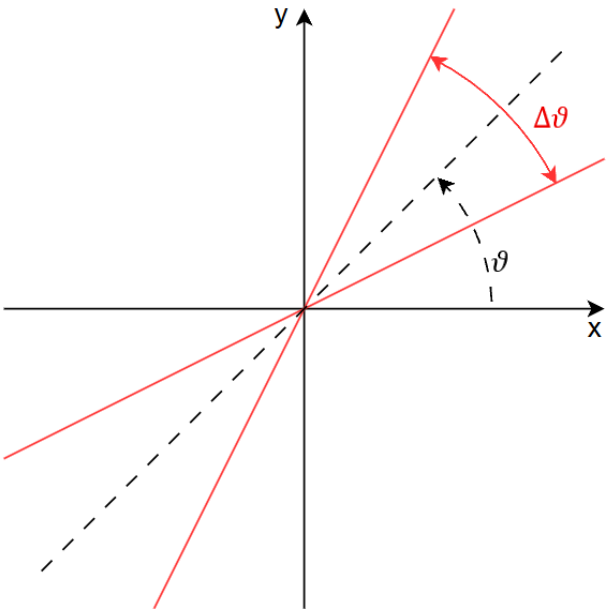


Figure 2.2.16: New definition of the limiting angular sector (Bounded by the Red Lines)

The definition interval [0°, 180°] was chosen because only the direction, and not the propagation sense, is relevant (for example, 30° and 210° indicate the same direction but with opposite orientation, which is redundant for the purposes of this analysis).
In MATLAB, the new definition of the unit vectors is therefore:

```
theta1=40; % random example angle
conelimitangle=13; % angular range Δθ
theta1_botlim=theta1-conelimitangle; % upper angular limit
theta1_uplim=theta1+conelimitangle;  % lower angular limit
% Generation of random waves
```

```
v = zeros(n_waves, 2);
phase_shift = zeros(n_waves, 1);
for i = 1:n_waves
    theta = randi([theta1_botlim,theta1_uplim])*pi/180;
    v(i, :) = [cos(theta), sin(theta)];
end
```

This is analogous for both the construction of the GRF of the primary structure and that of the secondary structure.

2.2.4.2 Detached parts issue in hierarchical lamellar spinodoids

Before proceeding to the description of the complete code, attention is focused on an issue that arose during the creation of the structures: the detachment and separation of the lamellae in the transition from the primary to the secondary structure. To introduce the problem, an illustrative image is presented:

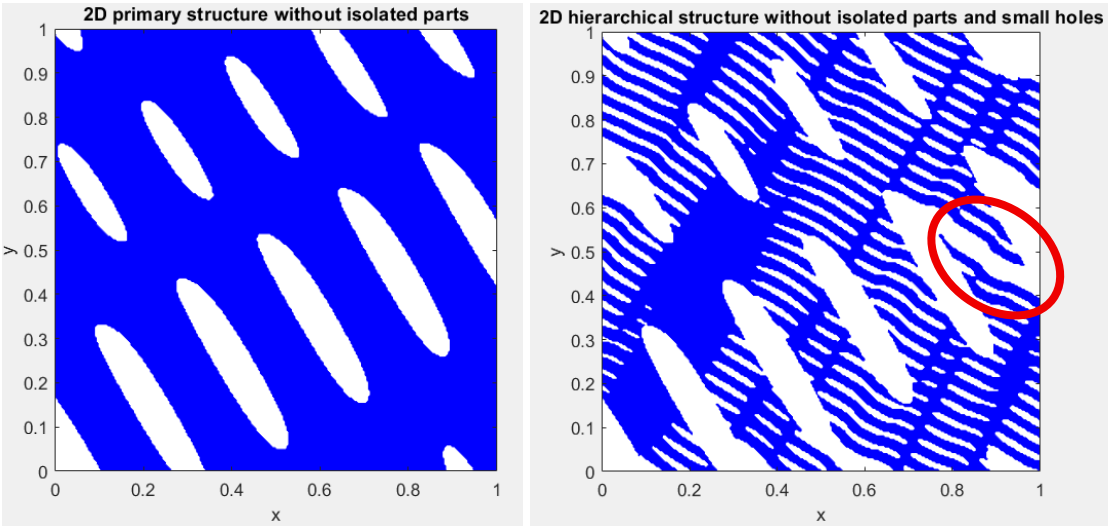


Figure 2.2.17: Secondary lamellae intersect primary lamellae, creating disconnected parts

As can be observed from the previous figure, the voids formed by the alternating lamellae of the secondary structure generated a region detached from the main body, which was subsequently removed as it was identified by the code as a detached part. A more detailed analysis of the occurrence of this effect and its handling is provided in Section 4.2.1 – Disconnected parts issue and influence of generation parameters.

2.2.4.3 Code overview

The following code encompasses the generation and morphological characterization of the structures, the writing of the .inp file, the execution of the finite element analysis using the .inp file via a batch file without relying on the Abaqus GUI, the calculation of stiffness from the obtained results, and the saving of the data necessary for the creation of the dataset.
Most of the procedures illustrated in the MATLAB codes related to 3D structures are applied in an entirely analogous way to the analysis of 2D structures. Therefore, they will not be

re-explained in detail, in order to avoid redundancies in the description of the developed code.
Although it is possible to generalize all the subsequent work to hierarchical lamellar structures with any lamellae orientation, from this point onward the study will focus on structures in which the primary and secondary theta angles are randomly chosen among four possible values ([0°, 45°, 90°, 135°]). This discretization in the input features will also reflect on the output data, whose distribution will be analyzed in detail later, in Section 3.4.2 – Dataset characterization. The relationship between the imposed limit angle theta and the orientation of the lamellae in the resulting structure is discussed in detail in Section 4.2.2 – Correlation between static wave propagation angle ϑ and lamellae orientation.

2.2.4.4 Primary structure generation and characterization

After these preliminaries, the complete code is described, starting with the definition of the number of structures that will form the dataset and with the initialization of the vector containing the possible orientations of the primary and secondary structure lamellae. Each structure will have a combination of these two angles randomly chosen from the vector *possiblethetas* using the *randi* function. The code then proceeds to create the primary structure and remove the detached parts, as previously shown:

```
%% FULL 2D SCRIPT
clc
clear
rng('shuffle');
%% DATASET CREATION
dataset_length = 12000;
possiblethetas = [0, 45, 90, 135]; % Only 4 possible orientations of the lamellae
                                      are considered
choosenthetas = zeros(dataset_length, 2); % Storing lamella orientations for both
                                             primary and secondary structures
for d = 1:dataset_length
    choosenthetas(d, :) = possiblethetas(randi(length(possiblethetas), 1, 2));
    domain_size = 1;
    grid_res = 170;
    voxelLength=domain_size/(grid_res-1);
    relative_density = 0.87;
    relative_density_percent=relative_density*100;
    x_vec = linspace(0, domain_size, grid_res);
    y_vec = linspace(0, domain_size, grid_res);
    [x_grid,y_grid]=ndgrid(x_vec,y_vec);
    theta1=choosenthetas(d,1);
    conelimitangle=13; % angular range Δθ
    theta1_botlim=theta1-conelimitangle; % upper angular limit
    theta1_uplim=theta1+conelimitangle;  % lower angular limit
    % Generation of random waves
    n_waves = 300;
    ratio=4.1;
    wavelength = domain_size/ratio;
    wavenumber = 2 * pi / wavelength;
    v = zeros(n_waves, 2);
    phase_shift = zeros(n_waves, 1);
    for i = 1:n_waves
        theta = randi([theta1_botlim,theta1_uplim])*pi/180;
        v(i, :) = [cos(theta), sin(theta)];
```

```
    phase_shift(i) = 2 * pi * rand;
end
% Computation of the GRF field
GRF = zeros(grid_res);
for i = 1:length(x_vec)
    for j = 1:length(y_vec)
        for w = 1:n_waves
            GRF(i, j) = GRF(i, j) + cos(wavenumber * v(w, :) * [x_vec(i),
                        y_vec(j)]' + phase_shift(w));
        end
    end
end
% Computation of the threshold
GRF=sqrt(2/n_waves)*GRF;
GRF = (GRF - mean(GRF(:))) / std(GRF(:));
threshold = sqrt(2) * erfinv(2 * relative_density - 1);
% Application of the threshold to obtain solid and void domains
domain_bin = GRF < threshold;

%% REMOVING ISOLATED PARTS FROM PRIMARY STRUCTURE
CC = bwconncomp(domain_bin,4);
numPixels = cellfun(@numel, CC.PixelIdxList);
[~, idx_max] = max(numPixels);
domain_bin = false(size(domain_bin));
domain_bin(CC.PixelIdxList{idx_max}) = true;
%computing relative density
finalrelativedensity = sum(domain_bin(:)) / numel(domain_bin);
fprintf('Relative density of the %d primary structure is
%.2f\n',d,finalrelativedensity)
```

Thinning is applied to obtain the skeleton of the structure:

```
%% THINNING ALGORITHM ON PRIMARY STRUCTURE
skel = bwskel(domain_bin);  % obtain 2D skeleton of the binary domain
[yIdx, xIdx] = find(skel);  % find row and column indices of skeleton pixels
% Map pixel indices to physical coordinates using grid vectors
skeletonpoints = [x_grid(sub2ind(size(x_grid), xIdx, yIdx)), ...
                  y_grid(sub2ind(size(y_grid), xIdx, yIdx))];
% Define maximum allowed distance between connected nodes (diagonal of a
  pixel)
maxDist = 1.415 * (1 / (grid_res - 1));
connections = [];
% Build graph connections by linking points within maxDist
for i = 1:size(skeletonpoints, 1)
    for j = i+1:size(skeletonpoints, 1)
        if norm(skeletonpoints(i,:) - skeletonpoints(j,:)) <= maxDist
            connections = [connections; i, j];
        end
    end
end
% Create graph object from connections
G = graph(connections(:,1), connections(:,2));
```

Closed triangles in the skeleton are removed, which in 2D are right-angled triangles with two sides of length equal to *voxelLength* (*delta*) and a hypotenuse equal to $\sqrt{2} \cdot delta$:

```
%% FINDING AND REMOVING CLOSED LOOP TRIANGLES IN PRIMARY STRUCTURE SKELETON
% Obtain adjacency matrix of the graph
A = adjacency(G);
% Find all triangles (3 nodes all mutually connected)
triangles = [];
nNodes = numnodes(G);
for i = 1:nNodes-2
    for j = i+1:nNodes-1
        if A(i,j)
            for k = j+1:nNodes
                if A(i,k) && A(j,k)
                    triangles = [triangles; i, j, k];
                end
            end
        end
    end
end
edgesToRemove = [];
% Loop through each triangle to detect closed right triangles and remove
  diagonal edges
for k = 1:size(triangles, 1)
    tri = triangles(k, :);
    pts = skeletonpoints(tri, :);
    % Calculate the distances between the 3 nodes of the triangle
    d12 = norm(pts(1,:) - pts(2,:));
    d23 = norm(pts(2,:) - pts(3,:));
    d31 = norm(pts(3,:) - pts(1,:));
    dists = sort([d12, d23, d31]);  % sort distances ascending
    % Define expected pixel size and tolerance
    delta = voxelLength;
    tol = delta * 0.2;  % 20% tolerance
    % Check if triangle is a closed right triangle (two sides ~delta,
      hypotenuse ~sqrt(2)*delta)
    if abs(dists(1) - delta) < tol && abs(dists(2) - delta) < tol &&
       abs(dists(3) - sqrt(2)*delta) < tol
        % Identify the longest edge (hypotenuse) to remove
        if abs(dists(3) - d12) < 1e-12
            edgeToRemove = sort([tri(1), tri(2)]);
        elseif abs(dists(3) - d23) < 1e-12
            edgeToRemove = sort([tri(2), tri(3)]);
        else
            edgeToRemove = sort([tri(3), tri(1)]);
        end
        % Append edge to removal list if not already included
        if isempty(edgesToRemove) || ~ismember(edgeToRemove, edgesToRemove,
           'rows')
            edgesToRemove = [edgesToRemove; edgeToRemove];
        end
    end
end
% Remove the identified diagonal edges that close right triangles
for k = 1:size(edgesToRemove,1)
    edgeNodes = edgesToRemove(k,:);
    edgeID = findedge(G, edgeNodes(1), edgeNodes(2));
    if edgeID > 0
        G = rmedge(G, edgeID);
    end
end
```

The degree of the nodes in the skeleton is calculated:

```
%% COMPUTING NODE DEGREE FOR PRIMARY STRUCTURE
NodeDegree = degree(G);
maxNodeDegree = max(NodeDegree);
nodesdegree_vec = zeros(maxNodeDegree, 1);
for i = 1:length(NodeDegree)
    nodesdegree_vec(NodeDegree(i)) = nodesdegree_vec(NodeDegree(i)) + 1;
end
```

All branch chains of the skeleton and the nodes composing them are identified; their length is then calculated:

```
%% BRANCH SEGMENT ANALYSIS FOR PRIMARY STRUCTURE
% Calculate lengths of branch chain connecting branch nodes (degree >= 3)
branchNodes = find(NodeDegree >= 3); % Identify branch nodes (nodes with
                                        degree >= 3)
chainLengths = [];  % Array to store chain lengths
chainNodes = {};    % Cell object to store the nodes of each chain
visitedPairs = false(numnodes(G), numnodes(G));  % squared logical matrix of
numnodes(G)xnumnodes(G) dimension, to track if a pair of nodes has already
been processed
for i = 1:length(branchNodes)
    startNode = branchNodes(i);
    nbrs = neighbors(G, startNode); % returns a vector of nodes directly
                                      connected to startNode in the graph G
    for j = 1:length(nbrs)
        if visitedPairs(startNode, nbrs(j)) || visitedPairs(nbrs(j),startNode)
            continue;  % if the segment has already been processed
        end
        % if the selected neighbor is a branchnode a branch chain is
        % generated
        if NodeDegree(nbrs(j)) >= 3
            segLength = norm(skeletonpoints(startNode, :) -
                        skeletonpoints(nbrs(j), :));
            chainLengths(end+1) = segLength; % add the segment lenght in
                                               chainLengths vector
            visitedPairs(startNode, nbrs(j)) = true; % mark connection of
                                                       visited points
            visitedPairs(nbrs(j), startNode) = true;
            chainNodes{end+1} = [startNode, nbrs(j)];  % add in chainNodes{}
            cell the vector containing index of nodes of the branch segment
        else
            % if the first neighbor isn't a branch nodes continue on the
            % chain
            currentNode = nbrs(j);
            prevNode = startNode;
            segLength = norm(skeletonpoints(startNode, :) -
                        skeletonpoints(currentNode, :)); % update the chain
                                                           length
            segmentNodesChain = [startNode, currentNode]; % adding the two
            first nodes of the chain
            while true
                visitedPairs(prevNode, currentNode) = true;
                visitedPairs(currentNode, prevNode) = true;
```

```
                nbrs2 = neighbors(G, currentNode); % Returns a vector of nodes
                directly connected to currentNode in the graph G
                nbrs2(nbrs2 == prevNode) = [];  % Removing prevNode from the
                neighbors vector to not go back in the chain
                % Filter already visited connections
                unvisited = nbrs2(~visitedPairs(currentNode, nbrs2) &
                            ~visitedPairs(nbrs2, currentNode));
                if isempty(unvisited)
                    break;  % No unvisited neighbor, end chain
                end
                nextNode = unvisited(1);  % Pick the first unvisited neighbor
                segLength = segLength + norm(skeletonpoints(currentNode, :) -
                            skeletonpoints(nextNode, :)); % update the chain
                                                          length
                segmentNodesChain = [segmentNodesChain, nextNode]; % adding
                nextNode to the chain
                if NodeDegree(nextNode) >= 3
                    visitedPairs(currentNode, nextNode) = true;
                    visitedPairs(nextNode, currentNode) = true;
                    break; % if nextNode is a branch node a branch chain is
                             generated
                else
                    prevNode = currentNode; %update prevNode
                    currentNode = nextNode; %update currentNode
                end
            end
            chainLengths(end+1) = segLength; % adding the final chain length
                                             to the chainLengths vector
            chainNodes{end+1} = segmentNodesChain;  % add in chiainNodes{}
            cell the vector containing nodes indexes of the final branch chain
        end
    end
end
```

The orientations of the primary lamellae and their dispersion are analyzed, calculating their standard deviation:

```
%% COMPUTING ANGULAR DISPERSION IN PRIMARY STRUCTURE LAMELLAE'S ORIENTATION
window_size = grid_res*0.025; % number of nodes used to compute direction
all_angles = []; % vector where all angles will be stored
for c = 1:length(chainNodes)
    chain = chainNodes{c}; % node indices in the chain
    coords = skeletonpoints(chain, :); % coordinates (Nx3)
    % Skip if the chain is too short
    if size(coords,1) < window_size
        continue
    end
    for i = 1:(size(coords,1) - window_size + 1)
        p1 = coords(i, :);                      % starting point
        p2 = coords(i + window_size - 1, :);    % ending point of the window
        vec = p2 - p1; % direction vector
        % Compute direction in the XY plane
        angle = atan2(vec(2), vec(1)); % radians, range [-pi, pi]
        angle = mod(angle, pi);  % wrap angle into range [0, pi)
        all_angles(end+1) = angle;
    end
end
% Convert to degrees for easier interpretation
```

```
all_angles_deg = rad2deg(all_angles);
angular_dispersion_deg = std(all_angles_deg);
```

The study of the structure's cross-section along the branches is then performed, this time comprehensively on all chains, and not only on a few sampled ones, as shown in the 3D case. These data, particularly for the hierarchical structure, constitute a fundamental element of the descriptive features of each spinodoid, used during the construction of the dataset.

```
%% THICKNESS ANALYSIS ON PRIMARY STRUCTURE
distance_map = bwdist(~domain_bin); % Distance from background
chainnodes_coords = cell(1, numel(chainNodes));
for i = 1:numel(chainNodes)
    chain = chainNodes{i};
    coords = skeletonpoints(chain, :);  % Already in physical coordinates
    chainnodes_coords{i} = coords;
end
% Compute local radius for each point along the chains
chain_radii = cell(1, numel(chainNodes));
segment_subs = cell(1, numel(chainNodes));
for i = 1:numel(chainNodes)
    points_indices = round(chainnodes_coords{i} ./ domain_size .* (grid_res -
                     1)) + 1;
    segment_subs{i} = points_indices;
    x = points_indices(:,1);
    y = points_indices(:,2);
    idx_linear = sub2ind(size(domain_bin), y, x);
    chain_radii{i} = distance_map(idx_linear);
end
% Concatenate all radii into a single vector
all_radii = [];
for i = 1:numel(chain_radii)
    all_radii = [all_radii; double(chain_radii{i})];
end
% Compute mean
mean_radius_primary = mean(all_radii);
% Display the results
fprintf('Mean radius for the %d primary structure: %.2f\n',d,
         mean_radius_primary);
```

2.2.4.5 Hierarchical structure generation and characterization

The implementation of the code for generating hierarchical 2D lamellar structures follows the same principle used in the 3D case, as described in Section 2.2.1.7 – Hierarchical structures. It obviously differs in that it operates in a two-dimensional space, as already discussed in Section 2.2.3 – Transition to the 2D case, and in the new definition of the unit vectors of the static waves, explained in Section 2.2.4.1 – New implementation of propagation direction of static wave unit vectors. Moreover, regarding the subsequent characterization, the steps are completely analogous to those used for the primary structure and are therefore not repeated here.
However, in addition to what has just been shown for the primary structure, immediately after the removal of the detached parts, the secondary structure is further processed to

reduce the computational load without significantly compromising the overall accuracy of the method.

2.2.4.5.1 Computational Load Reduction by Removing Small Cavities from the Hierarchical Structure

As can be observed from the figures below (Figure 2.2.18: Example of small voids in the structure), the structures exhibit small voids caused by certain points with a value of 0 in the binary domain. Those consisting of a single grid point with a false value take the shape of a rhombus (the false point can be imagined at the center of the cavity, while the contour is defined by the four adjacent true points); other, larger voids are composed of a greater number of grid points with false values and take on various shapes:

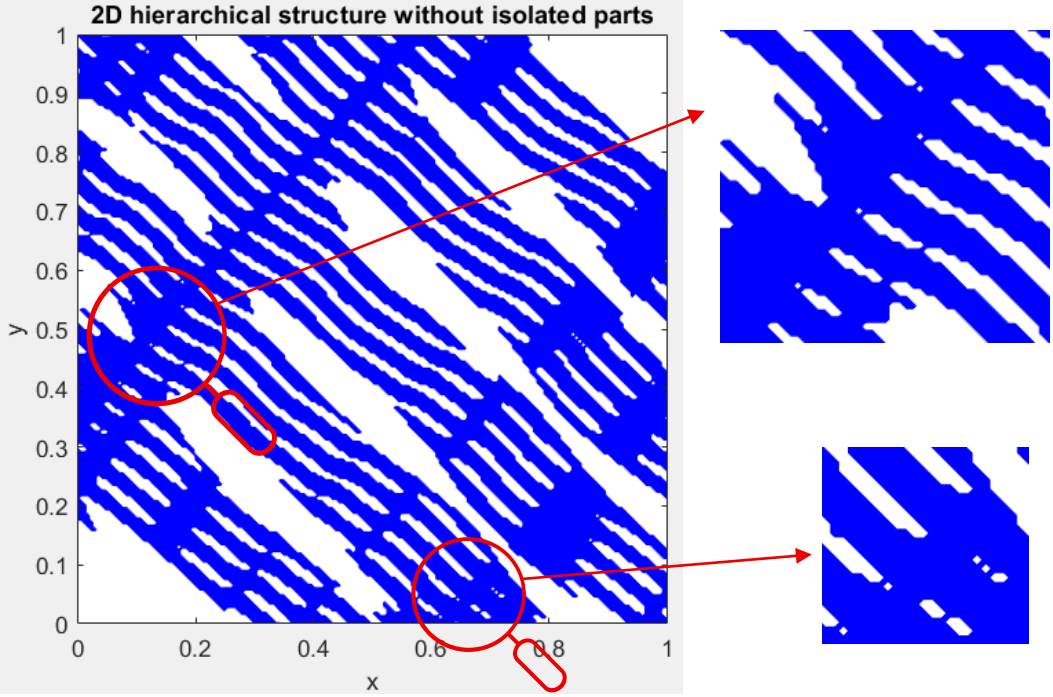


Figure 2.2.18: Example of small voids in the structure

These small cavities, which are of a smaller scale than the voids resulting from the alternation of the lamellae, complicate the subsequent stages of voxelization, .inp file generation, and FEM analysis. Their presence does not significantly affect the computation of the effective stiffness if the maximum removable size is properly controlled, as shown in the Section 4.2.3 – Influence of the Size of Removed Voids on Effective Stiffness. Therefore, a specific "filling" procedure was implemented to eliminate these features, but only if their size is below a predetermined threshold:

```
%% REMOVING SMALL HOLES FROM HIERARCHICAL STRUCTURE
filled = imfill(domain_bin_2, 'holes'); % Fill all holes inside the binary
                                          structure
holes = filled & ~domain_bin_2;     % Extract only the holes by subtracting the
                                      original structure
CC = bwconncomp(holes, 4);          % Find connected components (holes) with
                                      4-connectivity
```

```
stats = regionprops(CC, 'Area');   % Measure the area of each hole
idxSmall = find([stats.Area] <= grid_res*0.025);  % Identify holes with an
area smaller or equal to a certain number of points
smallHolesMask = ismember(labelmatrix(CC), idxSmall);  % Create a mask of the
                                                         small holes
domain_bin_2(smallHolesMask) = 1;   % Fill the identified small holes in the
                                      binary structure
```

The procedure begins with the use of the imfill function, which generates a new binary matrix in which all internal voids within the structure are filled, including the larger ones resulting from the alternation of lamellae and cavities. Subsequently, through a Boolean operation, a "negative" matrix of the original structure is obtained, in which the holes correspond to true values. These regions are then analyzed using the bwconncomp function (whose operation has already been described previously), while the regionprops function calculates the area of each hole.
The cavities with very small areas are then selected, and their indices are stored in the variable *idxSmall*. All holes consisting of a number of points up to 2.5% of the grid's linear resolution are selected (a value chosen based on the results shown in the Section 4.2.3 – Influence of the Size of Removed Voids on Effective Stiffness), which, as demonstrated, prevents significant deviations in the effective stiffness from the original value and/or excessive morphological alterations of the structure.
At this point, the binary matrix *smallHolesMask* is created, in which the points corresponding to the holes to be removed are assigned a value of 1.
Finally, the original matrix is updated by filling the holes at the positions indicated by *smallHolesMask*.
The final result, compared to the original structure, is shown in the figure below:

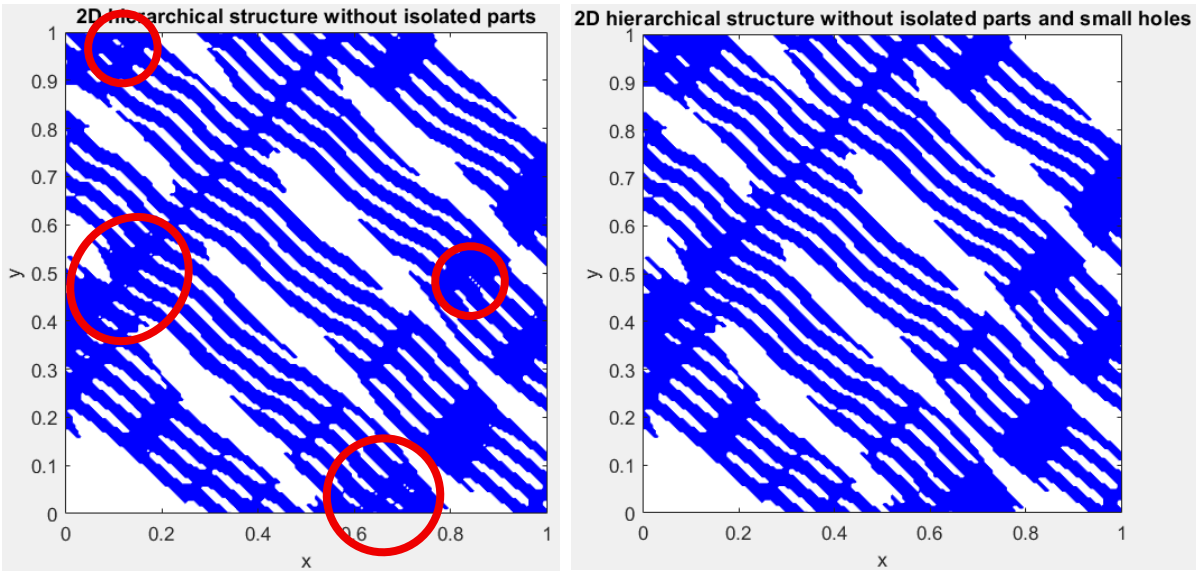


Figure 2.2.19: Structure before and after removal of small voids

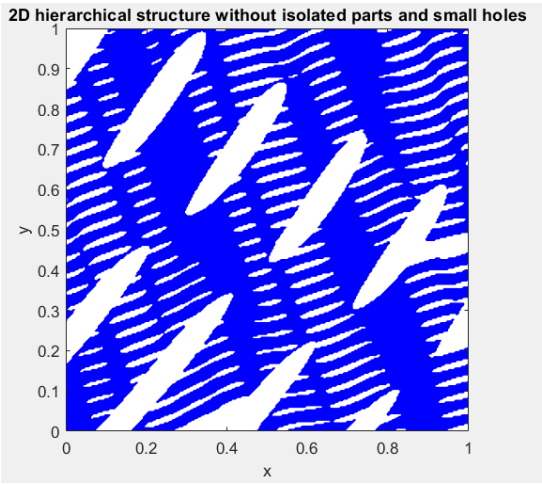
2.2.4.6 Misalignment index between primary and hierarchical structure

Each spinodoid, as previously described, was characterized in terms of lamellae orientation, both for the primary and secondary structures. The obtained data were subsequently processed to provide a clearer representation of the lamellae orientation distribution and

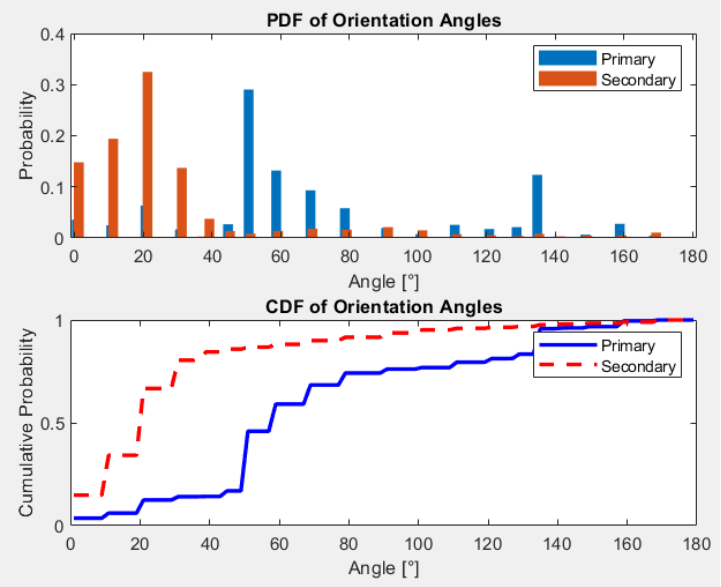
to extract a disorder index from this information. Specifically, probability density function (PDF) curves were derived for the lamellae orientations previously computed for both the primary and secondary structures, enabling direct comparison. The overlap index was then calculated, defined as the common area enclosed by the two PDF curves of lamellae orientations for the primary and secondary structures. This index, ranging from 0 to 1, quantifies the degree of coincidence between the two distributions: values close to 1 indicate a highly aligned orientation, while low values reveal a marked misalignment. For completeness, the corresponding cumulative distribution function (CDF) was also calculated for each PDF.

```
%% COMPUTING MISALIGNMENT INDEXES BETWEEN PRIMARY AND SECONDARY STRUCTURE
bin_edges = 0:2:180; % Define the edges of the bins (0° to 180° with 2° steps)
% Compute bin centers for plotting or integration
bin_centers = bin_edges(1:end-1) + diff(bin_edges)/2;
% === NORMALIZED HISTOGRAMS (PDF) ===
% PDF of lamellae orientation for primary structure
pdf1 = histcounts(all_angles_deg, bin_edges, 'Normalization', 'probability');
% PDF of lamellae orientation for hierarchical structure
pdf2 = histcounts(all_angles_deg_2, bin_edges, 'Normalization','probability');
% === CUMULATIVE DISTRIBUTION FUNCTIONS (CDF) ===
cdf1 = cumsum(pdf1);  % Compute the cumulative distribution function for
                      primary structure
cdf2 = cumsum(pdf2);  % Compute the cumulative distribution function for
                      hierarchical structure
% === OVERLAP INDEX ===
overlap = sum(min(pdf1, pdf2)); % Compute the common area under both PDFs
```

In the case, for example, of a structure where ϑ_1 = 120° and ϑ_2 = 160°, the following results are obtained:



Overlap index = 0.290



The primary lamellae are mainly oriented between 50° and 60°, whereas those of the secondary structure are oriented around 20°. The alignment between the two is only partial, and the overlap index is slightly below 30%.
It should be recalled that the detailed explanation of the relationship between the imposed theta angle value and the resulting lamellae orientation is provided in Section 4.2.2 –

Correlation between static wave propagation angle ϑ and lamellae orientation. A more detailed analysis of the overlap index, including others significant cases, is presented in Section 3.4.1 – Examples of Misalignment Analysis based on the Overlap Index.

2.2.4.7 FEM analysis (2D)

The code then continues with the generation of the .inp file, which produces the complete model ready for FEM analysis. The code is contained within the function *export2D_voxel_structure_with_plates*:

```
%% INP FILE GENERATION
filename_inp = sprintf('%dres%drho%.0fthet%.0frho%.0fthet%.0f.inp',
d,grid_res,relative_density_percent, theta1, relative_density_percent_2,
theta1_2);
export2D_voxel_structure_with_plates(domain_bin_2, filename_inp,voxelLength)
```

To avoid making the main code excessively long, the function *export2D_voxel_structure_with_plates* is defined separately and then called:

```
function export2D_voxel_structure_with_plates(BW, filename, voxelLength)
    [Nx, Ny] = size(BW);
    nodeMap = containers.Map;  % Map to store unique nodes
    nodeList = [];             % List of all nodes
    elemList = [];             % List of all elements
    nodeID = 1;                % Node counter
    elemID = 1;                % Element counter
    % Function to add a voxel and its corner nodes
    function addVoxel(i, j)
        % Coordinates of the 4 nodes (corners of the voxel)
        corners = [ (j-1)*voxelLength, (i-1)*voxelLength;
                     j   *voxelLength, (i-1)*voxelLength;
                     j   *voxelLength,  i   *voxelLength;
                    (j-1)*voxelLength,  i   *voxelLength ];
        nodeIndices = zeros(1,4);
        for k = 1:4
            % Key to identify each node uniquely by coordinates
            key = sprintf('%.6f_%.6f', corners(k,1), corners(k,2));
            if ~isKey(nodeMap, key)
                % If node does not exist, add it to nodeMap and nodeList
                nodeMap(key) = nodeID;
                nodeList(nodeID,:) = [corners(k,1), corners(k,2), 0];
                nodeIndices(k) = nodeID;
                nodeID = nodeID + 1;
            else
                % If node already exists, use existing ID
                nodeIndices(k) = nodeMap(key);
            end
        end
        % Add new element defined by its 4 nodes
        elemList(elemID,:) = nodeIndices;
        elemID = elemID + 1;
    end

    % Create the central structure (internal voxels)
```

```
for i = 1:Nx
    for j = 1:Ny
        if BW(i,j) == 1
            addVoxel(i, j);
        end
    end
end
% Create bottom plate (row 0) and top plate (row Nx+1)
for j = 1:Ny
    addVoxel(0, j);       % bottom plate
    addVoxel(Nx+1, j);    % top plate
end
% Find the lowest and highest nodes (min and max y)
ymin = min(nodeList(:,2));
ymax = max(nodeList(:,2));
tol = 1e-6;
nodes_bottom = find(abs(nodeList(:,2) - ymin) < tol);
nodes_top = find(abs(nodeList(:,2) - ymax) < tol);
% Find central reference node in bottom and top plates
[~, i_bot] = min(abs(nodeList(nodes_bottom,1) -
             mean(nodeList(nodes_bottom,1))));
[~, i_top] = min(abs(nodeList(nodes_top,1) - mean(nodeList(nodes_top,1))));
node_bot = nodes_bottom(i_bot);
node_top = nodes_top(i_top);
% Find elements connected to top and bottom nodes (for ELSETs)
elem_top = find(any(ismember(elemList, nodes_top), 2));
elem_bottom = find(any(ismember(elemList, nodes_bottom), 2));
if isempty(elem_top)
    error('Elset TOP is empty! Check structure definition.');
end
if isempty(elem_bottom)
    error('Elset BOTTOM is empty! Check structure definition.');
end
% Auxiliary function to write ELSET (max 16 elements per line)
function write_elset(fid, name, elems)
    fprintf(fid, '*Elset, elset=%s, instance=spinodoid\n', name);
    maxPerLine = 16;
    count = 0;
    for i = 1:length(elems)
        fprintf(fid, '%d', elems(i));
        count = count + 1;
        if i < length(elems)
            fprintf(fid, ',');
        end
        if count == maxPerLine && i < length(elems)
            fprintf(fid, '\n');
            count = 0;
        end
    end
    fprintf(fid, '\n');
end
% Auxiliary function to write NSET (max 16 nodes per line)
function write_nset(fid, name, nodes)
    fprintf(fid, '*Nset, nset=%s, internal, instance=spinodoid\n', name);
    maxPerLine = 16;
    count = 0;
    for i = 1:length(nodes)
        fprintf(fid, '%d', nodes(i));
        count = count + 1;
```

```
        if i < length(nodes)
            fprintf(fid, ', ');
        end
        if count == maxPerLine && i < length(nodes)
            fprintf(fid, '\n');
            count = 0;
        end
    end
    fprintf(fid, '\n');
end
% Assign unique IDs for Reference Points to avoid overlap
RP_top_ID = size(nodeList,1) + 1;
RP_bottom_ID = size(nodeList,1) + 2;

%% Write .inp file for Abaqus
filepath = fullfile('C:\Users\greco\Politecnico Di Torino Studenti
Dropbox\Carlo Alberto Greco\Erasmus NTNU master thesis\.inp files', filename);
fid = fopen(filepath, 'w');
if fid == -1
    error('Unable to open file for writing.');
end
% Part definition
fprintf(fid, '** 2D Voxel Structure with Top and Bottom Plates\n');
fprintf(fid, '*Part, name=spinodoid\n');
% Nodes
fprintf(fid, '*Node\n');
for i = 1:size(nodeList,1)
    fprintf(fid, '%d, %g, %g, %g\n', i, nodeList(i,1), nodeList(i,2),
            nodeList(i,3));
end
% Elements
fprintf(fid, '*Element, type=CPE4\n');
for i = 1:size(elemList,1)
    fprintf(fid, '%d, %d, %d, %d, %d\n', i, elemList(i,1), elemList(i,2),
    elemList(i,3), elemList(i,4));
end
% All elements ELSET
fprintf(fid, '*Elset, elset=ALL_ELEMS, generate\n');
fprintf(fid, '1, %d, 1\n', size(elemList,1));
% Section assignment
fprintf(fid, '*Solid Section, elset=ALL_ELEMS, material=Material-1\n');
fprintf(fid, '*End Part\n');
% Assembly
fprintf(fid, '*Assembly, name=Assembly\n');
fprintf(fid, '*Instance, name=spinodoid, part=spinodoid\n');
fprintf(fid, '*End Instance\n');
% Reference Point (RP) nodes with unique IDs
fprintf(fid, '*Node\n');
fprintf(fid, '%d, %g, %g, %g\n', RP_top_ID, nodeList(node_top,1),
nodeList(node_top,2), nodeList(node_top,3));
fprintf(fid, '%d, %g, %g, %g\n', RP_bottom_ID, nodeList(node_bot,1),
nodeList(node_bot,2), nodeList(node_bot,3));
% NSETs for RPs
fprintf(fid, '*Nset, nset=RP_top\n');
fprintf(fid, '%d,\n', RP_top_ID);
fprintf(fid, '*Nset, nset=RP_bottom\n');
fprintf(fid, '%d,\n', RP_bottom_ID);
% NSETs for plate top and bottom nodes
write_nset(fid, 'Plate_Up_TopNodes', nodes_top);
```

```
    write_nset(fid, 'Plate_Bottom_BottomNodes', nodes_bottom);
    % ELSETs for plate top and bottom elements
    write_elset(fid, 'TOPPLATE_ELEMS', elem_top);
    write_elset(fid, 'BOTTOMPLATE_ELEMS', elem_bottom);
    % Surfaces for loading/constraints (using correct element sides)
    fprintf(fid, '*Surface, type=ELEMENT, name=TOP_SURFACE, internal\n');
    fprintf(fid, 'TOPPLATE_ELEMS, S2\n');  % top side
    fprintf(fid, '*Surface, type=ELEMENT, name=BOTTOM_SURFACE, internal\n');
    fprintf(fid, 'BOTTOMPLATE_ELEMS, S4\n');  % bottom side
    % Kinematic couplings between surfaces and reference points
    fprintf(fid, '** Constraint: Constraint-1\n');
    fprintf(fid, '*Coupling, constraint name=Constraint-1, ref node=RP_top,
    surface=TOP_SURFACE\n');
    fprintf(fid, '*Kinematic\n');
    fprintf(fid, '** Constraint: Constraint-2\n');
    fprintf(fid, '*Coupling, constraint name=Constraint-2, ref node=RP_bottom,
    surface=BOTTOM_SURFACE\n');
    fprintf(fid, '*Kinematic\n');
    fprintf(fid, '*End Assembly\n');
    % Material definition
    fprintf(fid, '*Material, name=Material-1\n');
    fprintf(fid, '*Elastic\n');
    fprintf(fid, '1000., 0.3\n');
    fprintf(fid, '*Plastic\n');
    fprintf(fid, '10., 0.\n');
    % Step definition
    fprintf(fid, '*Step, name=Step-1, nlgeom=YES, inc=200\n');
    fprintf(fid, '*Static\n');
    fprintf(fid, '0.01, 1., 1e-05, 0.05\n');
    % Boundary conditions
    fprintf(fid, '*Boundary\n');
    fprintf(fid, 'RP_bottom, ENCASTRE\n');
    fprintf(fid, '*Boundary\n');
    fprintf(fid, 'RP_top, 1, 1\n');
    fprintf(fid, 'RP_top, 2, 2, -0.01\n');
    fprintf(fid, 'RP_top, 6, 6\n');
    % Output requests
    fprintf(fid, '*Restart, write, frequency=0\n');
    fprintf(fid, '*Output, field, variable=PRESELECT\n');
    fprintf(fid, '*Output, history\n');
    fprintf(fid, '*Node Output, nset=RP_top\n');
    fprintf(fid, 'RF2, U2\n');
    fprintf(fid, '*End Step\n');
    fclose(fid);
end
```

The function performs the export of a two-dimensional voxel structure, complete with top and bottom plates, into an .inp file compatible with Abaqus, starting from the binary matrix representing the structure's geometry. In the 2D case, the voxelized structure and plates are translated into Abaqus using a mesh of quadrilateral CPE4 elements. Each *true*-valued point in the binary grid corresponds to a square element, with that point representing the top-right vertex, as noted in the subsequent definition of the *addVoxel* function.
The code begins by determining the dimensions of the matrix and creating three main data structures: *nodeMap*, a *containers.Map* object that allows unique keys to be associated with values; *nodeList*, an array containing the coordinates of all generated nodes; and *elemList*, a matrix that stores the quadrilateral elements through the indices of the four nodes composing each element. The *nodeMap* is essential to avoid node duplication: keys

are strings created from the X and Y coordinates of the nodes, while the values correspond to the IDs in *nodeList*. The function assigns a unique identifier to each new node; if a node with the same coordinates already exists, its existing ID is simply reused, preventing duplication. The *nodeList* vector contains three columns: the first two correspond to the X and Y coordinates, while the third column is set to zero because the model is two-dimensional, and Abaqus still requires three coordinates for each node.
The nested *addVoxel* function, as mentioned, takes as input the grid index of the considered *true* node and generates the coordinates of the four voxel nodes, stored in the variable *corners*. For each node, a textual key is created, and using *nodeMap*, it is determined whether to add a new node or reuse an existing one. Once the IDs of the four nodes are determined, a new quadrilateral voxel is created and added to *elemList*. The code then iterates over the entire binary matrix, adding a voxel for each *true* node present. At the end, the top and bottom plates are generated by adding voxels in rows 0 and Nx+1 of the grid.
Subsequently, the minimum and maximum vertical coordinates are calculated to identify the nodes belonging to the bottom and top plates, and a central reference node is identified for each plate, which will serve as the reference point in simulations to apply displacements (*RP_top*) and constraints (*RP_bottom*). Two new unique IDs, separate from the existing node IDs, are assigned to these reference points. The *ELSETs* of the top and bottom plate elements are defined, and using the S2 (select top side) and S4 (select bottom side) commands, the external "surfaces" on which displacements and constraints are applied are identified. These surfaces, as in the 3D case, are linked to their respective reference points through kinematic constraints.
The auxiliary functions *write_elset* and *write_nset* allow writing the element and node sets into the .inp file, following Abaqus syntax and limiting the maximum number of entries per line to sixteen, thus ensuring readability and file compatibility. The writing of the .inp file then follows the same logic as in the three-dimensional case, including the definition of the part, nodes, and elements, assignment of the solid section, assembly, boundary conditions, material properties, and output requests.
The .inp file opened in Abaqus shows the following result:

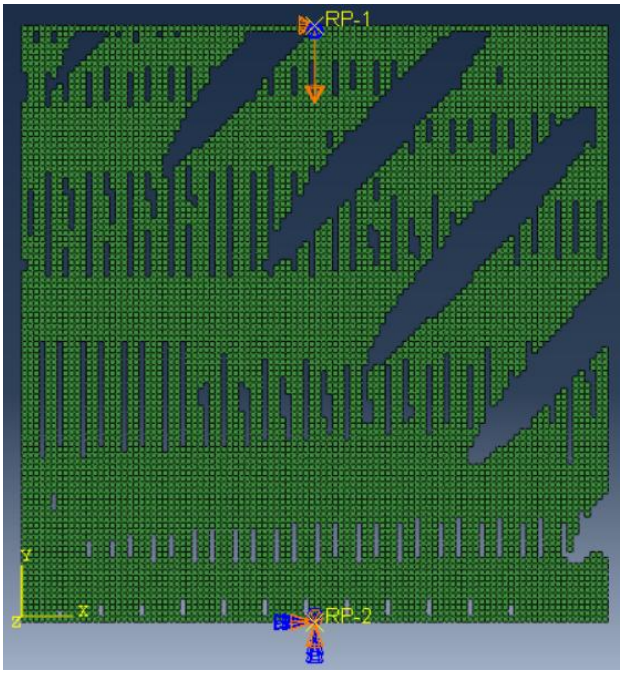


Figure 2.2.20: model opened in Abaqus through the .inp file

The finite element analysis in Abaqus, starting from the input file with the .inp extension, is executed using a batch file called from within the working environment via the *system* function. This approach allows automating the initiation of simulations and the management of their outputs:

```
%% Run simulations with batch file
[~,~] = system('"C:\Users\greco\Politecnico Di Torino Studenti Dropbox\Carlo
               Alberto Greco\Master_thesis_Denis_&_Carlo\Carlo_Alberto_Greco\
               run_simulations.bat"');
```

At the end of the FEM analysis, the code enables the output file with the .odb extension, essential for the subsequent calculation of the effective stiffness, to be moved into a dedicated folder. At the same time, all other output files generated by Abaqus but not relevant to the objectives of the study are deleted:

```
@echo off
setlocal enabledelayedexpansion
set "input_folder=C:\Users\greco\Politecnico Di Torino Studenti Dropbox\Carlo Alberto
                  Greco\Erasmus NTNU master thesis\.inp files"
set "output_folder=C:\Users\greco\Politecnico Di Torino Studenti Dropbox\Carlo Alberto
                   Greco\Erasmus NTNU master thesis\.odb files"
for %%f in ("%input_folder%\*.inp") do (
    echo Processing file: %%~nxf
    pushd "%input_folder%"
    abaqus job=%%~nf cpus=6 interactive
    popd
    if exist "%input_folder%\%%~nf.odb" (
        move /Y "%input_folder%\%%~nf.odb" "%output_folder%\%%~nf.odb"
        echo ODB file moved to %output_folder%
    ) else (
        echo WARNING: file %%~nf.odb not found!
    )
    rem Suppress all delete output
    for %%x in (dat msg sta log prt sim com mdl) do (
        >nul 2>&1 ( if exist "%input_folder%\%%~nf.%%x" del /Q
"%input_folder%\%%~nf.%%x" )
    )
)
echo Processing complete.
Pause
```

The .odb file, when opened through the Abaqus GUI, can be used to visualize the structure under the applied load and the corresponding stress distribution:

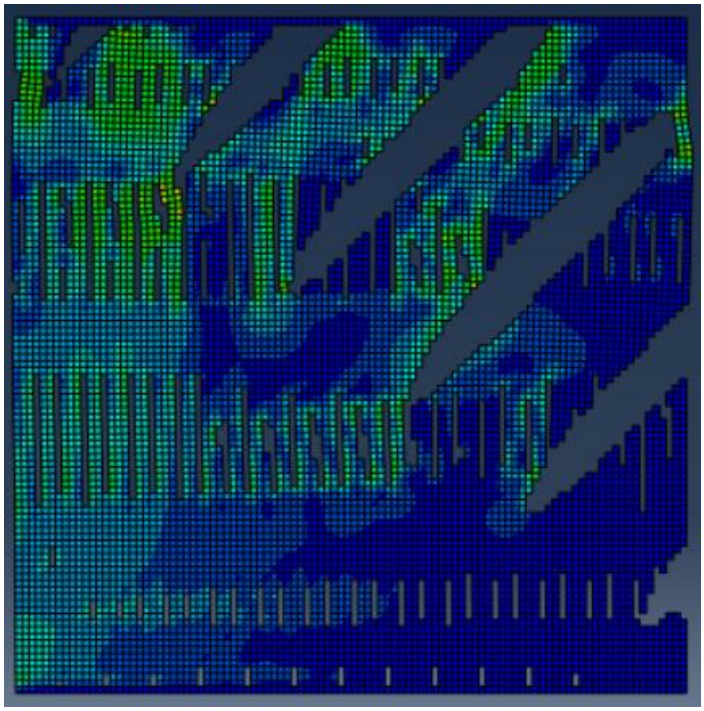


Figure 2.2.21: Stress distribution on the .odb file resulting from the FEM analysis

2.2.4.8 Computation of effective stiffness

The calculation of the effective stiffness is also performed in a fully automated manner for each structure. The process begins with the extraction of the necessary data from the .odb file using the function *extract_RF2_U2.py*. This function was developed in MATLAB and will be illustrated and explained below:

```
%% EXTRACT RF2 U2
clc
clear
folderpath_py = 'C:\Users\greco\Politecnico Di Torino Studenti Dropbox\Carlo
Alberto Greco\Master_thesis_Denis_&_Carlo\Carlo_Alberto_Greco\Matlab_code';
filename_py = 'extract_RF2_U2.py';
filepath_py = fullfile(folderpath_py, filename_py);
fid = fopen(filepath_py, 'w');
fprintf(fid, 'from odbAccess import openOdb\n');
fprintf(fid, 'import numpy as np\n');
fprintf(fid, 'import sys\n\n');
fprintf(fid, "# Reads the file path from the command line\n");
fprintf(fid, "odb_path = sys.argv[1]\n");
fprintf(fid, "odb = openOdb(path=odb_path)\n\n");
fprintf(fid, "step = odb.steps['Step-1']\n");
fprintf(fid, "frame_data = step.frames\n\n");
fprintf(fid, "# Extraction of the displacement component U2 and the reaction force
RF2 at the RP_TOP reference node\n");
fprintf(fid, "region = odb.rootAssembly.nodeSets['RP_TOP']\n");
fprintf(fid, "u2 = []\n");
```

```
fprintf(fid, "rf2 = []\n");
fprintf(fid, "for frame in frame_data:\n");
fprintf(fid, "    u_field = frame.fieldOutputs['U']\n");
fprintf(fid, "    rf_field = frame.fieldOutputs['RF']\n");
fprintf(fid, "    u = u_field.getSubset(region=region).values[0].data[1]  #
                    U2\n");
fprintf(fid, "    rf = rf_field.getSubset(region=region).values[0].data[1]  #
                     RF2\n");
fprintf(fid, "    u2.append(u)\n");
fprintf(fid, "    rf2.append(abs(rf))\n\n");
fprintf(fid, "# Extraction of the coordinates of the first three points of the
             curve\n");
fprintf(fid, "u2 = np.array(u2[:3])\n");
fprintf(fid, "rf2 = np.array(rf2[:3])\n\n");
fprintf(fid, "# Print to stdout (CSV format: U2, RF2)\n");
fprintf(fid, "for u, f in zip(u2, rf2):\n");
fprintf(fid, '    print("{:.10e},{:.10e}".format(u, f))\n\n');
fprintf(fid, "odb.close()\n");
fclose(fid);
```

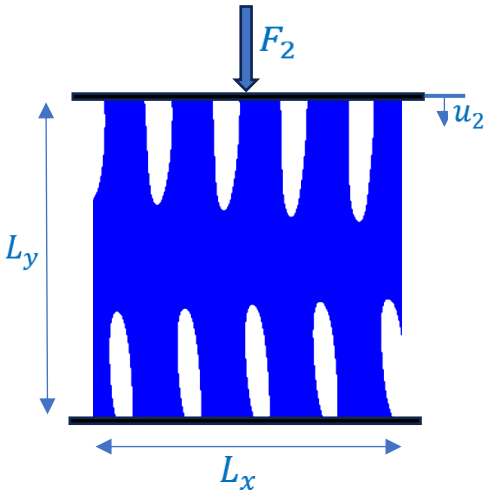
In practice, this MATLAB code generates a Python script that accesses the *.odb* file produced by the FEM analysis of the structure and extracts the displacement and reaction force values at the reference node of the top plate, corresponding to the points of the force–displacement curve associated with the plate.
For the calculation of the effective stiffness, defined subsequently as the slope of the initial linear portion of the stress–strain curve, only a few initial points are required (theoretically two). In this case, the first three points are considered and printed as output. This function is called in the main code as follows:

```
%% Compute effective stiffness
    folderpath_odb = 'C:\Users\greco\Politecnico Di Torino Studenti Dropbox\Carlo
                      Alberto Greco\Erasmus NTNU master thesis\.odb files';
    files = dir(fullfile(folderpath_odb, '*.odb'));
    if isempty(files)
        error('No .odb file found in the folder.');
    elseif length(files) > 1
        warning('More than one .odb file was found; only the first one will be
                 used: %s', files(1).name);
    end
    odb_file = files(1).name;
    % full path of the .odb file
    full_odb_path_py = strrep(fullfile(folderpath_odb, odb_file), '\', '/');
    folderpath_py = 'C:\Users\greco\Politecnico Di Torino Studenti Dropbox\Carlo
                     Alberto Greco\Master_thesis_Denis_&_Carlo\Carlo_Alberto_Greco
                     \Matlab_code';
    filename_py = 'extract_RF2_U2.py';
    % full path of the .py file
    filepath_py = fullfile(folderpath_py, filename_py);

    command = sprintf('abaqus python "%s" "%s"', filepath_py, full_odb_path_py);
    [status, cmdout] = system(command);
    lines = strsplit(strtrim(cmdout), '\n');
    u2 = zeros(length(lines), 1);
    rf2 = zeros(length(lines), 1);
    for i = 1:length(lines)
        nums = sscanf(lines{i}, '%e,%e');
        u2(i) = abs(nums(1));
```

```
    rf2(i) = nums(2);
end
Lx = domain_size;
Ly = domain_size;
epsilon = u2 / Ly;
sigma = rf2 / Lx;
p = polyfit(epsilon, sigma, 1);
E_eff = p(1);
normalized_Eeff=E_eff/finalrelativedensity_2;
fprintf('Effective stiffness normalized with relative density for the %d
         structure is: %.2f Mpa\n',d, normalized_Eeff);
```

After defining the full path of the *.odb* file and of the previously generated Python script *extract_RF2_U2.py*, the system command (*system*) invokes Abaqus in Python mode, executing the script and passing the .odb file path as an argument. The output of the Python script, consisting of the U2 and RF2 values for the first three points of the curve, is captured in the variable *cmdout*.
For each line of the output, the values are read using *sscanf*, stored in the arrays *u2* and *rf2*, and, in the case of U2, converted to absolute values to consider only the displacement magnitude. For 2D metamaterials, it holds that:



$$\varepsilon_y = \varepsilon = \frac{\Delta y}{L_y} = \frac{u_2}{L_y}$$

$$\sigma = \frac{F_2}{L_x \cdot 1} \left[\frac{N}{mm^2}\right]$$

A unit thickness is assumed by convention

In the linear portion of the stress-strain curve:

$$E = \frac{\sigma}{\varepsilon} = \frac{F_2}{L_x \cdot 1} \cdot \frac{L_y}{u_2}$$

In this case $L_x = L_y$ (square cells):

$$E = \frac{F_2}{u_2}$$

Figure 2.2.22: Calculation of effective stiffness E

The code calculates the strain and the stress σ as described previously.
To estimate the effective stiffness, a linear regression is performed using *polyfit* between the stress and strain, obtaining the slope of the linear portion of the curve, which represents the effective stiffness (E_eff). This value is then normalized with respect to the final relative density of the structure (*finalrelativedensity_2*), yielding *normalized_Eeff*.

2.2.4.9 Dataset definition for neural network development

The final step consists of saving the data necessary for training the neural network, which will be developed and described in Section 2.2.5 – Development of the Message Passing Neural Network. For each hierarchical spinodoid structure, the information is collected within a *struct* variable, which contains:

- A matrix with the coordinates and the topological degree of the end nodes of each skeleton chain [Nnodes×3];
- A matrix with the indices of the initial and final nodes of each chain [2×Nedges];
- Vectors of lengths and average thicknesses of the branch chains;
- The effective stiffness normalized with respect to the relative density.

Regarding the construction of the node matrix, the code initially uses the variable *branchNodes_2*, which contains the indices of the nodes with degree greater than or equal to 3 (i.e., the branching nodes). However, these are not sufficient to fully describe the structure: the chains can also begin or end at terminal nodes (degree 1).

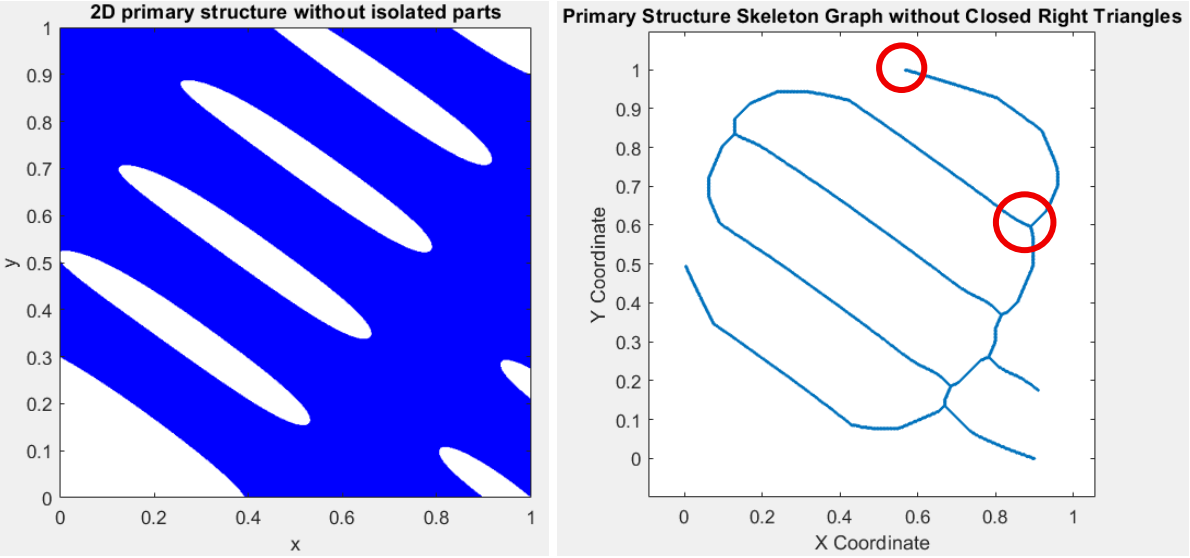


Figure 2.2.23: Example of a branch chain in which one end is a terminal node and the other end is a branching node

For this reason, the code is designed to complete the list of nodes by automatically identifying the missing ones and adding them to the variable *graphdata_nodes_2*.

```
%% DATA FOR GNN (HIERARCHICAL structure)
% matrix of node information, with order and size given by branchNodes_2
graphdata_nodes_2 = [skeletonpoints_2(branchNodes_2,1),
skeletonpoints_2(branchNodes_2,2), NodeDegree_2(branchNodes_2)];
graphdata_edgeindex_2 = zeros(2, numel(chainNodes_2));
current_node_list = branchNodes_2;  % keeps the nodes already included
for k = 1:numel(chainNodes_2)
    v = chainNodes_2{k};
```

```
        % Start node
        idx_start = find(current_node_list == v(1));
        if isempty(idx_start)
            % Add the start node
            graphdata_nodes_2(end+1, :) = [skeletonpoints_2(v(1),1),
            skeletonpoints_2(v(1),2), NodeDegree_2(v(1))];
            current_node_list(end+1) = v(1);
            idx_start = size(graphdata_nodes_2,1);  % new index (last element)
        end
        graphdata_edgeindex_2(1, k) = idx_start;
        % End node
        idx_end = find(current_node_list == v(end));
        if isempty(idx_end)
            % Add the end node
            graphdata_nodes_2(end+1, :) = [skeletonpoints_2(v(end),1),
            skeletonpoints_2(v(end),2), NodeDegree_2(v(end))];
            current_node_list(end+1) = v(end);
            idx_end = size(graphdata_nodes_2,1);  % new index (last element)
        end
        graphdata_edgeindex_2(2, k) = idx_end;
    end
    % chain lengths (with order consistent with graphdata_edgeindex)
    graphdata_edgelengths=chainLengths_2;
    % average thickness of the chains
    graphdata_edgethickness=cellfun(@mean,chain_radii_2);
```

As can be seen from the code above, for each skeleton chain of the structure, it is verified that the initial and final points are included in the list of nodes *graphdata_nodes_2*; if not, they are added.
Once the construction of the topological and geometric data is complete, all information is collected within a *struct* variable named *graphdata_2*. This contains the main fields *nodes*, *edgeindex*, *edgelengths*, *edgethickness*, and *effectivestiffness*, the latter corresponding to the normalized effective stiffness of the structure. Finally, the variable *graphdata_2* is saved in a .mat file within a folder dedicated to the GNN inputs, using a progressive name (*structure_d.mat*) that uniquely identifies each sample in the dataset.

```
    % Save everything for each structure
    graphdata_2 = struct();
    graphdata_2.nodes = graphdata_nodes_2;
    graphdata_2.edgeindex = graphdata_edgeindex_2;
    graphdata_2.edgelengths = graphdata_edgelengths;
    graphdata_2.edgethickness = graphdata_edgethickness;
    graphdata_2.effectivestiffness = normalized_Eeff;
    folder_GNN='C:\Users\greco\Politecnico Di Torino Studenti Dropbox\Carlo
    Alberto Greco\Master_thesis_Denis_&_Carlo\Carlo_Alberto_Greco\ GNN_inputdata\
    GNN_inputdata_hierarchical';
    filename_GNNdata = sprintf('struttura_%d.mat', d);
    fullpath = fullfile(folder_GNN, filename_GNNdata);  % builds the full path
    save(fullpath, 'graphdata_2');
```

```
              nodes: [282×3 double]
          edgeindex: [2×376 double]
        edgelengths: [0.0310 0.0902 0.0478 0.1257 0.0961 0.0502 0.1059 0.0818 0.0167 0.0488
      edgethickness: [1.7683 2.6318 6.1974 1.4703 2.5453 7.2789 2.3485 2.2037 3.9904 2.5943
effectivestiffness: 212.6028
```

Figure 2.2.24: Illustrative example of the contents of each struct

2.2.5 Development of the Message Passing Neural Network

As outlined in Section 2.1.2.3 – Graph Neural Networks and Message Passing Neural Networks, a Message Passing Neural Network (MPNN) was implemented in MATLAB with the objective of predicting the uniaxial compressive stiffness of two-dimensional spinodoid structures.
The dataset, constructed according to the methodology described in Section 2.2.4.9 – Dataset definition for neural network development, is divided into two parts: one used for training the neural network and the other for testing, aimed at evaluating the prediction error of the MPNN in estimating the effective stiffness of the analyzed structures.
The graph on which the network operates corresponds to the topological skeleton of each structure, which can be assimilated to that obtained through the thinning operation.
In this representation, the nodes correspond to the branching points or endpoints of the chains, while the edges represent the branch chains, defined in Section 2.2.1.6.3 – Branch chain analysis.
Since the MPNN can process features associated with edges, each edge is assigned two properties: the length and the average thickness of the corresponding branch chain. It should be recalled that the thickness along a chain is determined based on the radius of the largest inscribed sphere (or circle in 2D).
The nodal features, on the other hand, include the normalized x–y coordinates and the node degree. Finally, each structure is associated with its effective stiffness, calculated through finite element analysis (FEM).
Before presenting in detail the MATLAB code used for the neural network implementation, a flowchart is shown to summarize the main steps of the entire process:

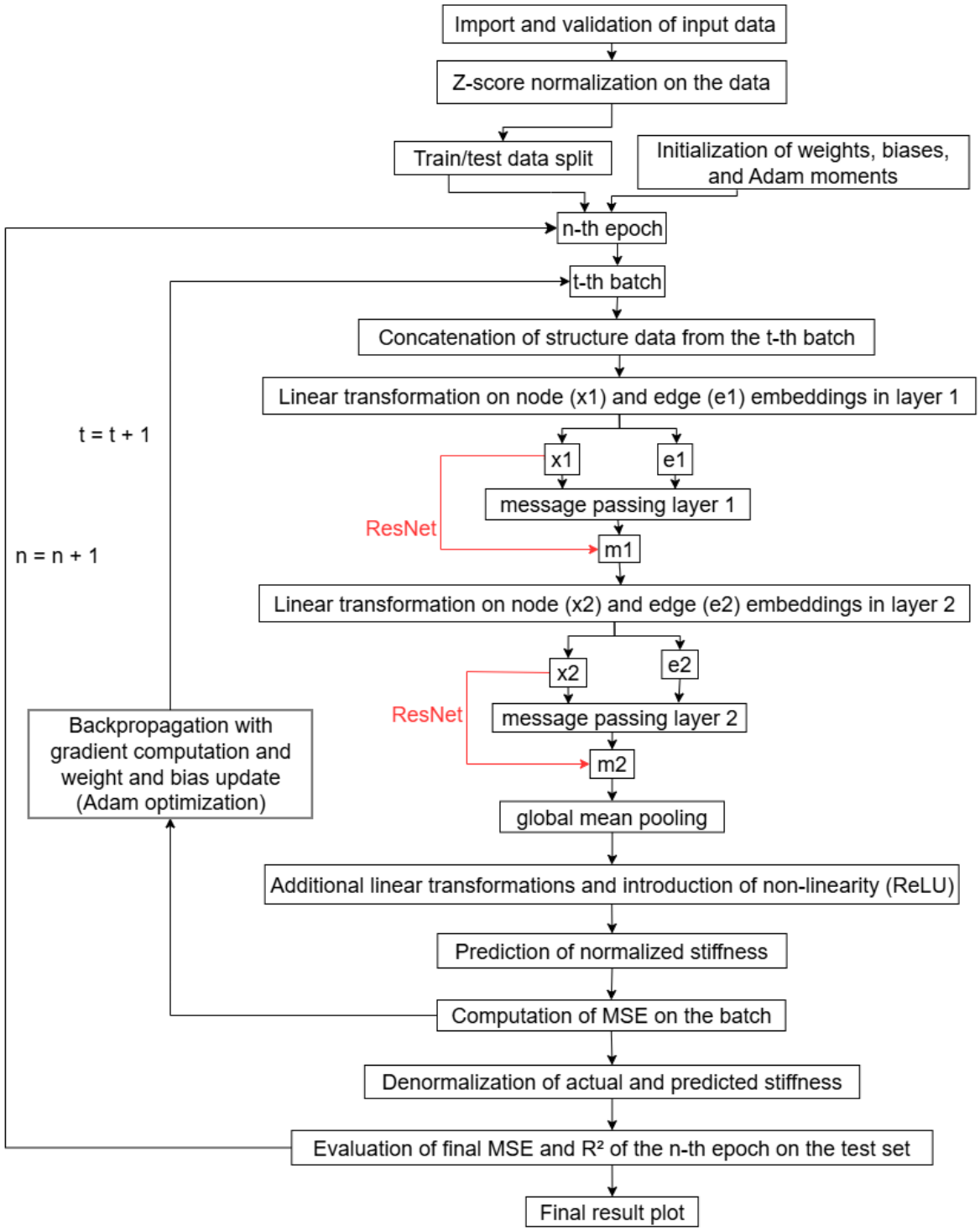


Figure 2.2.25: Flowchart of the Message Passing Neural Network (MPNN)

In the following section, the MATLAB code used to implement the MPNN will be analyzed, divided into several parts with corresponding explanations to enhance clarity.

2.2.5.1 Data Import, Processing, and Splitting

The initial phase concerns the import of the dataset:

```
%% INPUT DATA IMPORT AND SELECTION
folder = 'C:\Users\greco\Politecnico Di Torino Studenti Dropbox\Carlo Alberto
Greco\Erasmus NTNU master thesis\Dataset_results\finaldataset_12000\GNN_inputdata
\GNN_inputdata_hierarchical';
files = dir(fullfile(folder, '*.mat'));
num_files = length(files);
% — Sort the files based on the number in their name —
fileNums = zeros(num_files,1);
for k = 1:num_files
    filenumb = regexp(files(k).name, 'struttura_(\d+)\.mat', 'tokens');
    if ~isempty(filenumb)
        fileNums(k) = str2double(filenumb{1}{1});
    else
        error('Unexpected file name: %s', files(k).name);
    end
end
[~, sortIdx] = sort(fileNums);
files = files(sortIdx);
% Preallocate cells for graph data
node_features = cell(num_files,1);
edge_indices  = cell(num_files,1);
edge_features = cell(num_files,1);
targets       = zeros(num_files,1);
valid_graph_idx = 0;
for i = 1:num_files
    data = load(fullfile(folder, files(i).name));
    g = data.graphdata_2;
    % Check if the graph is valid
    if isempty(g.edgeindex) || size(g.edgeindex,2) == 0
        fprintf('Graph %d : not considered, no edges.\n', i);
        continue
    end
    if g.effectivestiffness <= 5
        fprintf('Graph %d : not considered, stiffness too low.\n', i);
        continue
    end
    % If valid, store graph features
    valid_graph_idx = valid_graph_idx + 1;
    node_features{valid_graph_idx} = g.nodes;
    edge_indices{valid_graph_idx}  = g.edgeindex;
    edge_feat = [g.edgelengths(:), double(g.edgethickness(:))];
    edge_features{valid_graph_idx} = edge_feat;
    targets(valid_graph_idx) = g.effectivestiffness;
end
% Trim arrays to actual valid size
node_features = node_features(1:valid_graph_idx);
edge_indices  = edge_indices(1:valid_graph_idx);
edge_features = edge_features(1:valid_graph_idx);
targets       = targets(1:valid_graph_idx);
```

The code begins with the import and selection of the input data, consisting of the *struct* variables defined for each structure. These data are first sorted according to the index assigned during their generation, in order to facilitate their identification during the subsequent selection phase.

Cell arrays are preallocated to store the node features, edge indices, edge features, and target values corresponding to the effective stiffness of each graph. Each cell contains a number of matrices equal to the total number of structures in the dataset; therefore, if the dataset includes 12,000 structures, the cell corresponding to the node features, for example, will contain 12,000 matrices, each representing the features of all nodes of a single graph.
A *for* loop iterates over all files (*struct* variables), loading the data of each graph and verifying their validity: graphs without edges in the hierarchical structure or with excessively low (often null) stiffness values are discarded. In both cases, these anomalies arise from the issue of detached parts, discussed in Section 2.2.1.2 – Removing detached parts and Section 2.2.4.2 – Detached parts issue in hierarchical lamellar spinodoids.
Thanks to the ordering of the files and their indexing, it is possible to easily verify the discarded structures through a visual inspection of the corresponding figures, which were previously saved and indexed in a consistent manner.

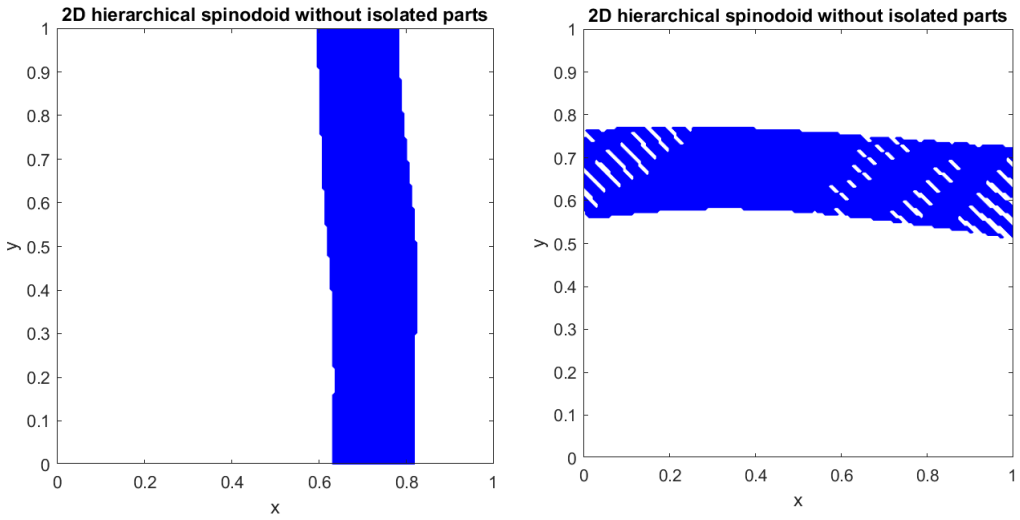


Figure 2.2.26: Examples of discarded spinodoid structures: one with no edges in the hierarchical network (left) and one with zero stiffness (right).

As illustrated in the figure on the left, although a lamella connects the upper and lower plates (located at y = 0 and y = 1), it belongs to the primary structure; since the MPNN operates on data from the hierarchical structure, this case is interpreted as a graph without edges.
In the structure shown on the right, the stiffness is null because the plates are not connected to the structure, preventing the calculation of a meaningful stiffness value.
For the valid graphs, the node features are stored directly, while for the edges, the length and average thickness are collected into a single matrix. The target vector corresponds to the effective stiffness values of all structures in the dataset, computed through FEM analysis. Finally, the cell arrays and the target vector are truncated to the actual size of the valid graphs, ensuring that all the structures considered contain consistent and complete data for the subsequent normalization and neural network training phases.
The next phase involves the normalization of the input data. In a neural network, this step ensures more stable, faster, and more robust learning, prevents certain features from dominating others, and improves the quality of the messages propagated within the MPNN. In this case, z-score normalization is applied: given a dataset, this normalization requires

the computation of the standard deviation σ and the mean μ. Each value x is then replaced by its normalized value z, calculated as:

$$z = \frac{x - \mu}{\sigma}$$

This transforms the dataset to have a zero mean and a unit standard deviation.

```
%% INPUT DATA NORMALIZATION
% Normalize x and y coordinates of all graphs
all_coords = cell2mat(cellfun(@(nf) nf(:,1:2), node_features, 'UniformOutput',
false));
% Compute mean and std for x and y separately
mean_coords = mean(all_coords, 1);  % [mean_x, mean_y]
std_coords  = std(all_coords, 0, 1); % [std_x, std_y]
% Apply z-score normalization to node coordinates
for i = 1:numel(node_features)
    node = node_features{i};
    node(:,1:2) = (node(:,1:2) - mean_coords) ./ std_coords;
    node_features{i} = node;  % Overwrite in cell array
end

% Normalize node degrees (column 3)
% Extract all degrees from all graphs
all_degrees = cell2mat(cellfun(@(nf) nf(:,3), node_features, 'UniformOutput',
false));
% Compute mean and std of degrees
mean_degree = mean(all_degrees);
std_degree  = std(all_degrees);
% Apply z-score normalization to node degrees
for i = 1:numel(node_features)
    node = node_features{i};
    node(:,3) = (node(:,3) - mean_degree) / std_degree;
    node_features{i} = node;
end

% Normalize edge lengths and thickness for all graphs
all_edge_lengths  = cell2mat(cellfun(@(ef) ef(:,1), edge_features,
'UniformOutput', false));
all_edge_thickness = cell2mat(cellfun(@(ef) ef(:,2), edge_features,
'UniformOutput', false));
% Compute mean and std separately
mean_length = mean(all_edge_lengths);
std_length  = std(all_edge_lengths);
mean_thickness = mean(all_edge_thickness);
std_thickness  = std(all_edge_thickness);
% Apply z-score normalization separately to length and thickness
for i = 1:numel(edge_features)
    ef = edge_features{i};
    ef(:,1) = (ef(:,1) - mean_length) / std_length;       % length
    ef(:,2) = (ef(:,2) - mean_thickness) / std_thickness; % thickness
    edge_features{i} = ef;
end

% Normalize targets (effective stiffness)
mean_target = mean(targets);
std_target  = std(targets);
targets_norm = (targets - mean_target) / std_target;
```

For the nodes of the graphs, the x–y coordinates and the node degree are normalized by computing the mean and standard deviation over the entire dataset and applying the z-score transformation to each node. Similarly, for the edges, the length and thickness are normalized, while for the targets, corresponding to the effective stiffness of the structures, the mean and standard deviation of the entire vector are computed and the same normalization is applied. In this way, all node features, edge features, and targets have zero mean and unit variance, facilitating network training and ensuring improved convergence.
Subsequently, the normalized dataset is divided into training and test sets, a fundamental step for evaluating the performance of a neural network on data that were not seen during training.

```
%% Train/Test Split
num_samples = length(targets_norm);
idx = randperm(num_samples);
num_train = floor(0.95*num_samples);
train_idx = idx(1:num_train);
test_idx = idx(num_train+1:end);
train_data = struct();
test_data = struct();
train_data.node_features = node_features(train_idx);
train_data.edge_indices = edge_indices(train_idx);
train_data.edge_features = edge_features(train_idx);
train_data.targets = targets_norm(train_idx);
test_data.node_features = node_features(test_idx);
test_data.edge_indices = edge_indices(test_idx);
test_data.edge_features = edge_features(test_idx);
test_data.targets = targets_norm(test_idx);
```

First, the total number of available samples is determined, and a random permutation of the corresponding indices is generated, ensuring a random distribution of the data between the two sets. Subsequently, the proportion of data to be allocated for training is defined, set at 95% of the total, while the remaining 5% is reserved for testing. Although 5% may seem small, considering that the dataset comprises approximately 12,000 structures (slightly fewer if some were discarded), the test is still performed on around 600 structures. From the generated random indices, the corresponding subsets for each component of the dataset are extracted: the node features, the edge information (both connections and attributes), and the normalized targets. These subsets are then organized into dedicated data structures (*structs*) for training and testing, respectively named *train_data* and *test_data*.

2.2.5.2 Initialization and Hyperparameter Setup

Next, the main hyperparameters used for training the Message Passing Neural Network are defined:

```
%% MPNN HYPERPARAMETERS
hidden_dim = 128;
learning_rate = 1e-3;
lr_reduced = false; % flag for learning rate scheduler
num_epochs = 120;
batch_size = 32;
```

```
beta1 = 0.9; % Exponential decay rate for 1st moment in Adam optimizer
beta2 = 0.999; % Exponential decay rate for 2nd moment in Adam optimizer
epsilon_adam = 1e-8; % Numerical stability term for Adam
```

The parameter *hidden_dim* represents the dimensionality of the latent space or the number of units in the hidden layers of the network. This value controls the representational capacity of the model: higher values allow the network to learn more complex relationships among the data but also increase the risk of overfitting and computational cost. Analytically, *hidden_dim* directly affects the size of the vectors and matrices used to represent the embeddings of nodes, edges, and other entities within the MPNN.
The *learning_rate* defines the step size used by the optimizer. It determines the magnitude of the steps taken by the optimization algorithm along the direction indicated by the gradient of the loss function, in this case the Mean Squared Error (MSE). A value of $1 \cdot 10^{-3}$ represents a compromise between convergence speed and numerical stability.
The boolean variable *lr_reduced* acts as a control flag for the potential automatic reduction of the learning rate (learning rate scheduler). As will be shown later, the learning rate is reduced by a predetermined factor (in this case, divided by four) if the coefficient of determination R^2 exceeds a specific threshold. In such cases, the *lr_reduced* flag is set to true.
Adjusting the learning rate during the loss descent is a common practice in neural network training, as it helps stabilize the optimization process and improve final performance. Generally, this adjustment occurs gradually over the epochs; in the present implementation, however, a simplified approach is adopted, in which the learning rate is reduced only once during the final phase of training.
The parameter *num_epochs* specifies the number of training epochs, i.e., the number of times the entire dataset is presented to the network. In this case, the model is trained for 120 epochs, although performance improvements in the final epochs are minimal.
The *batch_size* indicates the the number of samples processed simultaneously during each weight update. A value of 32 provides a good balance between optimization stability and computational efficiency.
The parameters *beta1* and *beta2* represent the exponential decay rates used by the Adam optimizer, which updates the network parameters (weights and biases) after each batch. The precise role of these parameters will be analyzed in more detail later, in the context of the Adam optimization discussion.
Finally, the terms *epsilon_adam* serve as numerical stability factors, used in the Adam optimizer.
Subsequently, the weights and biases of the neural network are initialized:

```
%% Initialize weights and biases with He initialization
% Layer 1
W_node1 = randn(3, hidden_dim) * sqrt(2/3);
b_node1 = zeros(1, hidden_dim);
W_edge1 = randn(2, hidden_dim) * sqrt(2/2);
b_edge1 = zeros(1, hidden_dim);
W_msg1  = randn(hidden_dim, hidden_dim) * sqrt(2/hidden_dim);
b_msg1  = zeros(1, hidden_dim);
% Layer 2
W_node2 = randn(hidden_dim, hidden_dim) * sqrt(2/hidden_dim);
b_node2 = zeros(1, hidden_dim);
W_edge2 = randn(2, hidden_dim) * sqrt(2/2);
b_edge2 = zeros(1, hidden_dim);
```

```
W_msg2  = randn(hidden_dim, hidden_dim) * sqrt(2/hidden_dim);
b_msg2  = zeros(1, hidden_dim);
% Output layers
W_out1 = randn(hidden_dim, hidden_dim) * sqrt(2/hidden_dim);
b_out1 = zeros(1, hidden_dim);
W_out2 = randn(hidden_dim, 1) * sqrt(2/hidden_dim);
b_out2 = 0;
```

The initialization is performed using a method known as He initialization. This choice is motivated by the need to maintain stable signal propagation within the network, preventing phenomena commonly referred to as vanishing or exploding gradients.
In a neural network, each neuron produces a value called an activation, which represents the neuron's output after the linear combination of the input values weighted by the parameters, the addition of the bias, and, optionally, the application of a nonlinear activation function. The variance of the activations measures how much these values deviate, on average, from their mean. Maintaining this variance under control is essential: if the variance is too small, the activations approach zero and the gradient tends to vanish (vanishing gradient), whereas if the variance is too large, the values explode, causing numerical instability (exploding gradient). He initialization ensures that the variance of the activations remains approximately constant across layers, enabling stable signal propagation.
Specifically, for the first layers of the network:

- *W_node1* and *b_node1* define the weight matrix and bias vector, respectively, for the transformation of the input node features into the latent space of dimension *hidden_dim*. The weights are initialized with Gaussian random values scaled by $\sqrt{2/3}$, where 3 corresponds to the number of input features per node (x coordinate, y coordinate, node degree).

- *W_edge1* and *b_edge1* perform a similar role for the edge features, with scaling $\sqrt{2/2}$ consistent with the input dimension of 2 (length, thickness).

- *W_msg1* and *b_msg1* define the transformation of messages between nodes, with dimensions compatible with the latent space *hidden_dim*.

The subsequent layers (*W_node2*, *b_node2*, *W_edge2*, *b_edge2*, *W_msg2*, *b_msg2*) are initialized in a similar manner, maintaining the latent dimension *hidden_dim* constant to ensure consistency in the aggregation and message-update operations between nodes and edges.
Finally, the output layers (*W_out1*, *b_out1*, *W_out2*, *b_out2*) map the aggregated latent representations first into an intermediate space of dimension *hidden_dim* and subsequently into a single scalar value, corresponding to the final prediction of the effective stiffness.
All network parameters are then grouped into a single cell array, and, based on these, the cell arrays containing the Adam optimizer moments are also initialized:

```
%% Initialize Adam moments for all parameters ---
params = {W_node1,b_node1,W_edge1,b_edge1,W_msg1,b_msg1, ...
          W_node2,b_node2,W_edge2,b_edge2,W_msg2,b_msg2, ...
          W_out1,b_out1,W_out2,b_out2};
```

```
m = cellfun(@(x) zeros(size(x)), params, 'UniformOutput', false); % cell array
matching size of params (first moment)
v = cellfun(@(x) zeros(size(x)), params, 'UniformOutput', false); % cell array
matching size of params (second moment)
```

In the Adam optimizer, each network parameter (weights *W* and biases *b*) maintains two auxiliary quantities during training: the first moment *m*, corresponding to the exponential moving average of the gradients of the MSE loss with respect to the parameter, and the second moment *v*, corresponding to the exponential moving average of the squared gradients.
In practice, the code defines a cell array *params* that collects all network parameters and creates two cell arrays, *m* and *v*, of the same size as each parameter, initialized to zero. These arrays are used during the weight and bias update steps in Adam optimization, enabling the computation of adaptive gradients that account for both the mean and variance of the past gradients.
The introduction of nonlinearity in this MPNN is implemented using the ReLU function:

```
%% Non linearity
relu = @(x) max(0,x);          % ReLU activation
drelu = @(x) double(x>0);      % Derivative of ReLU
```

Specifically:

- *relu = @(x) max(0,x)* implements the ReLU function, which returns the input value if positive and zero otherwise. This function introduces nonlinearity into the network, enabling it to learn complex relationships among the features of nodes, edges, and messages;

- *drelu = @(x) double(x>0)* represents the derivative of the ReLU, which is necessary during backpropagation to compute the gradients of the weights and biases. The derivative takes the value 1 for positive inputs and 0 otherwise, reflecting the behavior of the ReLU function.

2.2.5.3 Training and parameters Update

Following these preparatory steps, the actual training of the network is implemented:

```
%% Training loop
num_batches = ceil(num_train / batch_size); % number of batches per epoch
t = 0; % batch counter for Adam bias correction

% initialization of variables to monitor the best R²
best_R2 = -inf;
best_epoch = 0;
best_y_true_real = [];
best_y_pred_real = [];

for epoch = 1:num_epochs
    % Shuffle training data
    perm = randperm(num_train);
    for batch_i = 1:num_batches
        t = t + 1;
```

```
        batch_idx = perm((batch_i-1)*batch_size+1 : min(batch_i*batch_size,
                    num_train));
        % Prepare concatenated batch data
        % --- Concatenate nodes, edges, and targets for the batch ---
        % Create a batch graph by concatenating individual graphs with node index
          shifts
        [batch_x, batch_edge_idx, batch_edge_attr, batch_y, batch_graph_ptr] =
        prepare_batch(train_data, batch_idx); % batch_x: Mx3, batch_edge_idx: 2xE,
                                              batch_edge_attr: Ex2, batch_y: Bx1
        % FORWARD PASS
        % Layer 1
        x1 = batch_x * W_node1 + b_node1;  % Mxhidden_dim, node features linear
                                           transformation
        e1 = batch_edge_attr * W_edge1 + b_edge1; % Exhidden_dim, edge features
                                                    linear transformation
        % Message passing layer 1
        delta1 = message_passing(x1, batch_edge_idx, e1, W_msg1, b_msg1, relu);
        %propagate messages along edges
        m1 = relu(delta1 + x1);    % apply residual connection and activation
        % Layer 2
        x2 = m1 * W_node2 + b_node2;
        e2 = batch_edge_attr * W_edge2 + b_edge2;
        delta2 = message_passing(x2, batch_edge_idx, e2, W_msg2, b_msg2, relu);
        m2 = relu(delta2 + x2);
        % Global mean pooling: average node embeddings per graph
        graph_embed = global_mean_pool(m2, batch_graph_ptr); % Bxhd
        % Output MLP
        h = relu(graph_embed*W_out1 + b_out1);
        out = h*W_out2 + b_out2; % Bx1, normalized predictions
        % --- Compute MSE loss ---
        loss = mean((out - batch_y).^2);
        % BACKPROP (compute gradients of MSE w.r.t. parameters)
        grads = backward_pass(batch_x, batch_edge_idx, batch_edge_attr, batch_y,
                batch_graph_ptr,
                W_node1, b_node1, W_edge1, b_edge1, W_msg1, b_msg1, ...
                W_node2, b_node2, W_edge2, b_edge2, W_msg2, b_msg2, ...
                W_out1, b_out1, W_out2, b_out2, ...
                relu, drelu, epsilon_norm);
        % ADAM OPTIMIZATION
        for p = 1:length(params)
            % Update moments
            m{p} = beta1 * m{p} + (1-beta1) * grads{p}; % First moment estimate
            v{p} = beta2 * v{p} + (1-beta2) * (grads{p}.^2); % Second moment
                                                               estimate
            m_hat = m{p} / (1 - beta1^t); % Bias-corrected first moment
            v_hat = v{p} / (1 - beta2^t); % Bias-corrected second moment
            % Update parameter
            params{p} = params{p} - learning_rate * m_hat ./ (sqrt(v_hat) +
                        epsilon_adam);
        end
        % Overwrite parameters with updated values
        [W_node1,b_node1,W_edge1,b_edge1,W_msg1,b_msg1,...
         W_node2,b_node2,W_edge2,b_edge2,W_msg2,b_msg2,...
         W_out1,b_out1,W_out2,b_out2] = params{:};
    end
```

Training is performed over a predetermined number of epochs (*num_epochs*) and utilizes mini-batches of size *batch_size* randomly sampled from the training dataset.

At the beginning of each epoch, the training data are randomly shuffled (*randperm*) to ensure that each batch contains different samples, reducing the risk of overfitting and minimizing undesired correlations between consecutive batches.
Once the indices of the structures included in the *t*-th batch are defined, the data are concatenated using the *prepare_batch* function:

```
% Batch preparation function (concatenate nodes/edges with node index shift)
function [batch_x, batch_edge_idx, batch_edge_attr, batch_y, batch_graph_ptr] =
prepare_batch(data, batch_idx) % Concatenate node features, edge indices, edge
                               features, and targets into a batch
    batch_x = [];
    batch_edge_idx = [];
    batch_edge_attr = [];
    batch_y = [];
    batch_graph_ptr = [];
    node_offset = 0;
    for i = batch_idx  % loop over each graph in the batch
        nf = data.node_features{i};
        ei = data.edge_indices{i};
        ef = data.edge_features{i};
        batch_x = [batch_x; nf];              % concatenate node features
        batch_edge_idx = [batch_edge_idx, ei + node_offset]; % shift edge indices
                                                               to avoid collisions
        batch_edge_attr = [batch_edge_attr; ef];   % concatenate edge features
        batch_y = [batch_y; data.targets(i)];      % collect targets
        batch_graph_ptr = [batch_graph_ptr; size(nf,1)]; % store number of nodes
                                                           per graph (needed for pooling)
        node_offset = node_offset + size(nf,1);% update node offset for next graph
    end
end
```

This step is fundamental for enabling the MPNN to process multiple graphs simultaneously during the forward pass.
Specifically, for each graph included in the batch, the following operations are performed:

- Node feature concatenation: the features of each node (node_features) are accumulated into a single array *batch_x* of size [M×3], where M corresponds to the total number of nodes across all graphs in the batch;

- Edge index concatenation: the edge indices (*edge_indices*) are updated using a cumulative offset (*node_offset*) to prevent index duplication between nodes of different graphs in the batch. This ensures that connections between nodes are correctly maintained during message passing. The updated indices are then collected in the variable *batch_edge_idx* of size [2×E], where E represents the total number of edges across all graphs in the batch;

- Edge feature concatenation: the edge features (*edge_features*) are accumulated in *batch_edge_attr* [E×2], corresponding to the updated indices;

- Target values concatenation: the target values (effective stiffness) associated with each graph are concatenated into *batch_y*, allowing the subsequent computation of the loss function (MSE) for the entire batch. The vector *batch_y* contains B elements, with B being the total number of graphs in the batch.

The variable *batch_graph_ptr* stores the number of nodes present in each graph of the batch, providing essential information for the global mean pooling operation during the forward pass.
As previously described, to avoid index duplication, the nodes of individual graphs are identified using shifted indices through a cumulative batch counter, called *node_offset*.
When the node embedding vectors are later aggregated to obtain the global graph embedding (*global mean pooling*), the variable *batch_graph_ptr* allows correct identification of which nodes belong to each graph. In this way, even though all node embeddings are contained in a single matrix of size [M×hidden_dim], the averaging operation is applied only to the nodes of the corresponding graph, preserving the topological structure of each batch element.
The forward pass starts from layer 1, where the first operation is the linear transformation of node and edge features (*batch_x* and *batch_edge_attr*). Algebraically, this can be expressed as:

$$x_1 = batch\_x \cdot W\_node_1 + b\_node_1$$
$$e_1 = batch\_edge\_attr \cdot W\_edge_1 + b\_edge_1$$

Thus, x_1 will be a matrix of size [M×hidden_dim], since *batch_x* has dimensions [M×3], $W_{\text{node,1}}$ is [3×hidden_dim], and $b_{\text{node,1}}$ is [1×hidden_dim].
Similarly, e_1 will be a matrix of size [E×hidden_dim], as *batch_edge_attr* has dimensions [E×2], $W_{\text{edge,1}}$ is [2×hidden_dim], and $b_{\text{edge,1}}$ is [1×hidden_dim].
Once the node and edge embeddings are obtained, message passing is performed. To this end, the following message_passing function has been implemented:

```
function m_out = message_passing(x, edge_idx, e, W_msg, b_msg, relu_fun)
    % x: MxH node features
    % edge_idx: 2xE matrix (start; end nodes)
    % e: ExH edge features
    % W_msg,b_msg: weights and bias for message transformation
    % For each edge:
    xj = x(edge_idx(2,:),:);   % embedding vectors of target nodes
    msg = relu_fun((xj + e)*W_msg + b_msg); % ExH, compute messages
    % Sum messages for each start node (aggregation)
    N = size(x,1);
    m_out = zeros(N, size(W_msg,2));
    for i=1:size(edge_idx,2)
        start_node = edge_idx(1,i); % index of the start node for current edge
        m_out(start_node,:) = m_out(start_node,:) + msg(i,:); % sum message to
                                                              corresponding start node
    end
end
```

For each edge of the graph, the function retrieves the embedding of the destination node and combines them into a matrix (x_j) of size [E×hidden_dim]; this matrix is then added to the edge embedding matrix (e) of the same size. The rows of the two matrices are aligned, each corresponding to the same edge.
This combined representation is subsequently transformed linearly using the weights W_{msg} [hidden_dim×hidden_dim] and the bias b_{msg} [1×hidden_dim], and is finally passed through the ReLU activation function:

$$msg = ReLU((x_j + e) \cdot W\_msg + b\_msg)$$

Thus, the message matrix *msg* of size [E×hidden_dim] contains in its i-th row the message associated with the i-th edge.
In the second phase, the computed messages are aggregated for each source node. Initially, an empty output matrix *m_out* of size [M×hidden_dim] is defined. Subsequently, using a for loop, the index of the source node for each edge is identified, and the message corresponding to the considered edge is added to the row of *m_out* corresponding to the source node index (*start_node*), determined via the edge index matrix *edge_idx*.
The matrix *m_out* therefore contains the embedding of each node resulting from the message passing operation, obtained by aggregating information from neighboring nodes and incident edges.
Returning to the main code, the result of the message passing for the first layer is stored in the variable *delta1*. At this stage, the embedding of each node incorporates information exclusively from adjacent nodes and incident edges. To preserve the original node features and improve training stability and convergence, a residual connection (ResNet) has been introduced. Specifically, the initial node embeddings, represented by the matrix x_1 [Mxhidden_dim], are added to the output of the message passing operation.
In this way, the final node representations output by layer 1 are computed as:

$$m_1 = ReLU(delta_1 + x_1)$$

Layer 2 undergoes a process analogous to that of layer 1. However, in this case, the initial linear transformation takes as input node features the matrix m_1:

$$x_2 = m_1 \cdot W\_node_2 + b\_node_2$$
$$e_2 = batch\_edge\_attr \cdot W\_edge_2 + b\_edge_2$$

The new weights and biases have the same dimensions as in the previous layer, except for $W_{\text{node},2}$, which in this case must have dimensions [hidden_dim×hidden_dim]. The message passing phase and the use of residual connections are carried out in exactly the same manner as in layer 1.
The final matrix of embeddings for all nodes in the batch, m_2, is thus obtained, with size [M×hidden_dim]:

$$m_2 = ReLU(delta_2 + x_2)$$

At this stage, the node information from all graphs is combined into a single matrix, m_2. To obtain an initial graph-level embedding from the node embeddings, aggregation is performed using a global mean pooling operation. For this purpose, the *global_mean_pool* function is implemented:

```
% Global mean pooling function
function graph_embed = global_mean_pool(node_feats, graph_ptr)
    % graph_ptr: vector with number of nodes per graph in the batch
    % node_feats: concatenated node features (total nodes x hidden_dim)
    B = length(graph_ptr);
    graph_embed = zeros(B, size(node_feats,2)); % BxH, store graph embeddings
    start_idx = 1;
    for i=1:B
        n_nodes = graph_ptr(i);
        graph_embed(i,:) = mean(node_feats(start_idx:start_idx+n_nodes-1,:), 1); %
        average node embeddings
```

```
        start_idx = start_idx + n_nodes;
    end
end
```

The function computes the embedding of each graph by averaging the features of its constituent nodes. After initializing the output matrix *graph_embed* of size [B×hidden_dim], a for loop iterates over each graph in the batch, identifying the indices of the corresponding nodes. This is done using a global node counter *start_idx* and the variable *graph_ptr*, which was previously defined in the *prepare_batch* function as a vector of length B specifying the number of nodes in each graph.
The embeddings of the selected nodes belonging to the considered graph are then averaged along the first dimension of the matrix, i.e., the mean is calculated column-wise. In this way, for each graph, a vector of embeddings of size [1×hidden_dim] is obtained, where each i-th element represents the mean of the i-th elements of the node features composing the graph.
The final result is the matrix *graph_embed* of size [B×hidden_dim], where each row constitutes the global embedding of a graph in the batch.
The final output for each batch is obtained through the implementation of an MLP (Multi Layer Perceptron), which is responsible for transforming the normalized graph embeddings into scalar predictions.
The MLP consists of a first layer with a ReLU activation, defined as follows:

$$h = ReLU(graph\_embed \cdot W\_out_1 + b\_out_1)$$

The matrix of graph embeddings $(graph\_embed)$ is linearly transformed using the weight matrix W_{out1} [hidden_dim×hidden_dim] and the bias b_{out1} [1×hidden_dim]. A ReLU activation function is then applied, introducing nonlinearity and enhancing the expressive capacity of the model. The resulting matrix h retains size [B×hidden_dim].
The second layer performs a linear transformation without an activation function, using the weight matrix W_{out2} [hidden_dim×1] and the bias vector b_{out2}, which in this case is a single scalar value:

$$out = h \cdot W\_out_2 + b\_out_2$$

The final output of the forward pass is thus the vector *out* [B×1], whose elements correspond to the normalized predicted values of effective stiffness for each graph in the batch.
At this point, the loss function used for training the neural network is computed, specifically the Mean Squared Error (MSE):

$$MSE = \frac{\sum_{k=1}^{B}(out_k - batch\_y_k)^2}{B}$$

It is defined as the mean of the squared differences between predictions and true values, thus providing a measure of the discrepancy between the model estimates and the observed values. The variable *out* represents the effective stiffness predictions generated by the neural network for the current batch of graphs, while *batch_y* corresponds to the true effective stiffness values associated with that batch, obtained for each structure through FEM analysis. It should be noted, however, that both are normalized values. During the testing phase, the MSE and the coefficient of determination R^2 will instead be

computed on denormalized values, to make them comparable with the actual magnitude of the targets.
The use of MSE as the loss function allows larger errors to be penalized more heavily, facilitating the optimization of the network weights through gradient-based optimization.
The subsequent backpropagation phase involves computing the gradient of the MSE loss function with respect to each parameter θ. For the t-th batch, this is calculated as:

$$grad_t = \frac{\partial(MSE)}{\partial(\vartheta)}\bigg|_{\vartheta_{t-1}}$$

The computed gradient is then used to update the parameters through Adam optimization. For the gradient computation, the backward_pass function is implemented as follows:

```
function grads = backward_pass(batch_x, batch_edge_idx, batch_edge_attr, batch_y,
batch_graph_ptr, ...
    W_node1, b_node1, W_edge1, b_edge1, W_msg1, b_msg1, ...
    W_node2, b_node2, W_edge2, b_edge2, W_msg2, b_msg2, ...
    W_out1, b_out1, W_out2, b_out2, relu, drelu)
    % --- FORWARD PASS (recomputation) ---
    % Layer 1
    x1 = batch_x * W_node1 + b_node1;                  % MxH
    e1 = batch_edge_attr * W_edge1 + b_edge1;          % ExH
    delta1 = message_passing(x1, batch_edge_idx, e1, W_msg1, b_msg1, relu); % MxH
    m1 = relu(x1 + delta1);                            % MxH
    % Layer 2
    x2 = m1 * W_node2 + b_node2;                       % MxH
    e2 = batch_edge_attr * W_edge2 + b_edge2;          % ExH
    delta2 = message_passing(x2, batch_edge_idx, e2, W_msg2, b_msg2, relu); % MxH
    m2 = relu(x2 + delta2);                            % MxH
    % Global mean pooling
    graph_embed = global_mean_pool(m2, batch_graph_ptr); % BxH
    % Output MLP
    h = relu(graph_embed * W_out1 + b_out1);           % BxH
    out = h * W_out2 + b_out2;                         % Bx1

    % --- Derivatives of the MSE loss ---
    B = size(batch_y,1);
    dL_dout = 2*(out - batch_y) / B;                   % Bx1
    % --- Backpropagation of the output layer ---
    dL_dW_out2 = h' * dL_dout;                         % Hx1
    dL_db_out2 = sum(dL_dout,1);                       % 1x1
    dL_dh = dL_dout * W_out2';                         % BxH
    % --- Backprop hidden layer (ReLU) ---
    dL_dz = dL_dh .* drelu(graph_embed * W_out1 + b_out1); % BxH
    dL_dW_out1 = graph_embed' * dL_dz;                 % HxH
    dL_db_out1 = sum(dL_dz,1);                         % 1xH
    dL_dgraph_embed = dL_dz * W_out1';                 % BxH
    % --- Backprop global mean pooling ---
    M = size(m2,1);
    H = size(m2,2);
    B = length(batch_graph_ptr);
    dL_dm2 = zeros(M,H);
    start_idx = 1;
    for i = 1:B
        n_nodes = batch_graph_ptr(i);
```

```
    dL_dm2(start_idx:start_idx+n_nodes-1, :) = repmat(dL_dgraph_embed(i,:) /
                                                n_nodes, n_nodes, 1);
    start_idx = start_idx + n_nodes;
end
% --- Layer 2 ---
dL_after_relu2 = dL_dm2 .* drelu(x2 + delta2);
dL_dx2_from_id = dL_after_relu2;
dL_ddelta2 = dL_after_relu2;
% Backprop message passing 2
xj2 = x2(batch_edge_idx(2,:),:);
msg2_in = (xj2 + e2)*W_msg2 + b_msg2;
msg2 = relu(msg2_in);
dL_dmsg2_in = zeros(size(msg2));
for e = 1:size(batch_edge_idx,2)
    start_node = batch_edge_idx(1,e);
    dL_dmsg2_in(e,:) = dL_ddelta2(start_node,:) .* drelu(msg2_in(e,:));
end
dL_dW_msg2 = (xj2 + e2)' * dL_dmsg2_in;
dL_db_msg2 = sum(dL_dmsg2_in,1);
dL_dxj2_from_msg = dL_dmsg2_in * W_msg2';
dL_de2 = dL_dmsg2_in * W_msg2';
dL_dx2_from_msg = zeros(size(x2));
for e = 1:size(batch_edge_idx,2)
    end_node = batch_edge_idx(2,e);
    dL_dx2_from_msg(end_node,:) = dL_dx2_from_msg(end_node,:) +
                                  dL_dxj2_from_msg(e,:);
end
dL_dx2_total = dL_dx2_from_msg + dL_dx2_from_id;
dL_dW_node2 = m1' * dL_dx2_total;
dL_db_node2 = sum(dL_dx2_total,1);
dL_dm1 = dL_dx2_total * W_node2';
% --- Layer 1 ---
dL_after_relu1 = dL_dm1 .* drelu(x1 + delta1);
dL_dx1_from_id = dL_after_relu1;
dL_ddelta1 = dL_after_relu1;
% Backprop message passing 1
xj1 = x1(batch_edge_idx(2,:),:);
msg1_in = (xj1 + e1)*W_msg1 + b_msg1;
msg1 = relu(msg1_in);
dL_dmsg1_in = zeros(size(msg1));
for e = 1:size(batch_edge_idx,2)
    start_node = batch_edge_idx(1,e);
    dL_dmsg1_in(e,:) = dL_ddelta1(start_node,:) .* drelu(msg1_in(e,:));
end
dL_dW_msg1 = (xj1 + e1)' * dL_dmsg1_in;
dL_db_msg1 = sum(dL_dmsg1_in,1);
dL_dxj1_from_msg = dL_dmsg1_in * W_msg1';
dL_de1 = dL_dmsg1_in * W_msg1';
dL_dx1_from_msg = zeros(size(x1));
for e = 1:size(batch_edge_idx,2)
    end_node = batch_edge_idx(2,e);
    dL_dx1_from_msg(end_node,:) = dL_dx1_from_msg(end_node,:) +
                                  dL_dxj1_from_msg(e,:);
end
dL_dx1_total = dL_dx1_from_msg + dL_dx1_from_id;
dL_dW_node1 = batch_x' * dL_dx1_total;
dL_db_node1 = sum(dL_dx1_total,1);
% --- Gradients of the edge layers ---
dL_dW_edge2 = batch_edge_attr' * dL_de2;
```

```
    dL_db_edge2 = sum(dL_de2,1);
    dL_dW_edge1 = batch_edge_attr' * dL_de1;
    dL_db_edge1 = sum(dL_de1,1);
    % --- Final output ---
    grads = {dL_dW_node1, dL_db_node1, dL_dW_edge1, dL_db_edge1, dL_dW_msg1,
             dL_db_msg1,dL_dW_node2, dL_db_node2, dL_dW_edge2, dL_db_edge2,
             dL_dW_msg2, dL_db_msg2,dL_dW_out1, dL_db_out1, dL_dW_out2,
             dL_db_out2};
end
```

The procedure involves two main phases:

1. Forward pass (recalculation): the forward propagation of data through the node, edge, and message layers, the global pooling, and the final MLP is recalculated, including the ReLU activations and batch normalization.

2. Backpropagation via the chain rule: the loss gradients are propagated backward through all layers of the network, following the chain rule. This includes backpropagation through the output MLP, global pooling, residual connections, and the two message passing layers, down to the weights and biases of the nodes, edges, and messages.

In this way, the function returns all the gradients necessary to update the network parameters during optimization.
The code then proceeds with the update of the network weights using the ADAM (Adaptive Moment Estimation) algorithm.
For each network parameter, at the t-th batch, it involves the computation of two variables:

- The first moment m, defined as:

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_2) \cdot grad_t$$

  It represents the exponential moving average of the gradients up to step t; it is an estimate of the average gradient direction, that is, the direction in which the parameters should be updated.

- The second moment v, defined as:

$$v_t = \beta_1 \cdot v_{t-1} + (1 - \beta_2) \cdot {grad_t}^2$$

  represents the exponential moving average of the squared gradients. It is an estimate of the gradient variance, that is, how much the gradients vary in magnitude, and it helps to dynamically adapt the update step, reducing the impact of very large gradients.

The variables β_1 and β_2 are the exponential decay rates mentioned previously in the parameter definitions. As noted, they influence the weight that past moment values have on future estimates; high values of β_1 and β_2 reduce the impact of new gradients on the moment estimates.

Since the moments are initialized as $m_0 = v_0 = 0$, the moment estimates are initially biased towards zero, particularly in the first steps of training. This results in parameter update steps that are too small.
To compensate for this initial bias, the following correction is applied:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \quad ; \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

These bias-corrected quantities are then used in the parameter update.
The parameter θ_t is updated according to:

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon_{adam}}$$

where:

- η is the learning rate,
- ϵ_{adam} is a small numerical value for stability,
- $\hat{m}_t$ and $\hat{v}_t$ are the bias-corrected moments.

It should also be noted that the dimensions of the gradients and moments match those of the corresponding parameter.
The newly computed parameters are then stored in the params cell.

2.2.5.4 Evaluation on Test Set

At the end of each training epoch, the neural network is evaluated on the test set to estimate the general performance of the MPNN.
The procedure follows the steps of the forward pass, identical to those used during training:

```
% — Evaluate on test set at the end of the epoch —
    num_test = length(test_data.targets);
    num_test_batches = ceil(num_test / batch_size);
    all_y_true = [];
    all_y_pred = [];
    for tb = 1:num_test_batches
        % Extract test batch
        idx_tb = (tb-1)*batch_size + (1:batch_size);
        idx_tb = idx_tb(idx_tb <= num_test);
        [tx, te_idx, te_attr, ty, tptr] = prepare_batch(test_data, idx_tb);
        % Forward pass identical to training (no gradient computation)
        x1t = tx * W_node1 + b_node1;
        e1t = te_attr * W_edge1 + b_edge1;
        delta1t = message_passing(x1t, te_idx, e1t, W_msg1, b_msg1, relu);
        m1t = relu(delta1t + x1t);
        x2t = m1t * W_node2 + b_node2;
        e2t = te_attr * W_edge2 + b_edge2;
        delta2t = message_passing(x2t, te_idx, e2t, W_msg2, b_msg2, relu);
        m2t = relu(delta2t + x2t);
        gem = global_mean_pool(m2t, tptr);
        ht = relu(gem*W_out1 + b_out1);
```

```
        out_t = ht*W_out2 + b_out2;       % normalized predictions
        all_y_true = [all_y_true; ty];
        all_y_pred = [all_y_pred; out_t];
    end
    % Denormalize
    y_true_real = all_y_true * std_target + mean_target;
    y_pred_real = all_y_pred * std_target + mean_target;
    % Compute denormalized MSE and R^2
    mse_test = mean((y_true_real - y_pred_real).^2);
    SS_res = sum((y_true_real - y_pred_real).^2);
    SS_tot = sum((y_true_real - mean(y_true_real)).^2);
    R2 = 1 - SS_res / SS_tot;
    fprintf('Epoch %d/%d – Test MSE = %.4f – R^2 = %.4f\n', epoch, num_epochs,
            mse_test, R2);
```

In this case as well, the graphs are processed in batches, maintaining the same batch size used during training. Once the forward pass is completed and the predictions of the normalized effective stiffness values are obtained, these values are denormalized, along with the corresponding true effective stiffness values. Subsequently, both are used for the computation of the evaluation metrics:

- Mean Squared Error (MSE):

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_{\text{true\_real},i} - y_{\text{pred\_real},i})^2$$

- Coefficient of determination R^2:

$$R^2 = 1 - \frac{\sum_i (y_{\text{true\_real},i} - y_{\text{pred\_real},i})^2}{\sum_i (y_{\text{true\_real},i} - \bar{y}_{\text{true\_real}})^2}$$

Their values are displayed in the command window at the end of each epoch, allowing the monitoring of their progression during training.
At the end of all epochs, the one that yielded the best prediction values is identified, and the results are saved:

```
    if R2 > best_R2
        best_R2 = R2;
        best_epoch = epoch;
        best_y_true_real = y_true_real;
        best_y_pred_real = y_pred_real;
    end
```

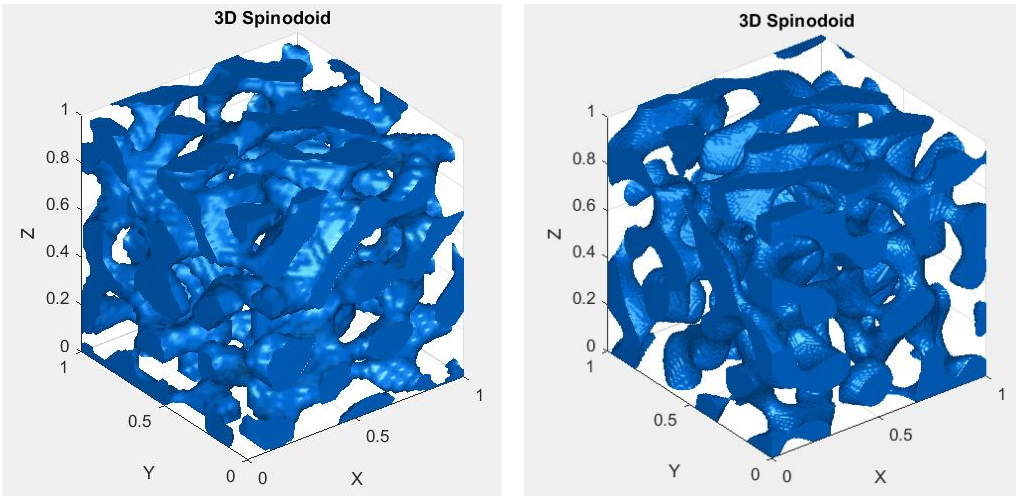
The results obtained are shown in Section 3.5 – MPNN results.
The same MPNN was also implemented with the addition of a third message-passing layer. To achieve this, it was necessary to include the parameters (weights and biases) of the third layer, update the forward pass by adding one more message-passing layer before the global mean pooling, and modify the computation of the gradients of the Loss Function. The results of this MPNN are also presented in Section 3.5 – MPNN results.
Finally, a general discussion on the accuracy of the overall method presented in the thesis is provided in Section 4.3 – Model Accuracy and Future Enhancements.

3. RESULTS

3.1 Generation and characterization of 3D structures

3.1.1 Effect of Generation Parameters on the Obtained Structures

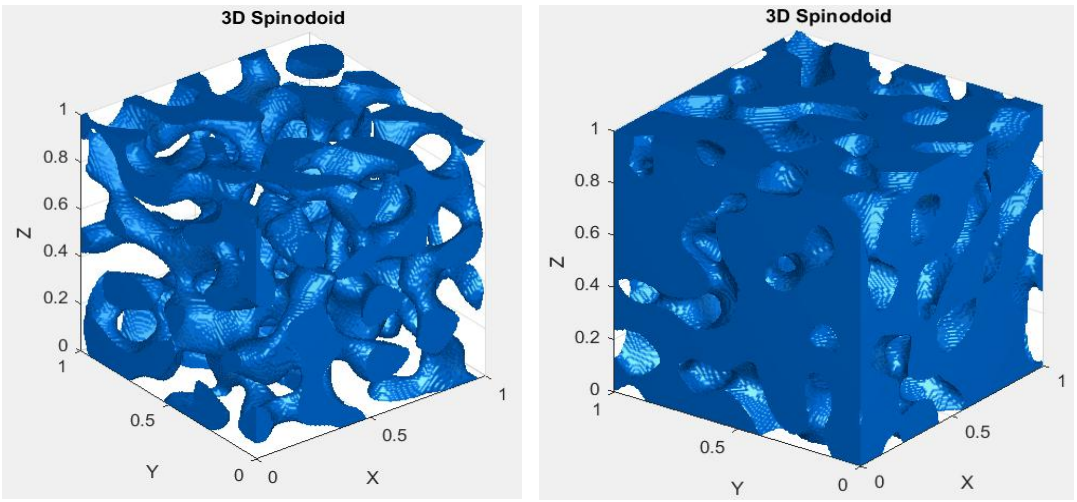
This paragraph shows the effects of varying the generation parameters on the results of the obtained microstructures.
The grid resolution allows for increasing or decreasing the level of detail of the resulting structure by refining the sampling of the scalar field on which the structure is defined. Below are two figures of structures obtained using the previously described method and plotted using the isosurface and isocaps commands. These structures share the same generation parameters (they differ only due to the randomness in the creation of the static waves) but were generated with different grid resolutions:



a) b)

Figure 3.1.1: Spinodoid generated with linear resolution of 50 (a) and 150 (b)

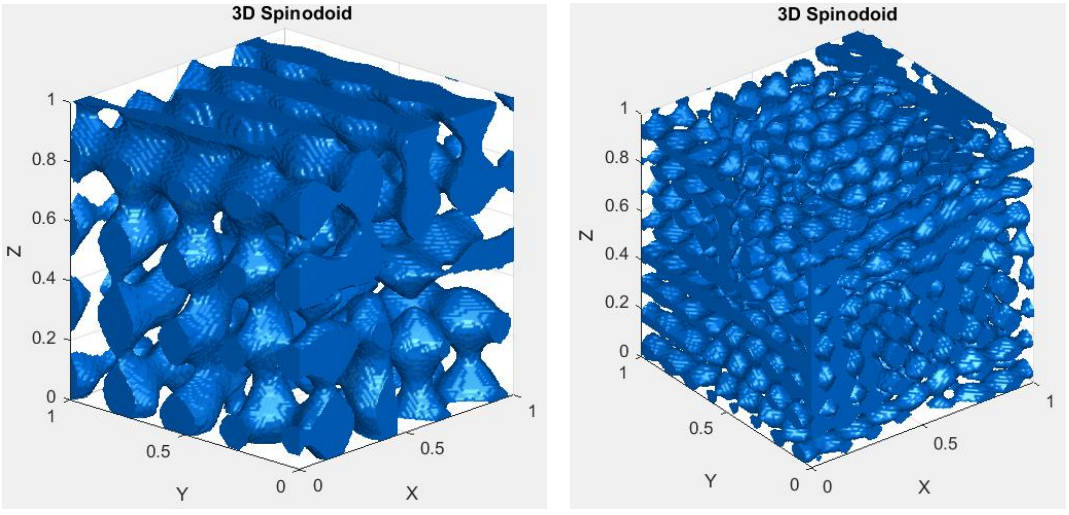
The relative density (ρ) of the structure determines how much of the total volume is occupied by the solid material and how much is left as void:



a) b)

Figure 3.1.2: Spinodoid generated with relative density o 0.3 (a) and 0.7 (b)

The wavelength (λ), and consequently the corresponding wavenumber, determines the scale of variation of the structural features, resulting in a more or less complex and detailed structure.
As the wavelength decreases, the alternation between solid and void regions becomes more frequent, leading to smaller morphological features:



a) b)

Figure 3.1.3: Spinodoid generated with waves of wavelength 4 (a) and 8 (b)

The limiting angles (ϑ) allow the creation of various types of spinodoid structures, among which the main ones are lamellar, columnar, cubic, and isotropic:

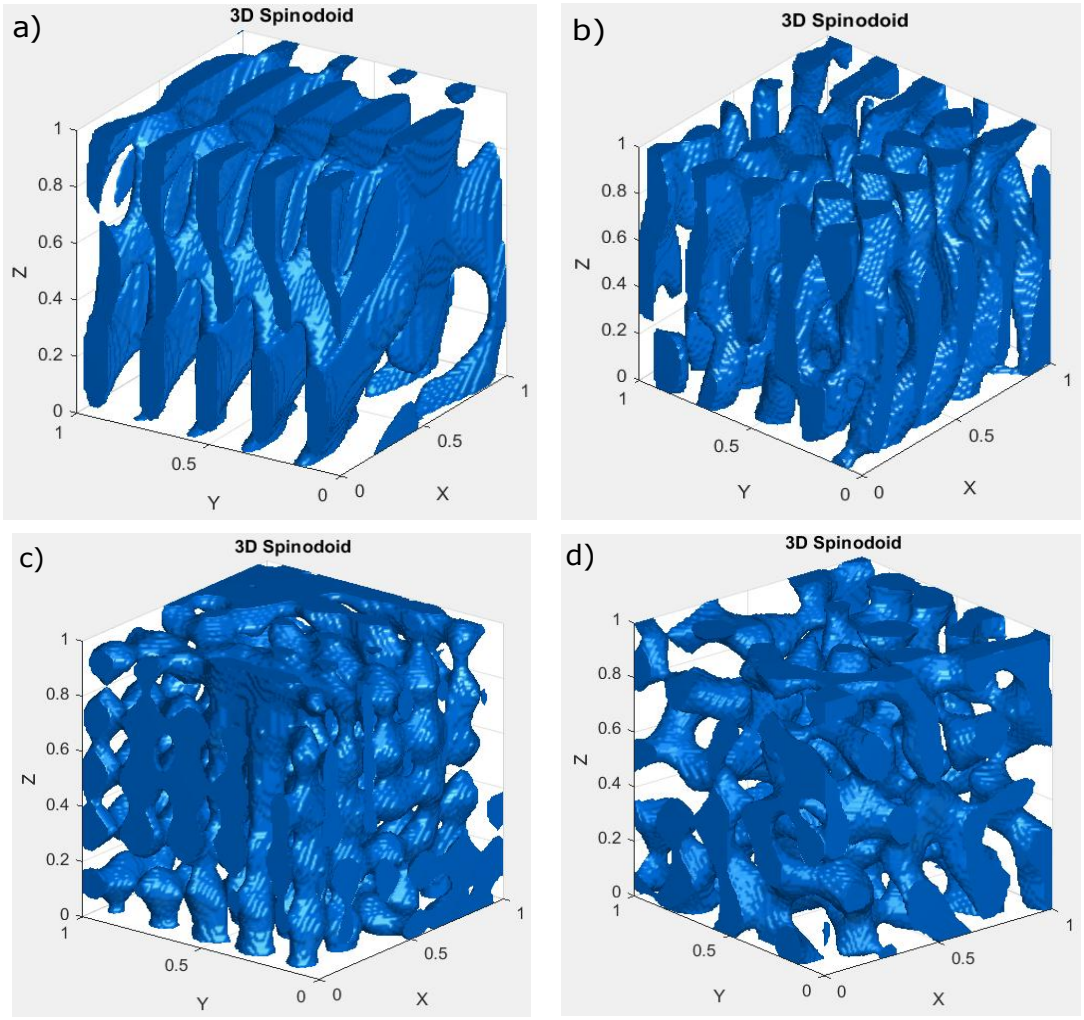


Figure 3.1.4: Lamellar structure obtained with limiting angles of $\boldsymbol{\vartheta_1 = 25°, \vartheta_2 = 0°, \vartheta_3 = 0°}$ (a), columnar $\boldsymbol{\vartheta_1 = 25°, \vartheta_2 = 25°, \vartheta_3 = 0°}$ (b), cubic $\boldsymbol{\vartheta_1 = 15°, \vartheta_2 = 15°, \vartheta_3 = 15°}$ (c), isotropic $\boldsymbol{\vartheta_1 = 90°, \vartheta_2 = 90°, \vartheta_3 = 90°}$

3.1.2 Structure After Removal of Disconnected Parts

The final result after removing the detached parts using the method described in Section 2.2.1.2 – Removing detached parts is shown in the following figure:

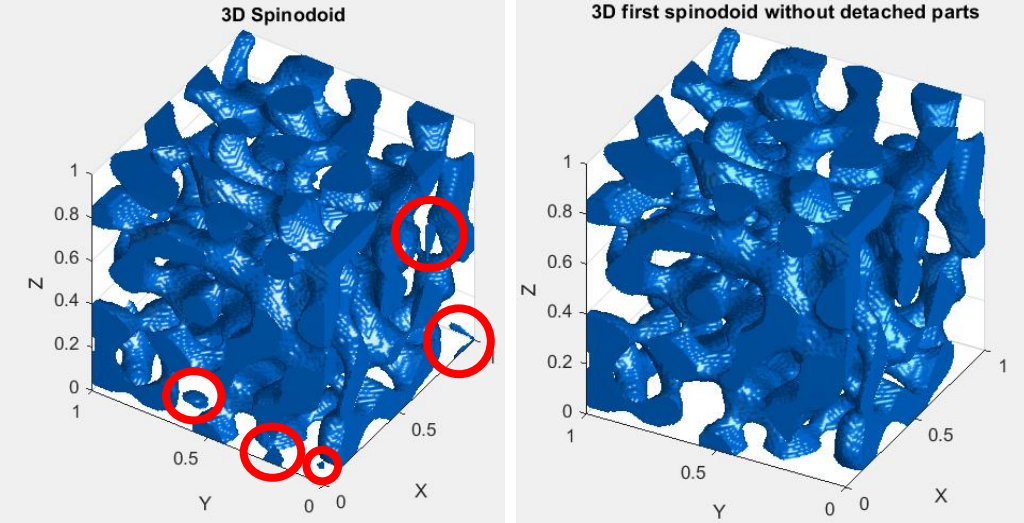


Figure 3.1.5: Structure before and after removing isolated components

3.1.3 Structure Skeleton After Thinning Algorithm

The following shows the plot of the skeleton of a structure obtained from the thinning process exposed in Section 2.2.1.3 – Thinning algorithm (skeletal graph). In this representation, each node corresponds to a skeletal point with known spatial coordinates, and each edge represents a connection between two nearby points:

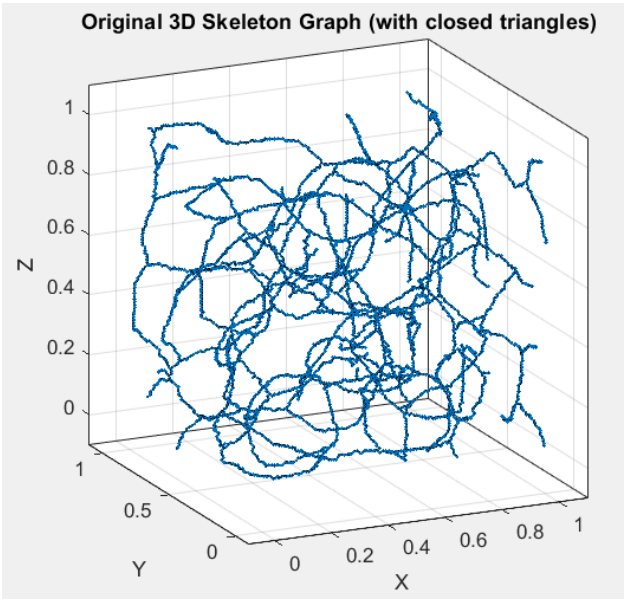


Figure 3.1.6: 3D skeleton of the spinodoid structure

3.1.4 Skeleton of the Structure After Removing Closed Triangulations

The skeleton after the algorithm that removes the small triangulations exposed in Section 2.2.1.4 – Removing small triangulations, appears as shown in the figure:

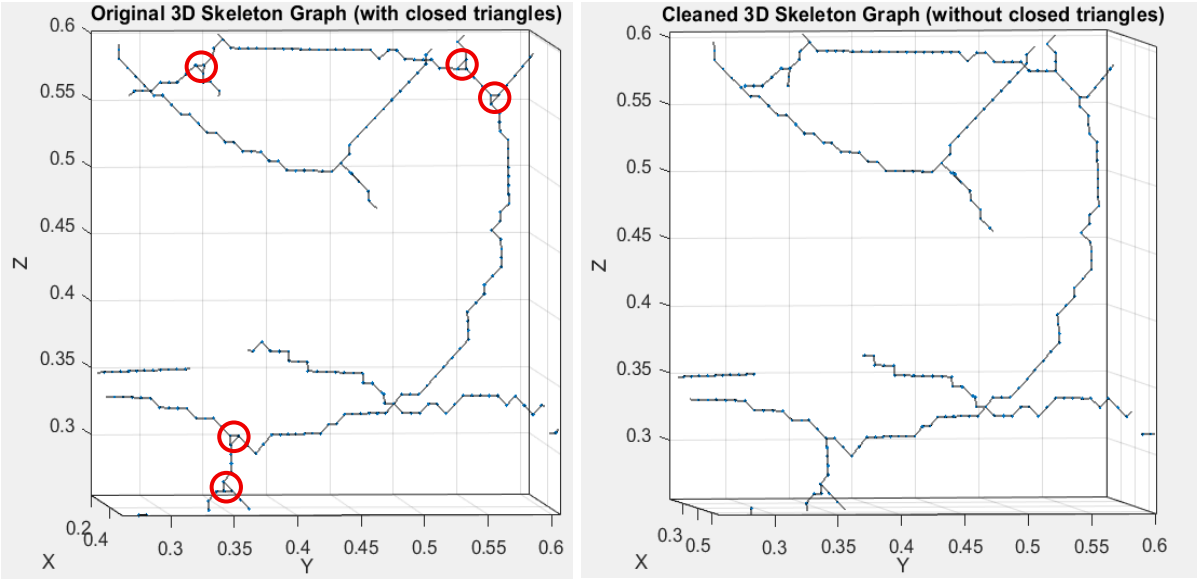


Figure 3.1.7: Zoom on skeleton before and after removing closed loop triangles

3.1.5 Reconstructed Structure Using Constant Radius Approximation

The result of the structure reconstruction with constant-radius approximation exposed in Section 2.2.1.5 – First thickness analysis (constant radius reconstruction), is shown below:

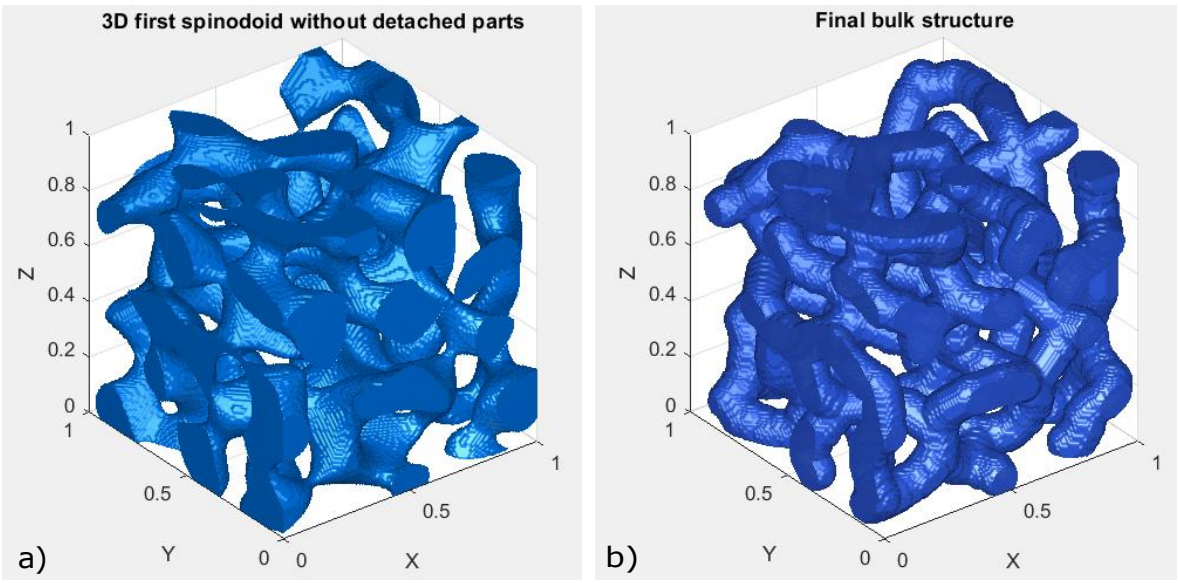


Figure 3.1.8: Original (a) and reconstructed structure (b)

3.1.6 Geometric and morphological characterization

3.1.6.1 Node Degree Distribution

The result of the node degree analysis exposed in Section 2.2.1.6.1 – Node degree, is presented in the bar chart below:

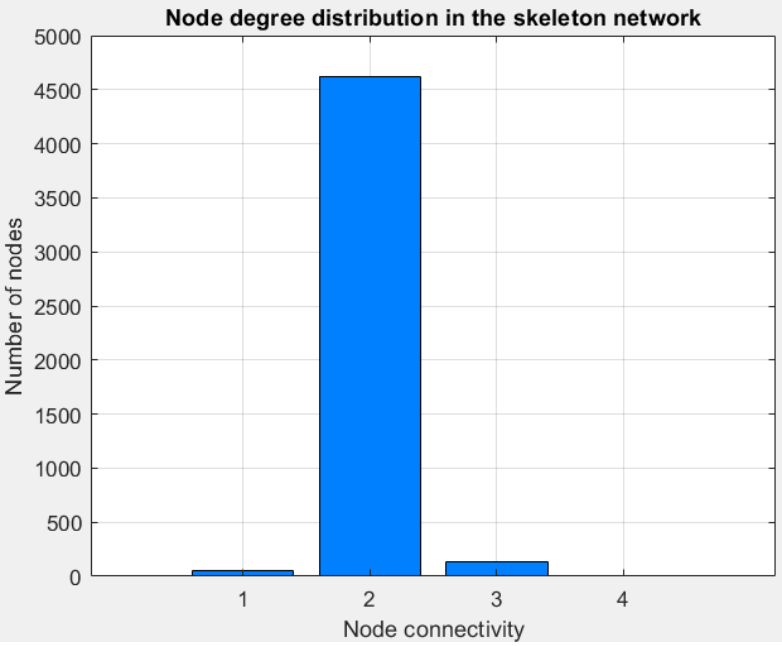


Figure 3.1.9: Bar chart of node degree distribution

3.1.6.2 Branch Chain Length Distribution

The distribution of branch chain lengths computed as exposed in Section 2.2.1.6.3 – Branch chain analysis, is shown in the bar chart below:

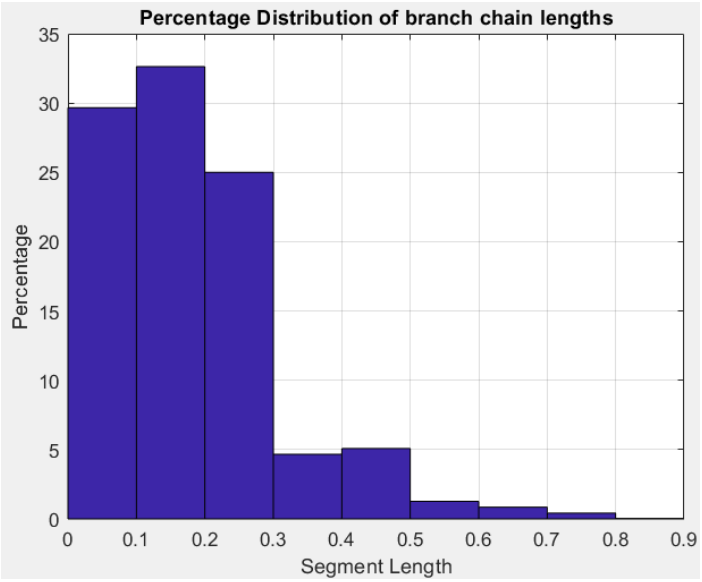


Figure 3.1.10: Bar chart of the branch chain length distribution

3.1.6.3 Probability Density Distribution of Skeleton Branch Orientation

The result of the branch orientation analysis along the skeleton, exposed in Section 2.2.1.6.4 – Segments orientation, is shown for the example case of an isotropic structure:

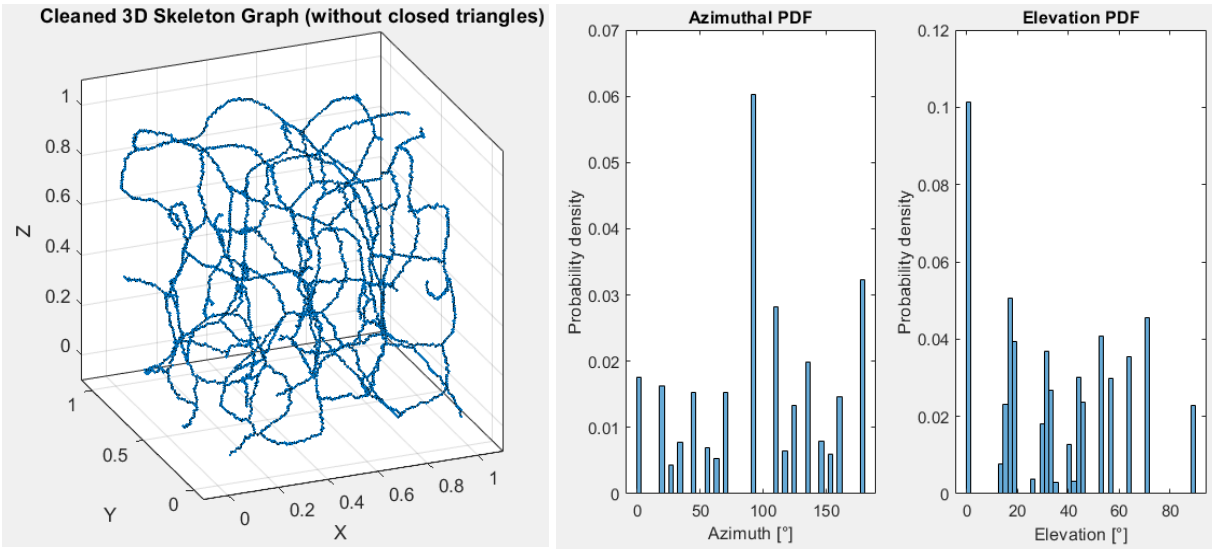


Figure 3.1.11: Probability density histograms of azimuthal and elevation orientations along branch chains for an isotropic spinodoid structure

The result is also shown for the case of a cubic spinodoid structure:

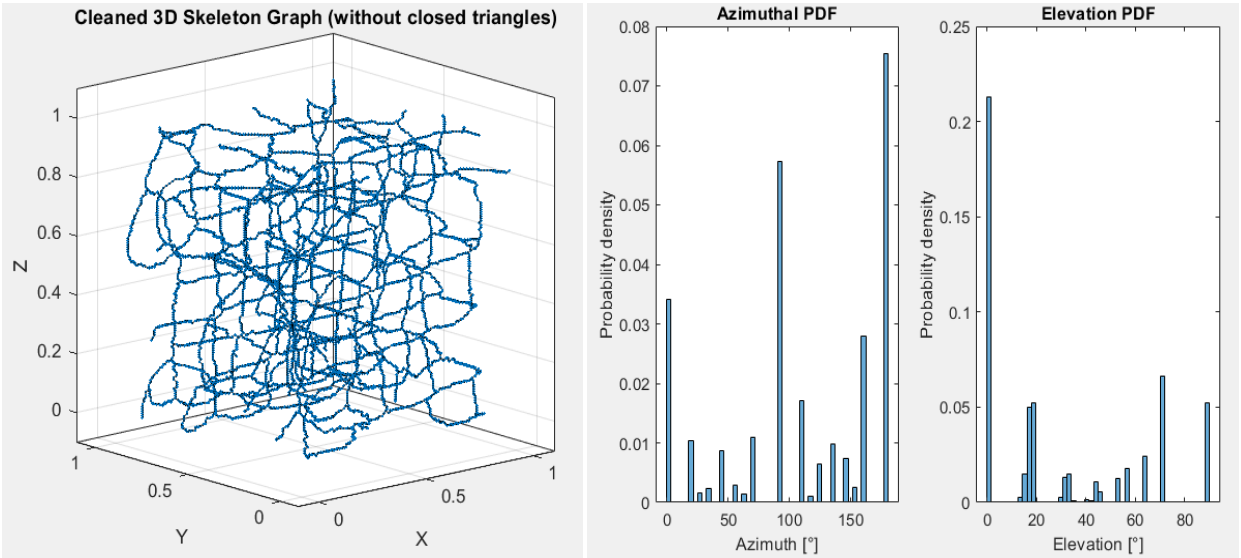


Figure 3.1.12: Probability density histograms of azimuthal and elevation orientations along branch chains for a cubic spinodoid structure

In the case of a columnar spinodoid structure, the following is obtained:

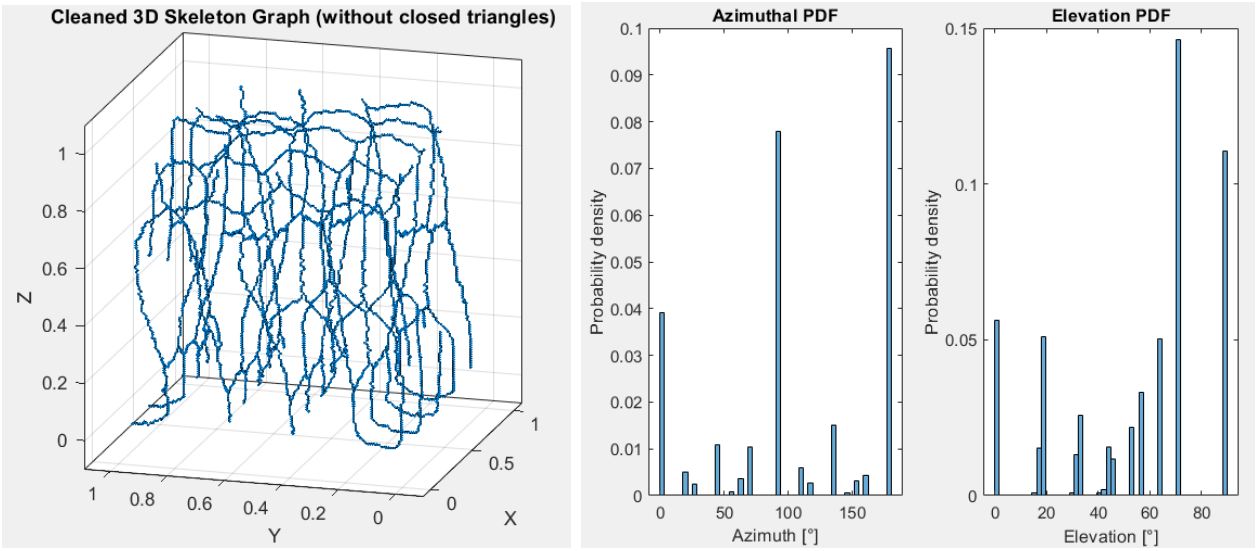


Figure 3.1.13: Probability density histograms of azimuthal and elevation orientations along branch chains for a columnar spinodoid structure $\vartheta_1 = 15°, \vartheta_2 = 15°, \vartheta_3 = 0°$

The results are discussed in Section 4.1.1.1 – Relationship Between Structure Type and Distribution of Preferred Branch Orientations.

3.1.6.4 Cross-Section Profile of the Structure Along the Skeleton

The study of the cross-sectional profile, exposed in Section 2.2.1.6.5 – Thickness analysis, along the skeleton chains for an isotropic structure yielded the following results:

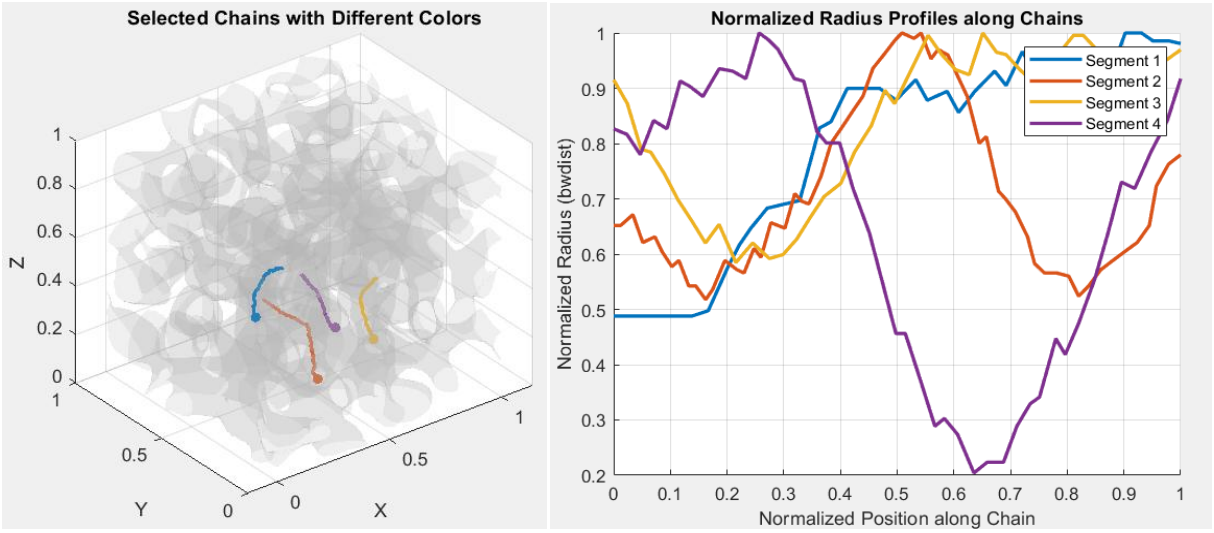


Figure 3.1.14: thickness profile (radius of the largest inscribed sphere) along selected chains for isotropic structure

To facilitate the verification of the obtained results, zoomed views of the various branches of the structure are provided, allowing a visual comparison of the thickness trends with those shown in the graph:

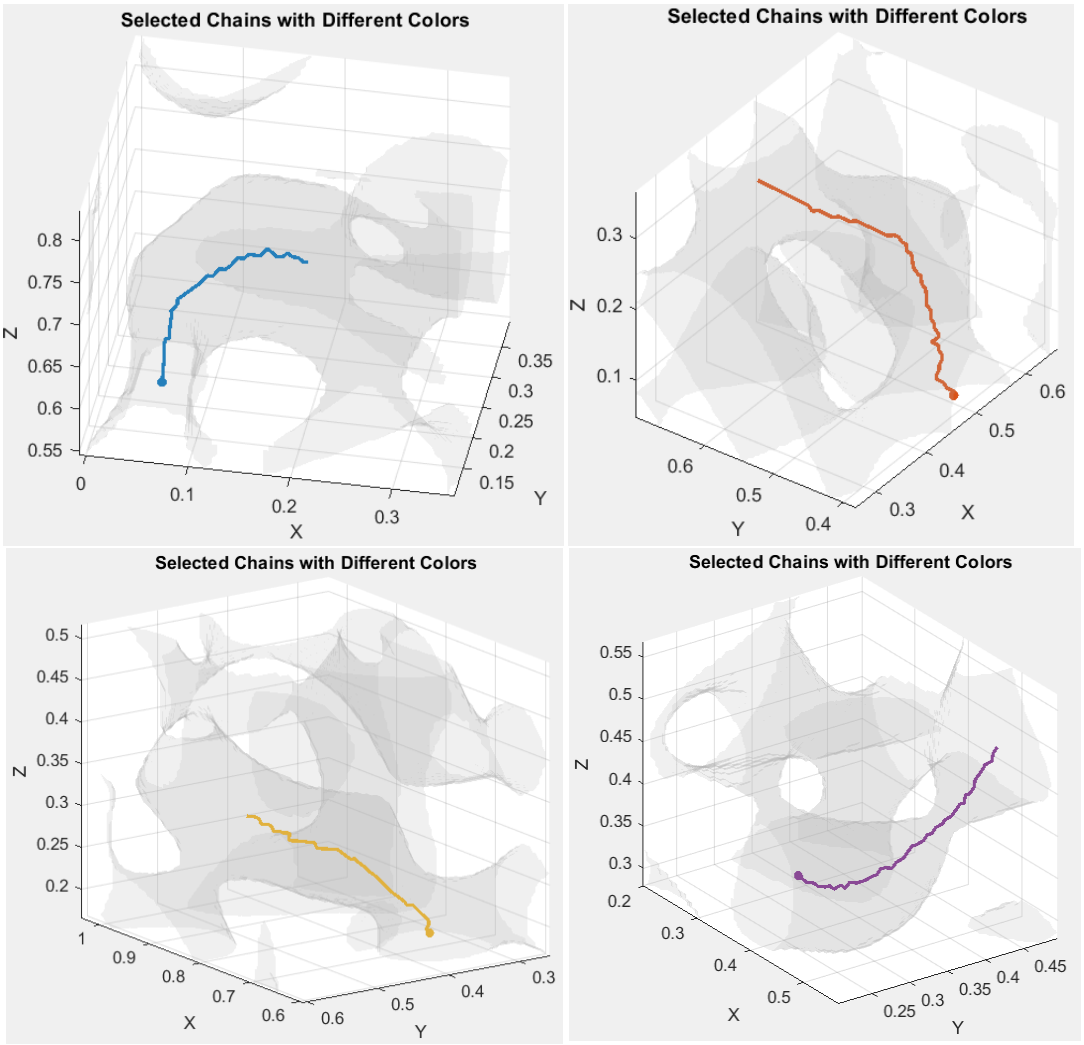


Figure 3.1.15: Zoom on selected chains

For completeness, the case of a cubic spinodoid structure is also shown, in which most of the branch chains exhibit the typical parabolic thickness profile between consecutive branch points:

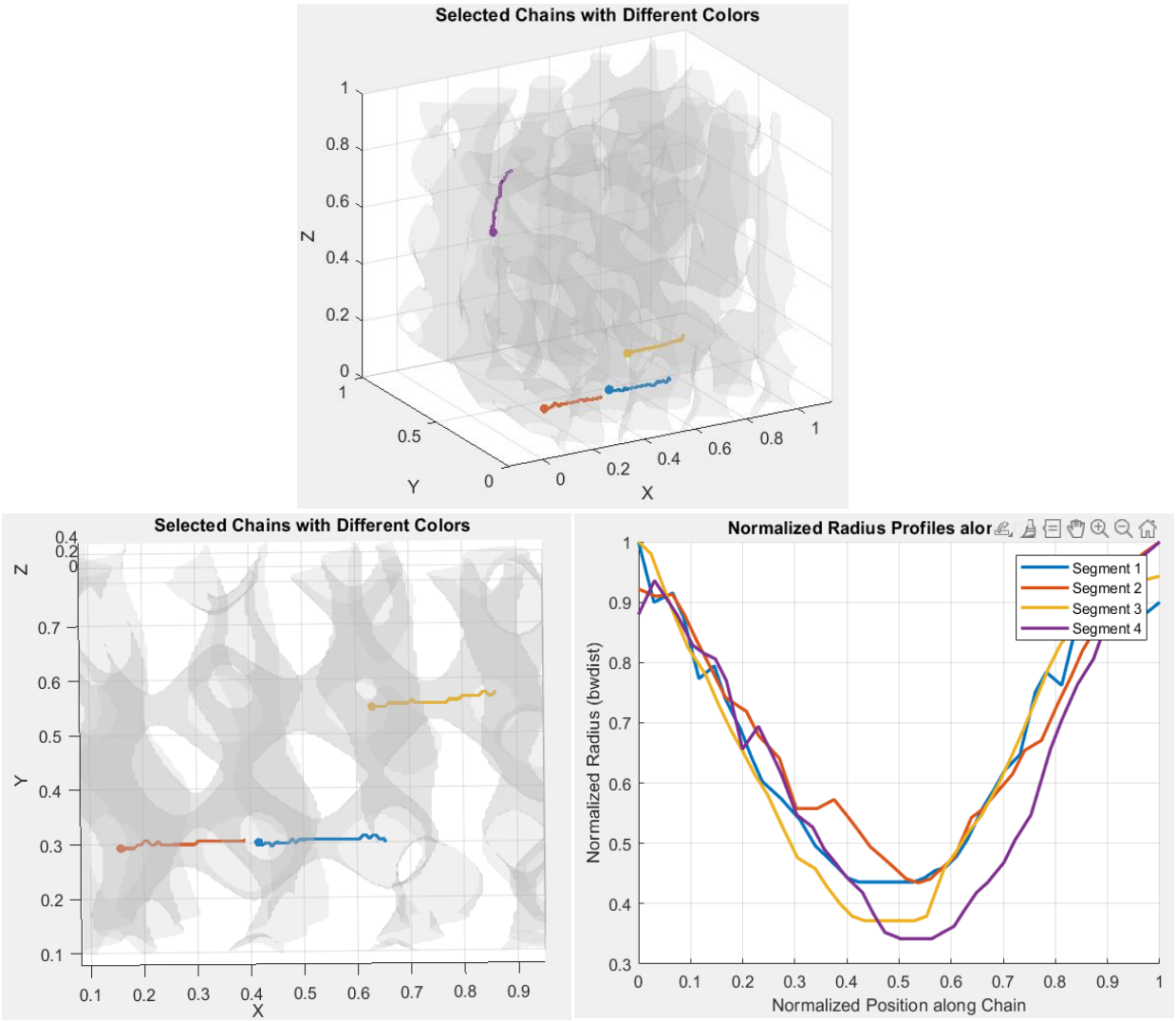


Figure 3.1.16: Thickness profile (radius of the largest inscribed sphere) along selected chains for cubic structure

3.1.7 Examples of Obtainable Hierarchical Structures

The following images show various examples of hierarchical structures, obtained with the method exposed in Section 2.2.1.7 – Hierarchical structures, by combining different limiting angles of the primary and secondary structures:

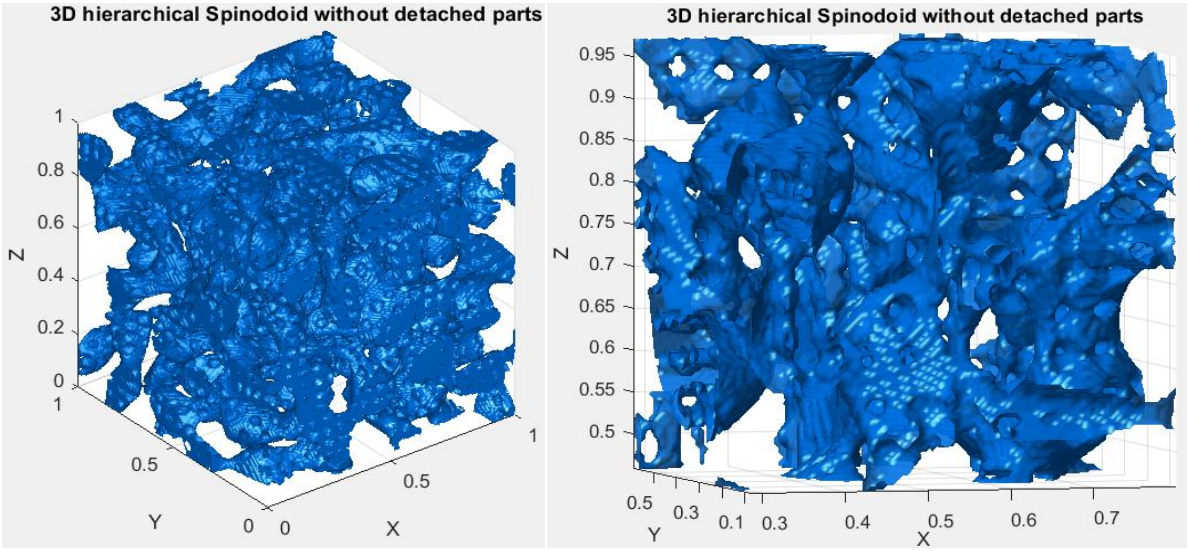


Figure 3.1.17: Hierarchical spinodoid with isotropic primary structure filled with cubic secondary structure

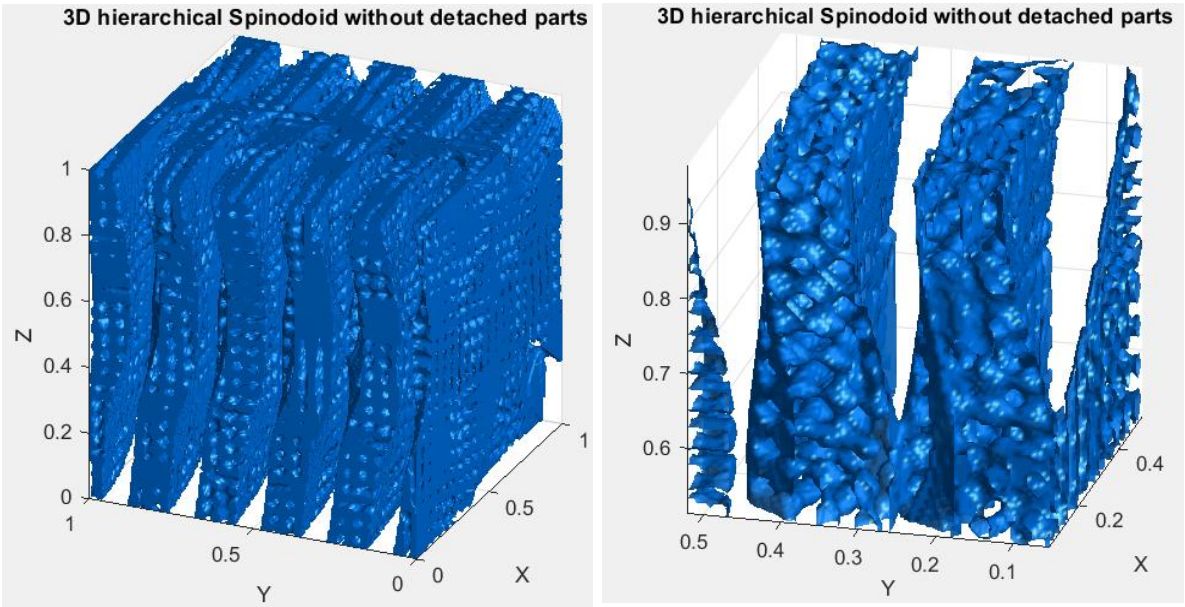


Figure 3.1.18: Hierarchical spinodoid with lamellar primary structure filled with cubic secondary structure

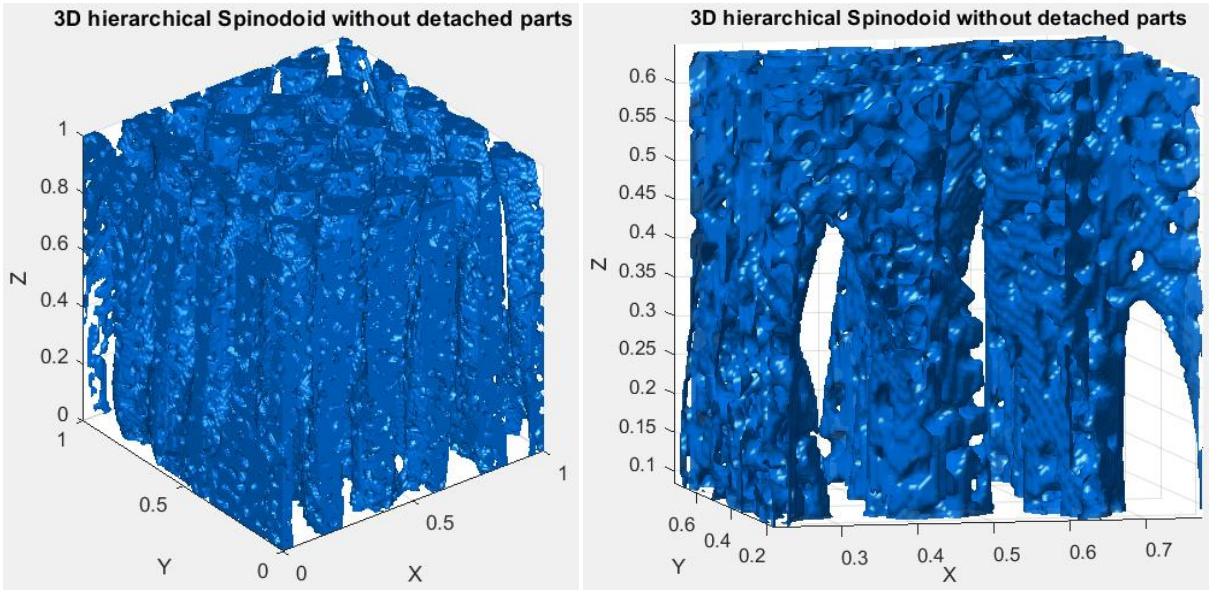


Figure 3.1.19: Hierarchical spinodoid with columnar primary structure filled with isotropic secondary structure

3.1.8 Voxelized Representation of the Structure Through Python File

The result of the voxelized structure, obtained by merging multiple cubic parts as exposed in Section 2.2.1.8 – Preliminary voxelization test, is shown below:

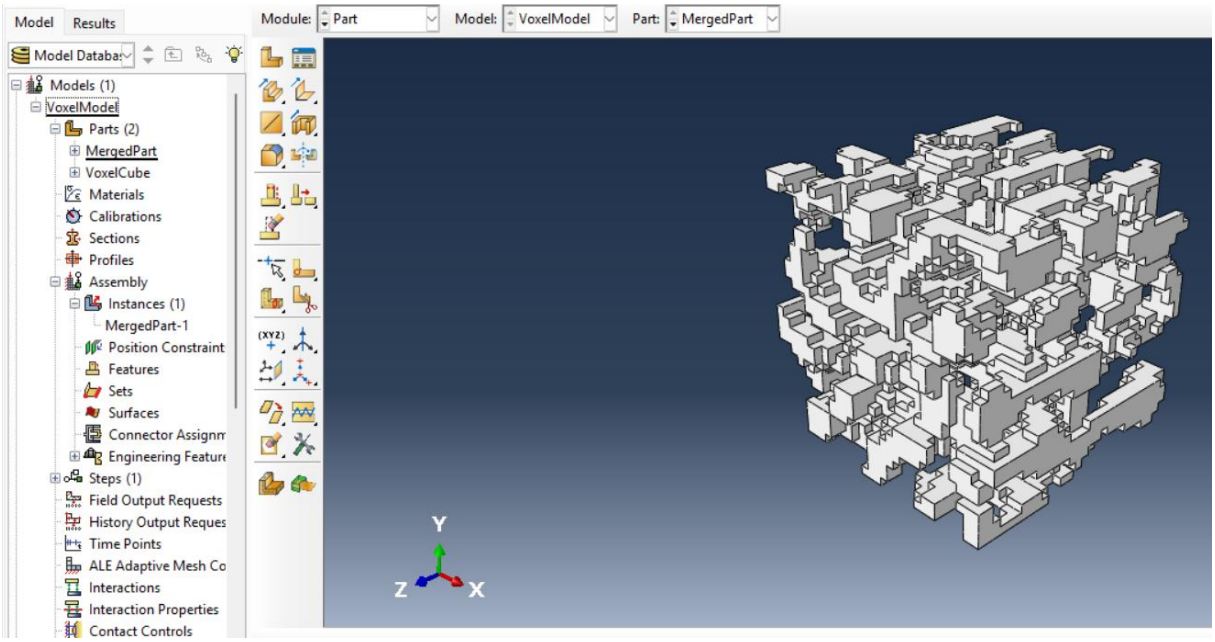


Figure 3.1.20: Voxelised structure obtained with Python script

3.2 FEM Analysis Results

Once the analysis has been performed, as exposed in Section 2.2.2 – FEM analysis (3D), it is possible to visualize the deformed structure and the stress distribution within it:

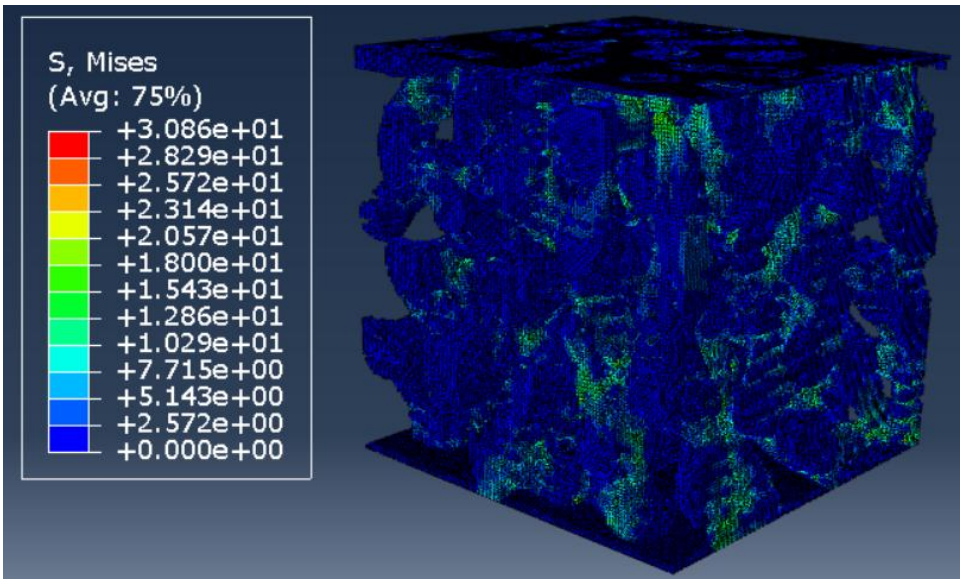


Figure 3.2.1: Stress distribution in deformed structure

Thanks to the data saved in the history output, the Force–Displacement curve can be obtained, and consequently, it is also possible to calculate the effective stiffness of the spinodoid structure (under uniaxial compression).

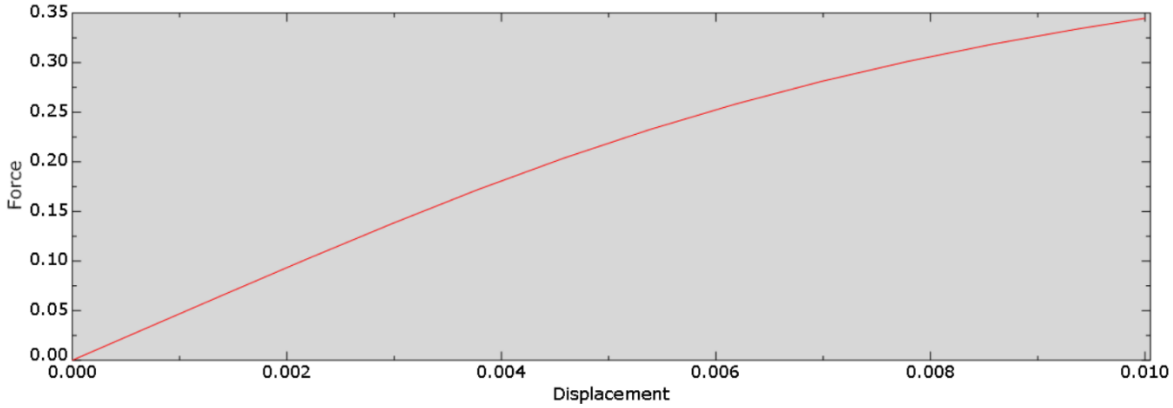


Figure 3.2.2: Force(N) /Displacement(mm) curve

The steps to convert the Force–Displacement curve into a Stress–Strain curve and subsequently calculate the effective stiffness will be explained in detail for 2D structures (Section 2.2.4.8 – Computation of effective stiffness), but they can be applied in an analogous manner, with some adjustments, to 3D structures as well.

3.3 Examples of Obtainable 2D Hierarchical Structures

Several examples of structures obtainable (as exposed in Section 2.2.3 – Transition to the 2D case) by combining isotropic, square-symmetry, or lamellar patterns in the primary and secondary structures, through different combinations of the limiting angles ϑ, are shown below:

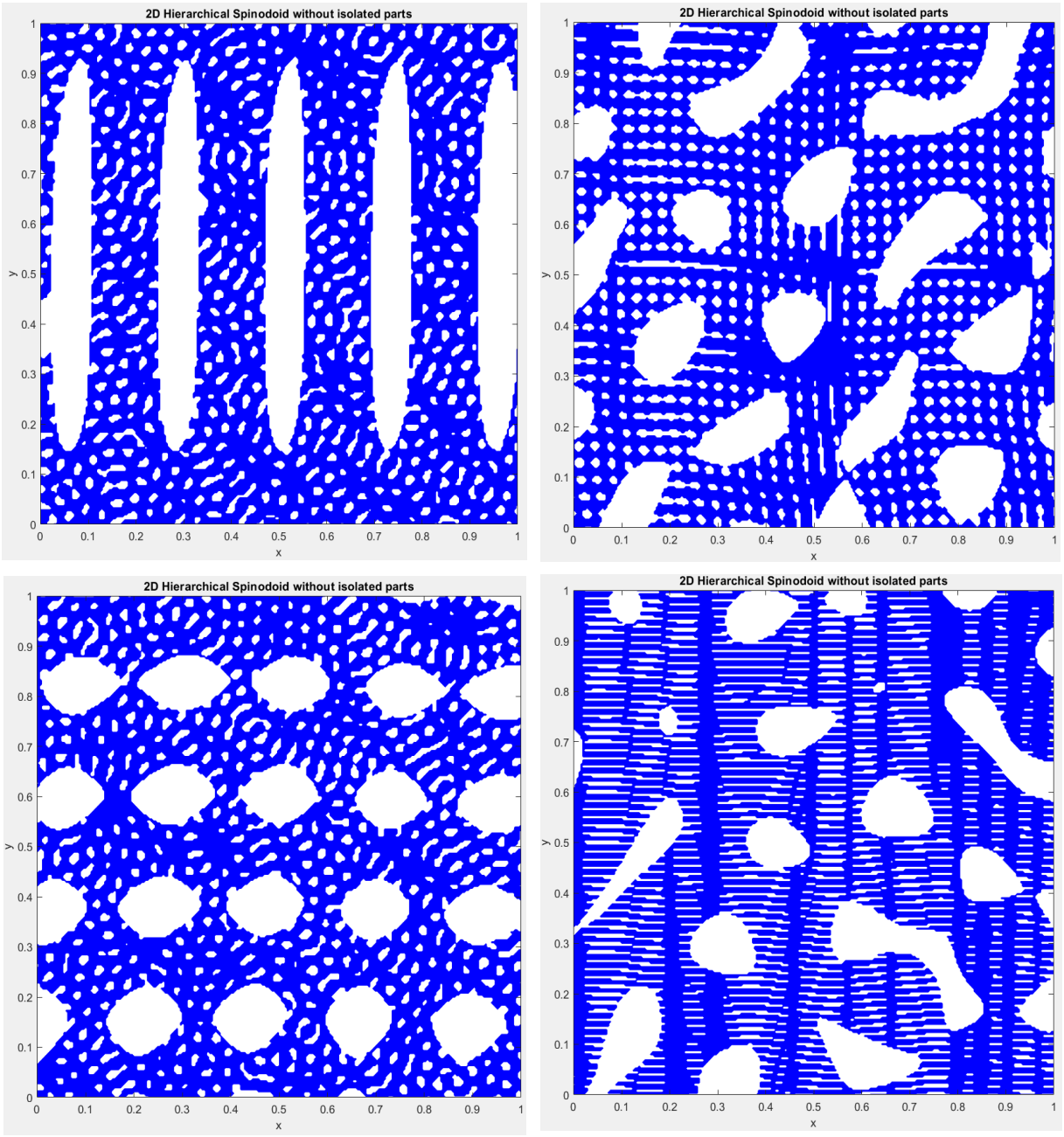
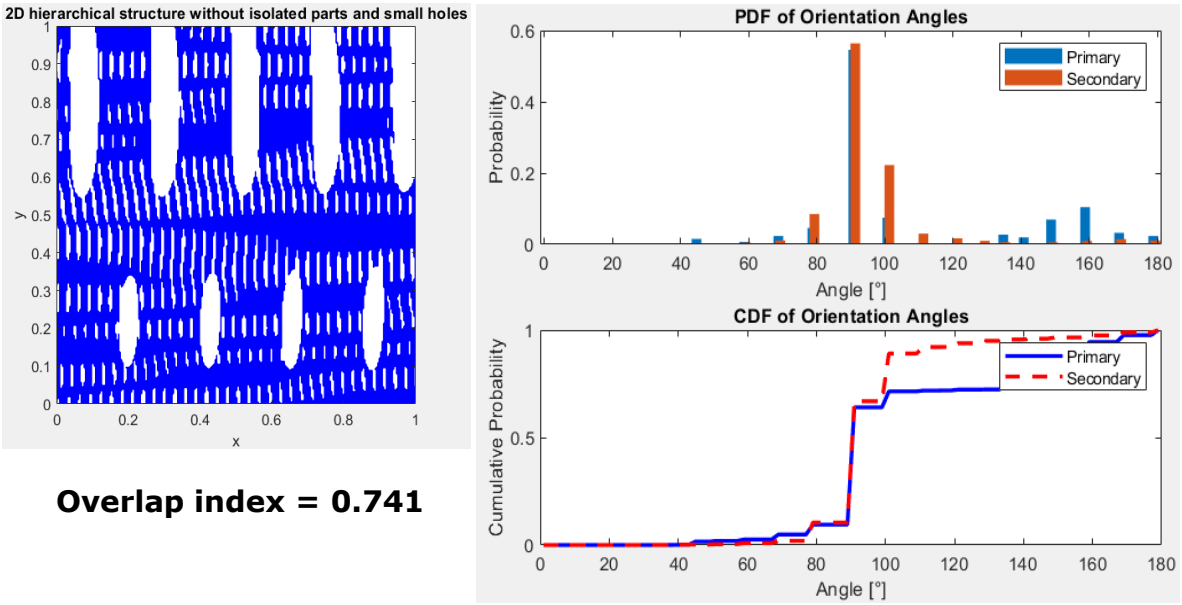


Figure 3.3.1: Examples of 2D hierarchical spinodoid structures

3.4 Hierarchical 2D lamellar spinodoids

3.4.1 Examples of Misalignment Analysis based on the Overlap Index

To further illustrate the significance of the overlap index (introduced in Section 2.2.4.6 – Misalignment index between primary and hierarchical structure), two notable cases are presented, namely those in which the lamellae of the primary and secondary structures are oriented either parallel or orthogonal to each other.
In the first case, for example, a structure where $\vartheta_1 = \vartheta_2 = 90°$ is analyzed; the following results are obtained:

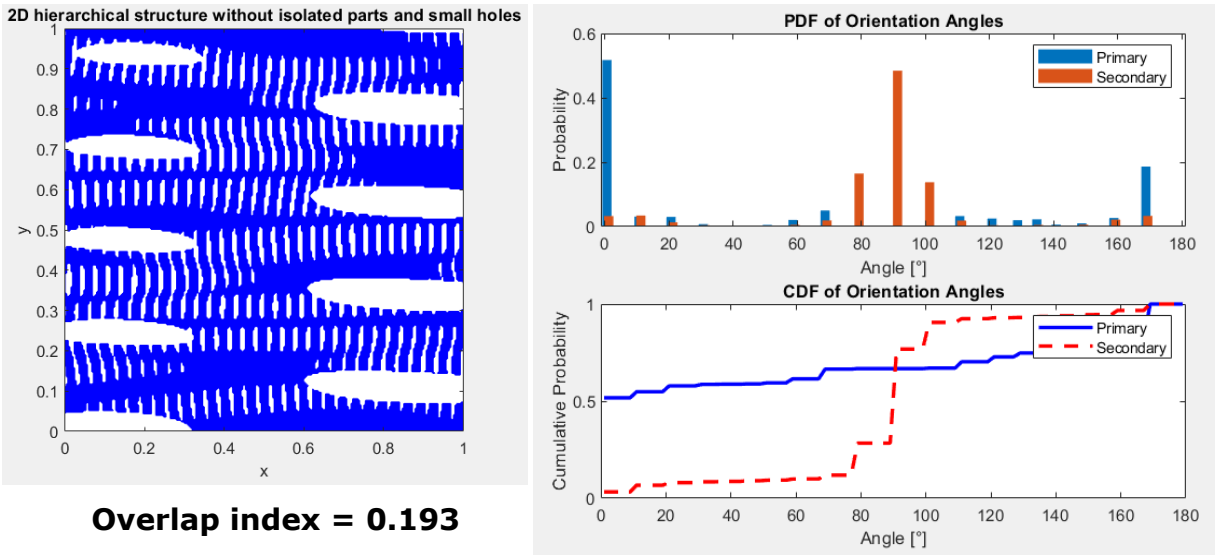


Overlap index = 0.741

Figure 3.4.1: Analysis of the misalignment between primary and secondary structure (case 1)

The image of the structure clearly shows that the lamellae of both the primary and secondary structures are aligned along the y-axis direction.
Since the theta angles are equal, the PDFs, and consequently the CDFs as well, exhibit similar trends, and the overlap index assumes a relatively high value of approximately 75%.

In the diametrically opposite case, where the theta angles θ₁ and θ₂ are orthogonal to each other, as in $\vartheta_1 = 0°$ and $\vartheta_2 = 90°$, the following results are obtained:



Overlap index = 0.193

Figure 3.4.2: Analysis of the misalignment between primary and secondary structure (case 2)

In this case, the figure shows that the primary lamellae are oriented parallel to the x-axis, while those of the secondary structure are orthogonal and aligned along the y-axis. As a result, the PDF and CDF curves exhibit different trends, with peaks that do not coincide at the same angle, and therefore the overlap index assumes a much lower value compared to the previous case, approximately 20%.

3.4.2 Dataset characterization

All the data collected for each structure during the dataset construction phase (Section 2.2.4.9 – Dataset definition for neural network development) can be analyzed to gain deeper insight into the intrinsic characteristics of the dataset itself.

3.4.2.1 Distribution of Effective Stiffness Values

For instance, it is possible to examine the distribution of normalized effective stiffness values for all the structures included in the dataset:

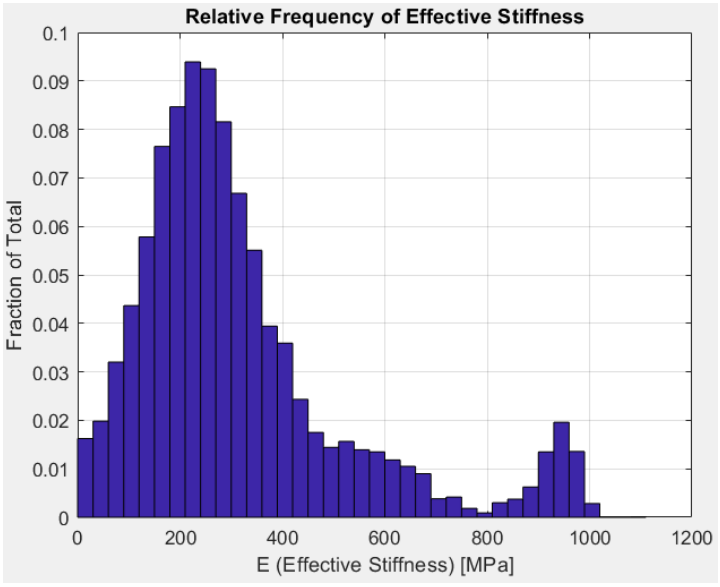
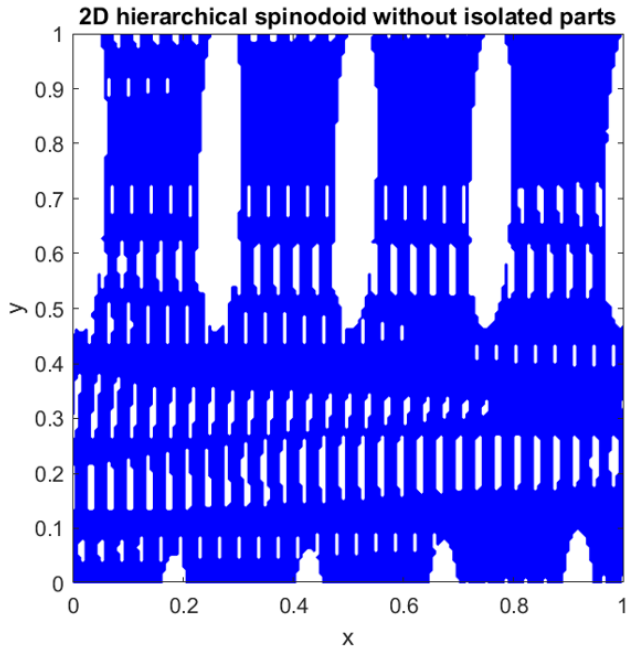


Figura 3.4.1: Distribution of effective stiffness values in the dataset

In general, the spinodoid structures included in the dataset exhibit effective stiffness values ranging from 0 to slightly above 1000 MPa. As illustrated in the graph, the distribution is not uniform but instead shows varying frequencies of E_{eff} values within the dataset.
As previously discussed in Section 2.2.4.3 – Code overview, the choice of lamellae orientations limited to only four discrete values (for both the primary and secondary structures), resulting in a total of 16 possible combinations, leads to a non-uniform distribution of the obtained stiffness values.
Since all structure images were saved and can be sorted according to decreasing stiffness, it is possible to establish a visual association between different stiffness ranges and the structural configurations that tend to exhibit such characteristic values.
The distribution appears bimodal, with a main peak around 250 MPa and a secondary peak near 950 MPa, while stiffness values between 700 and 850 MPa are comparatively less frequent.
The stiffness values corresponding to the higher E_{eff} peak are typically found in structures characterized by $\theta_1 = \theta_2 = 90°$, that is, in spinodoids whose primary and hierarchical lamellae are both aligned with the loading direction, as illustrated in the following figure:



Position **5/12000** in descending order of stiffness

Normalized E_eff = **1003.48 MPa**

Figure 3.4.3: Strucures with $\boldsymbol{\vartheta_1 = \vartheta_2 = 90°}$ tipically exhibit higher stiffness

The discrete number of lamellae orientation combinations resulted in only a few of the generated structures exhibiting stiffness values around 800 MPa, which correspondingly appear with low frequencies in the histogram.
Conversely, the bell-shaped distribution at lower stiffness values is characterized by structures corresponding to the remaining angular combinations, which are distributed relatively uniformly and with greater diversity. In these configurations, the random component introduced by the GRF, and thus by the resulting morphology, significantly influences the mechanical properties; however, from a visual standpoint, no systematic correlation emerges between these properties and the angles θ used in generating the structures.

3.4.2.2 Distribution of node degree values

In addition to the effective stiffness, it is useful to examine certain topological features of the structures in the dataset, particularly the distribution of node degrees within the graphs representing them.
This analysis provides information on local connectivity within the structures and allows for the assessment of the topological diversity of the dataset. Furthermore, understanding the degree distribution is also relevant from the perspective of neural network training: in Message Passing Neural Networks, the degree of each node influences the amount of information exchanged between adjacent nodes during message propagation, thereby affecting the dynamics of the network's training.

As a first step, the global distribution of node degrees across the entire dataset can be analyzed:

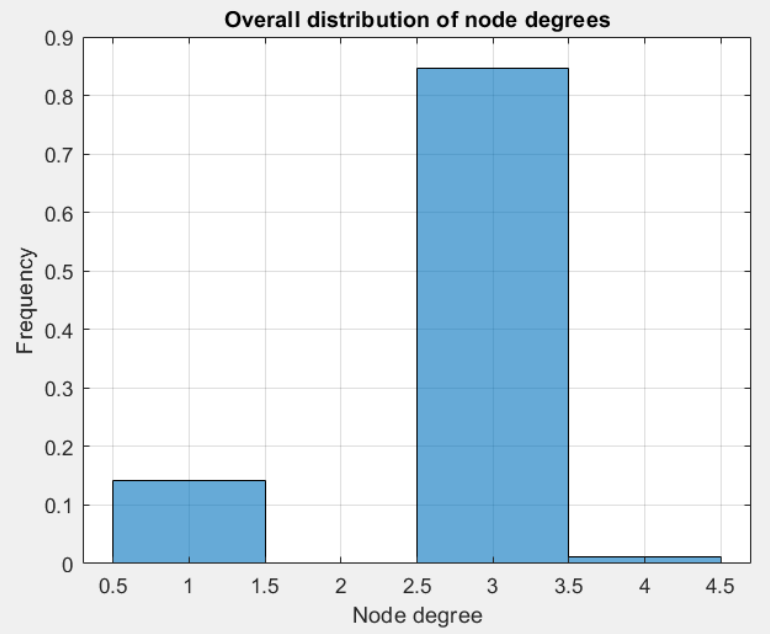


Figure 3.4.4: Global distribution of node degrees across the entire dataset

The graphs representing the structures do not contain any nodes of degree 2; as previously explained in Section 2.2.4.9 – Dataset definition for neural network development, the graphs were generated based on the skeleton of each spinodoid. Therefore, there is a direct analogy between the input graph of the neural network and the structure's skeleton. In this configuration, the graph nodes represent only the terminal nodes of branch chains, that is, nodes of degree 1 (terminal nodes) or nodes with degree greater than or equal to 3 (branching points), as illustrated in Figure 2.2.23: Example of a branch chain in which one end is a terminal node and the other end is a branching node.
In general, it can be observed that all nodes across all structures in the dataset have degrees ranging from 1 to a maximum of 4, with the majority (approximately 85%) having a degree of 3.

It is then possible to compute the average node degree for each structure and plot the results for the entire dataset:

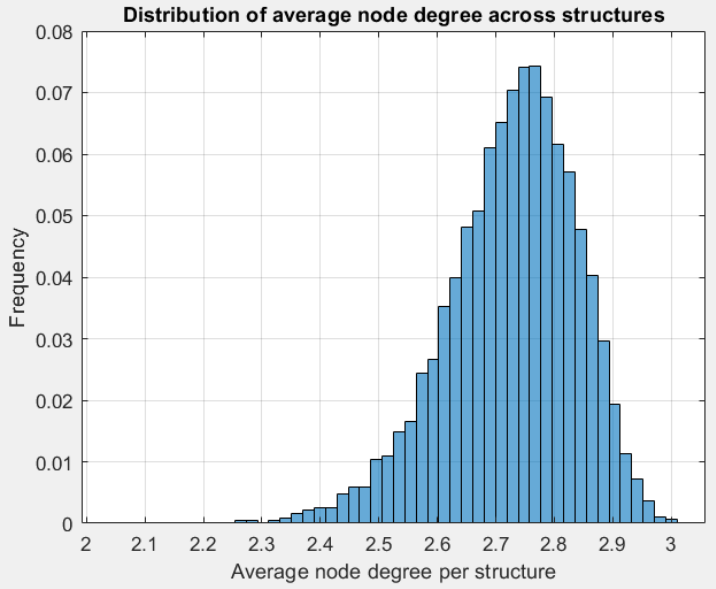


Figure 3.4.5: Distribution of average node degree across structures

The histogram reflects the previously described node distribution, with a peak at values slightly below 2.8.
To analyze the internal variability of node degrees within the structures, the standard deviation of node degrees is computed for each structure, and the results are then plotted for the entire dataset:

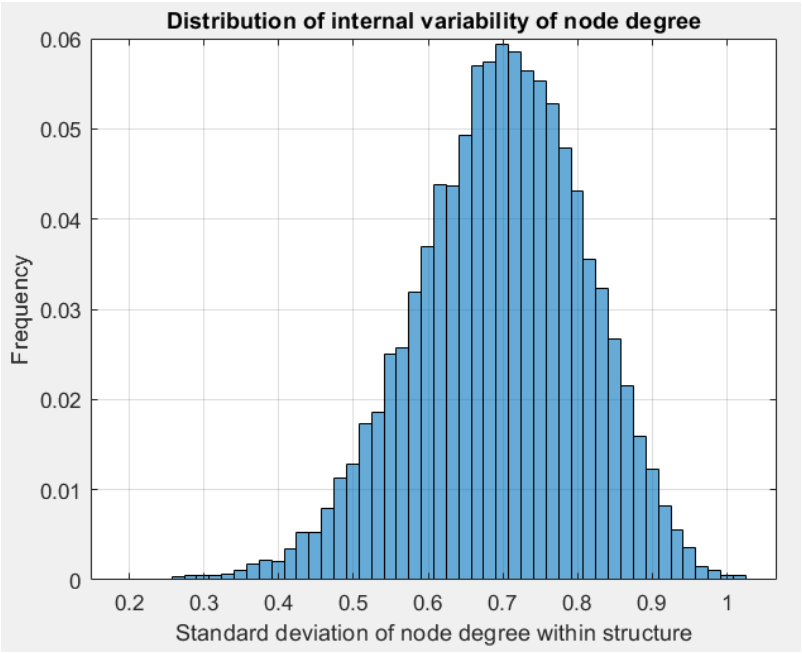


Figure 3.4.6: distribution of internal variability of node degrees within the structures

Moreover, it is possible to analyze the relationship between the average node degree and the standard deviation of the node degree for each structure.

In the figure below each point represents a structure, characterized by its own average degree value and by the dispersion of node degrees within the structure:

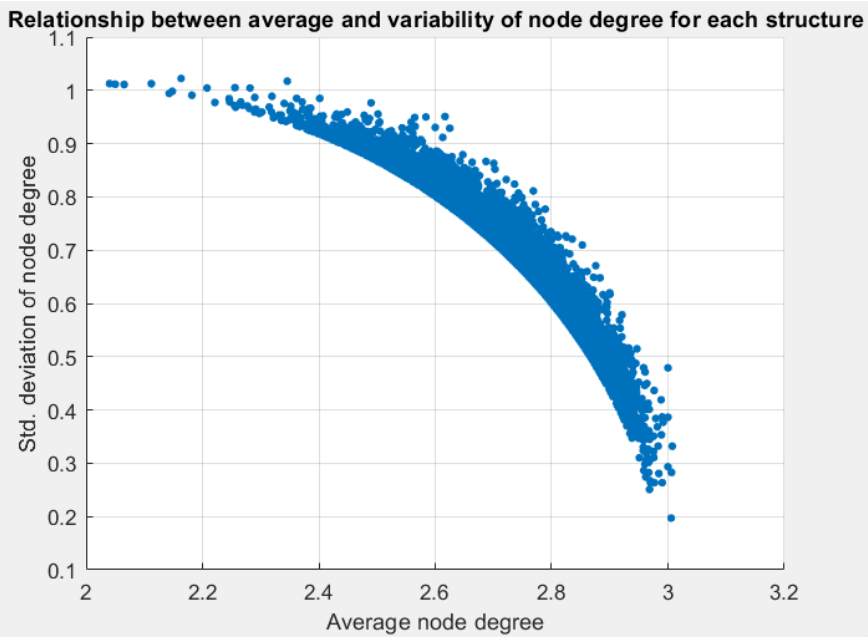


Figure 3.4.7: Relationship between the average node degree and its standard deviation for each structure

This pattern mainly reflects the fact that degree 3 is the most frequent value in the dataset. Structures with an average degree close to 3 tend to exhibit low variability because most nodes share the same degree, whereas structures with lower average degrees include a wider range of node degrees (e.g., 1–3), resulting in higher standard deviation values. Therefore, the observed trend is primarily a consequence of the discrete nature and frequency distribution of node degrees, rather than an intrinsic structural property.

3.5 MPNN results

3.5.1 MPNN with 2 layers

Once the training of the MPNN is completed, as exposed in Section 2.2.5.3 – Training and parameters Update, it is possible to plot a comparison between the actual effective stiffness values and those predicted by the neural network:

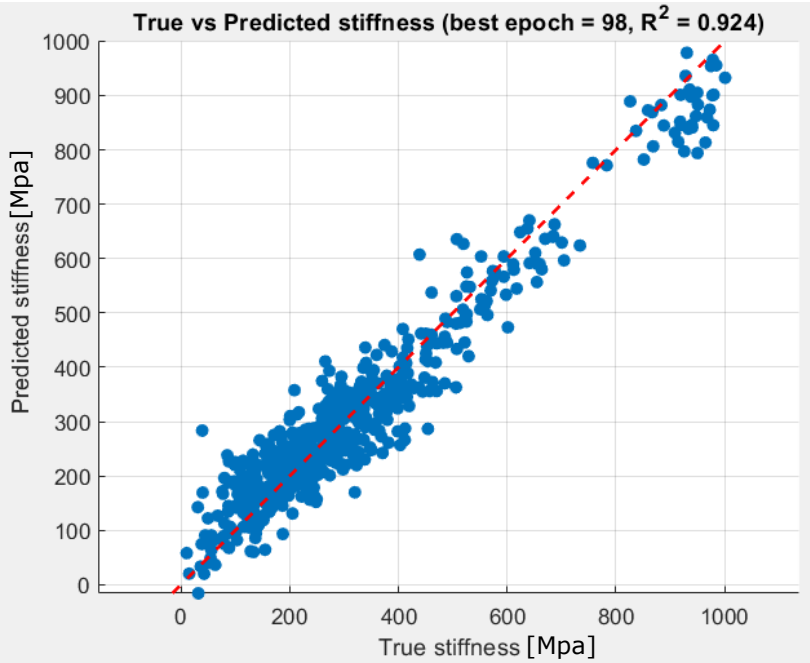


Figure 3.5.1: FEM-computed stiffness versus MPNN-predicted stiffness (MPNN with 2 layers)

The best performance is achieved within 100 epoches, with a maximum accuracy of 92,4% in terms of the coefficient of determination (R^2).

During the training, the values of MSE and R^2 were recorded for each epoch, and their trends can be plotted to illustrate the convergence behavior and speed:

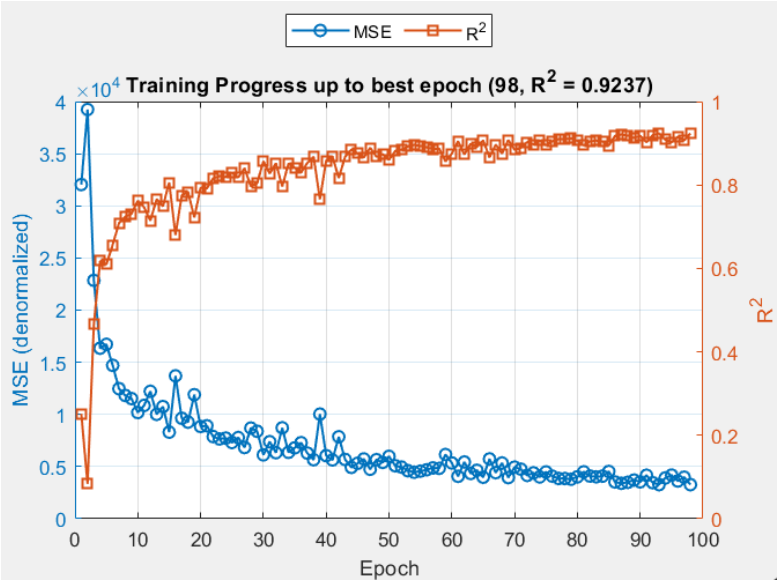


Figure 3.5.2: MPNN training curves showing MSE and R² over successive epochs (MPNN with 2 layers)

The most significant improvements occur during the first few tens of epochs. Once values around 0.85 for R^2 and 6000 for MSE are surpassed, the metrics exhibit an asymptotic behavior, with a reduced growth rate until stabilizing at the optimal maximum value.

3.5.2 MPNN with 3 layers

The same results are shown for the case of an MPNN with three layers:

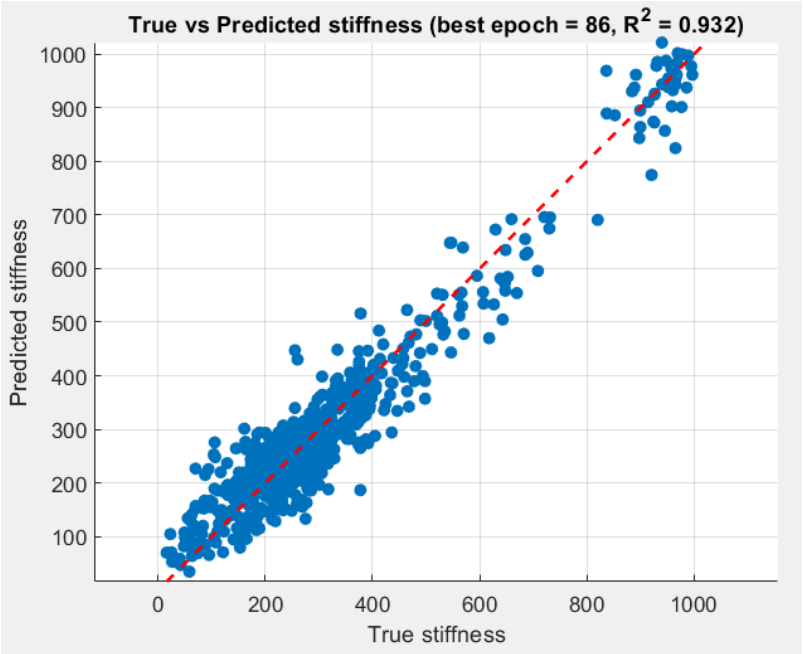


Figure 3.5.3: FEM-computed stiffness versus MPNN-predicted stiffness (MPNN with 3 layers)

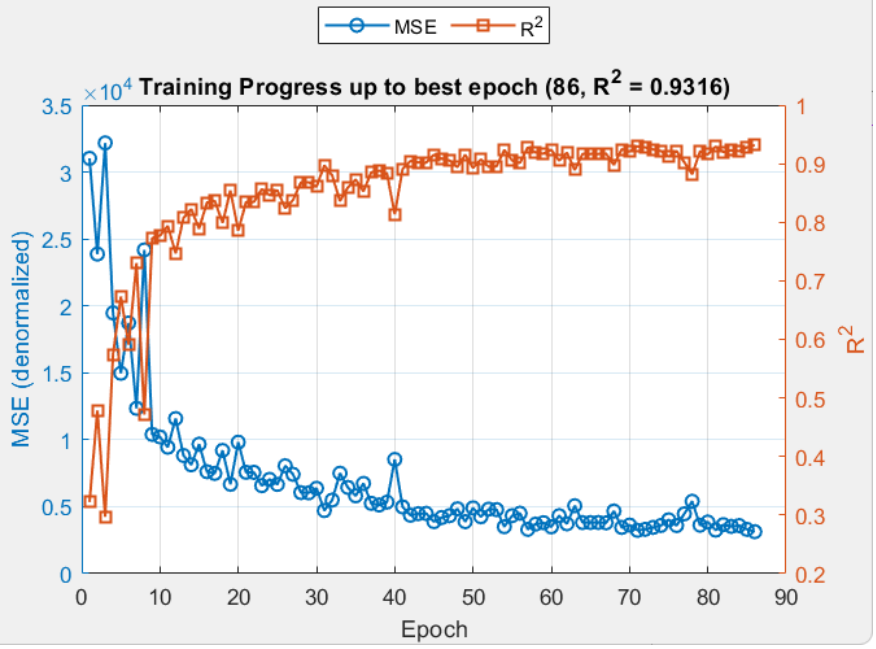


Figure 3.5.4: MPNN training curves showing MSE and R^2 over successive epochs (MPNN with 3 layers)

The neural network with the addition of one message-passing layer exhibits performance comparable to that of the two-layer model, with a slight increase in the coefficient of determination (R^2). Convergence also appears to be slightly faster but a little bit less stable in the first ten epoches.

4. DISCUSSION

4.1 Generation and characterization of 3D structures

4.1.1 Geometric and morphological characterization

4.1.1.1 Relationship Between Structure Type and Distribution of Preferred Branch Orientations

The characterization of the orientation of branches along the skeleton yielded the results presented in Section 3.1.6.3 – Probability Density Distribution of Skeleton Branch Orientation. In the case of a cubic spinodoid structure, the skeleton branches follow a cubic grid pattern and exhibit preferred orientations along the coordinate axes x, y, z as shown in the figure below:

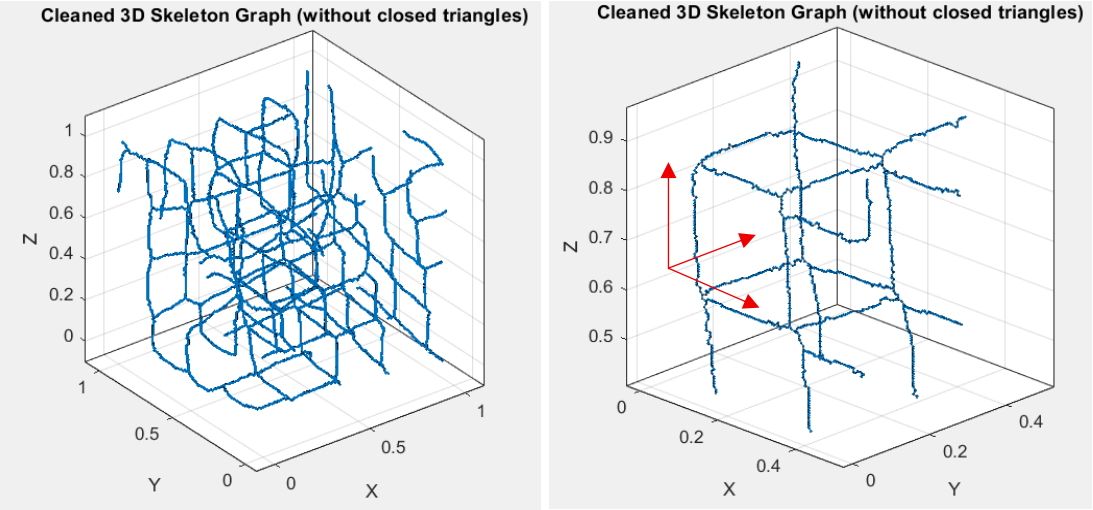


Figure 4.1.1: Cubic Spinodoid Skeleton Pattern

Consistently with the skeleton shown, for the azimuth, the highest probability density values occur at angles around 0 degrees (and thus also 180 degrees) and 90 degrees, corresponding to the X and Y axes. Regarding the elevation, the highest probability density values are observed around 0 and 90 degrees, corresponding to segments parallel or perpendicular to the Z-axis.
In the subsequently shown case of a columnar structure, the highest probability density values for elevation occur around 90 degrees, reflecting the fact that the columnar structure develops predominantly along the vertical Z direction.

4.1.2 Constraints of Voxelization via Python File

Although the code generation occurs very quickly, reading the file in Abaqus is excessively slow and therefore incompatible with the objectives of this work, which involve creating a dataset composed of thousands of structures. Even for a very low-resolution spinodoid,

such as the one shown in Figure 3.1.20 (30 voxels per axis), Abaqus requires over 30 minutes to complete the import. Note that this time does not include the FEM analysis itself, which would represent the most computationally demanding phase.
To overcome this limitation, instead of a Python file, a *.inp* file is written (still in MATLAB). Unlike the previous case, the structure is no longer imported as a part but as an orphan mesh of C3D8 cubic elements. Initially, to become familiar with Abaqus, subsequent steps such as material definition, section assignment, constraints, loads, analysis step, etc., were performed manually via the software's graphical interface. These steps were later integrated into a complete .inp file, containing all the information necessary for the analysis and automatic generation of the desired outputs, thus eliminating the time associated with using the GUI.

4.2 Hierarchical 2D lamellar spinodoids

4.2.1 Disconnected parts issue and influence of generation parameters

The presence of disconnected parts and interruptions in the primary structure caused by the voids of the secondary structure (as shown in Section 2.2.4.2 – Detached parts issue in hierarchical lamellar spinodoids) depends on various parameters.
The most significant among them is the relative orientation between the primary and secondary lamellae: when they are orthogonal to each other, the intersection effect of the secondary structure's voids with the primary lamellae is maximized, and consequently, the generation of cuts and detached regions becomes more pronounced.
The figure below shows an example in which, compared to Figure 2.2.17*: Secondary lamellae intersect primary lamellae, creating disconnected parts*, only the relative orientation of the primary and secondary lamellae has been modified, set to be orthogonal to illustrate the worst-case scenario, while the parameters of relative density and wavelength have been left unchanged:

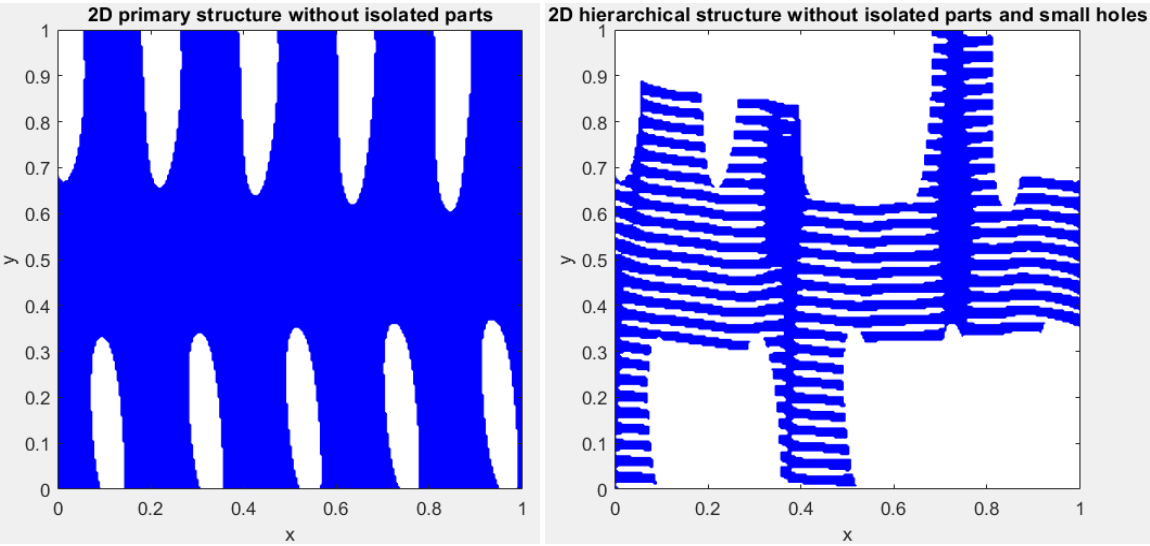


Figure 4.2.1: Detached parts issue when primary and secondary lamellae are orthogonal to each other

This issue is also influenced by the values of relative density and wavelength used in the generation of the primary and secondary GRFs. In particular, a low relative density of the

primary structure (thin primary lamellae) or a high wavelength of the secondary structure (very long secondary lamellae and voids) amplifies the effect.
Remaining in the worst-case scenario of primary and secondary lamellae being orthogonal to each other, the effects of the two aforementioned parameters are shown:

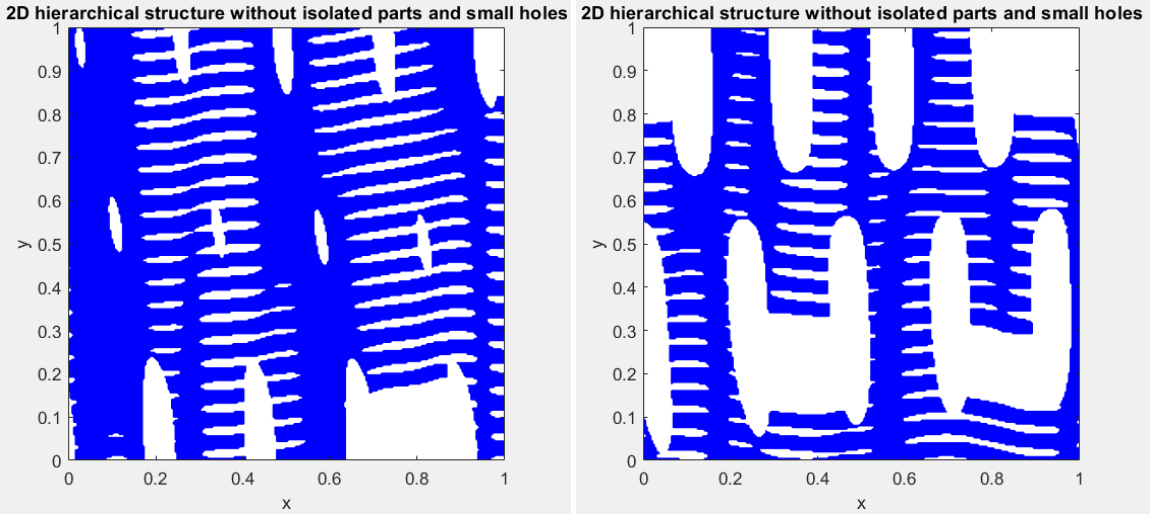


Figure 4.2.2: Structures with a relative density of the primary structure of 0.9 (left) and 0.7 (right), with all other generation parameters being the same

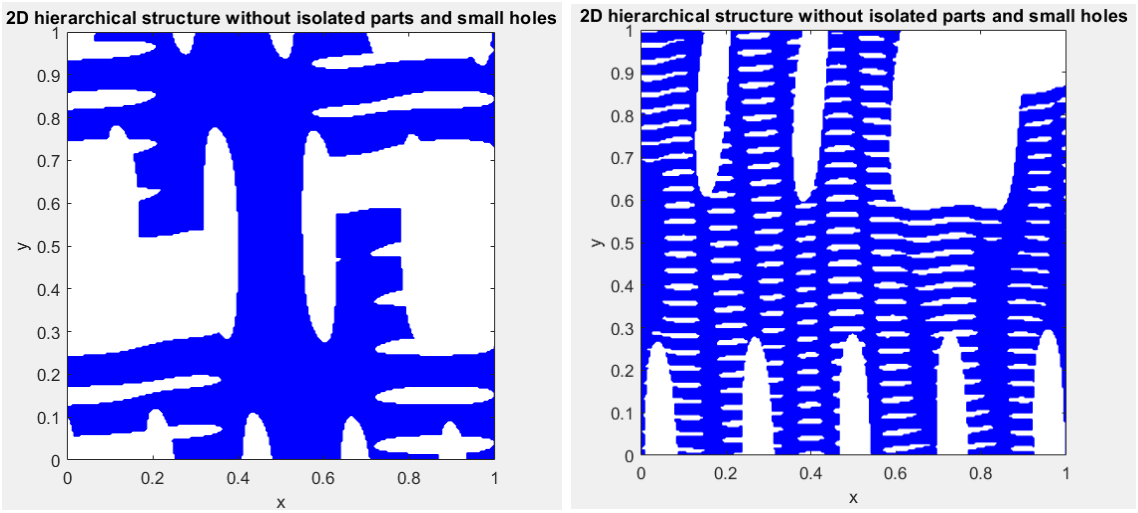


Figure 4.2.3: Structures with secondary structure wavelength equal to $^{1}/_{2}$ (left) and $^{1}/_{6}$ (right) of the primary structure wavelength, with all other generation parameters being the same

Given the objective of generating and characterizing the structures both morphologically and in terms of mechanical properties, this phenomenon can introduce noise and outliers in the creation of the dataset, producing excessively "empty" structures, very low or null compression stiffness values, and discontinuous structural skeletons. Therefore, an empirical tuning of these variables is necessary to minimize the occurrence of the problem. The relative density and wavelength values that will be used in the following are the result of these empirical trials.

4.2.2 Correlation between static wave propagation angle ϑ and lamellae orientation

When an angle ϑ is set for the direction of the unit vectors of the static waves, this only defines the propagation direction of each individual wave. However, the lamellae and voids in the final microstructure do not directly follow these directions. This is because the structure forms where the resulting random field (GRF) is below a certain threshold φ_0, as explained in Section 2.1.1 – Theoretical Background. The value of the GRF at each point is the sum of the contributions of all the superimposed waves, each with its own phase and direction.

Consequently, the orientation of the lamellae depends on the overall interference of the waves, and not on their propagation direction. Even if all the waves have the same direction ϑ ($\pm$ a range of variation), the sum of their values can create lamellae oriented in different directions, because the final pattern is determined by the spatial distribution of the peaks and troughs of the GRF.

It should be noted that both the angle ϑ and the lamellae orientation are expressed as angles with respect to the positive x-axis direction.

To better understand, at a practical level, how ϑ relates to the lamellae orientation, an example concerning only a primary spinodoid structure is presented.

For $\vartheta_1 = 30°$, the result is:

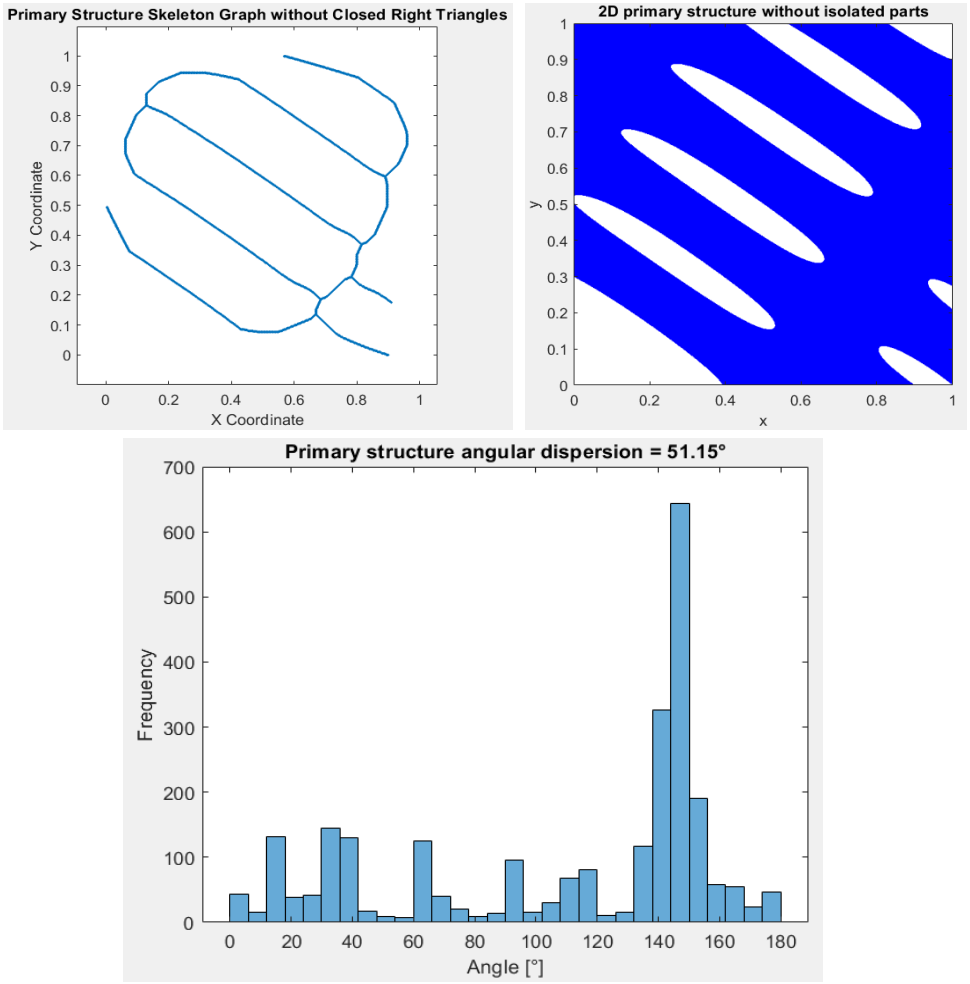


Figure 4.2.4: Correlation between static wave propagation angle and lamellae orientation $(\boldsymbol{\vartheta_1 = 30°})$

Therefore, with a propagation angle of the static waves equal to 30°, the lamellae are mostly oriented at an angle around 150°.
The case of $\vartheta_1 = 60°$ is also reported:

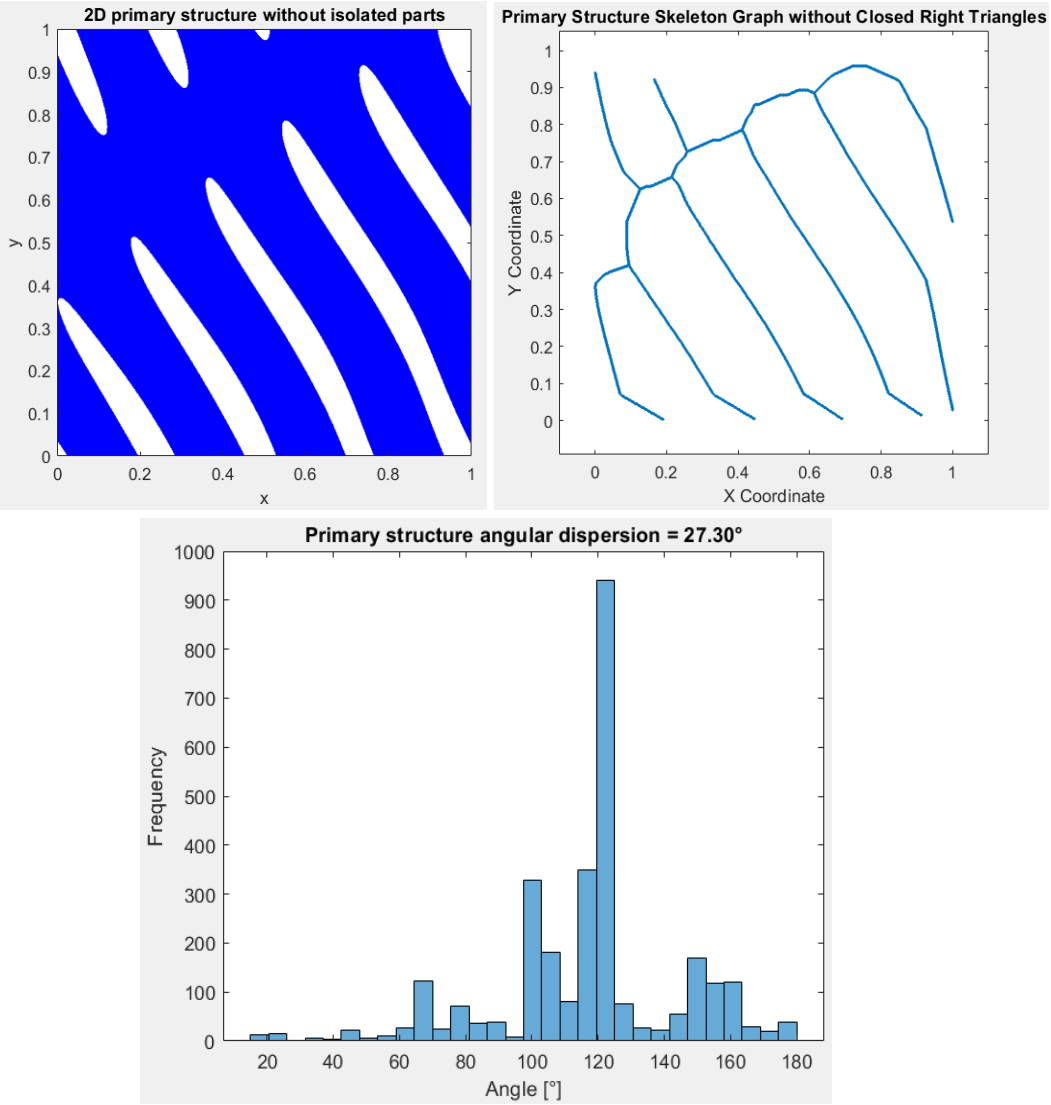


Figure 4.2.5: Correlation between static wave propagation angle and lamellae orientation $(\boldsymbol{\vartheta_1 = 60°})$

In this case, the lamellae are mostly oriented around 120°.
Various tests with different theta angles suggest that the indicative orientation of the lamellae, following the convention used in this thesis, is approximately $180 - \vartheta_1$, that is, the supplementary angle of ϑ_1.

4.2.3 Influence of the Size of Removed Voids on Effective Stiffness

To study the influence of the size of removed voids (introduced in Section 2.2.4.5.1 – Computational Load Reduction by Removing Small Cavities from the Hierarchical Structure), four different structures will be analyzed in the following. They all originate from the same original structure (Case 1), but with progressively increasing maximum removable cavity sizes. For each of them, the resulting structure will be shown along with

the corresponding value of normalized effective stiffness (i.e., divided by the relative density obtained after filling certain cavities), calculated as described in Section 2.2.4.8 – Computation of effective stiffness.
The maximum threshold size of the fillable cavities (d) will be expressed as a number of points (in percentage of the grid's linear resolution), so that the analysis can be as general as possible. For example, if the grid has a linear resolution of 100 (that is, a total of 10,000 grid points), a value of *d* equal to 5 means that all cavities composed of up to 5 *false*-valued points are filled.
The images of the structures and the results obtained for the four different cases are reported below:

	Case 1 (original)	Case 2	Case 3	Case 4
d, voids size (%grid_res)	0	2.5%	10%	30%
Relative density (after removing voids)	0.725	0.726	0.734	0.779
Normalized E_{eff} [Mpa]	210.56	212.23	222.64	359.83

Figure 4.2.6: Correlation between Removed Cavity Size, Relative Density, and Normalized Effective Stiffness

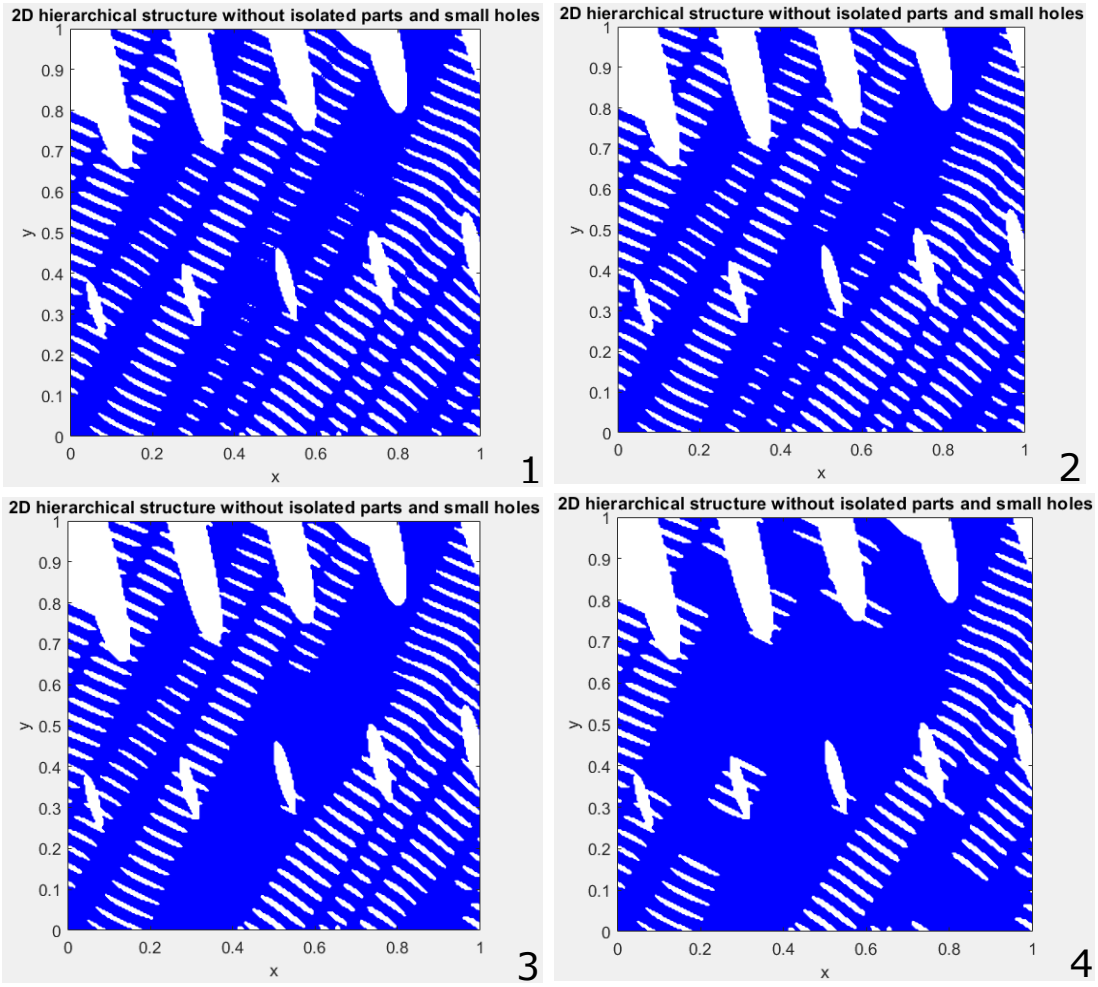


Figure 4.2.7: Final Structures after Filling Voids of Different Sizes

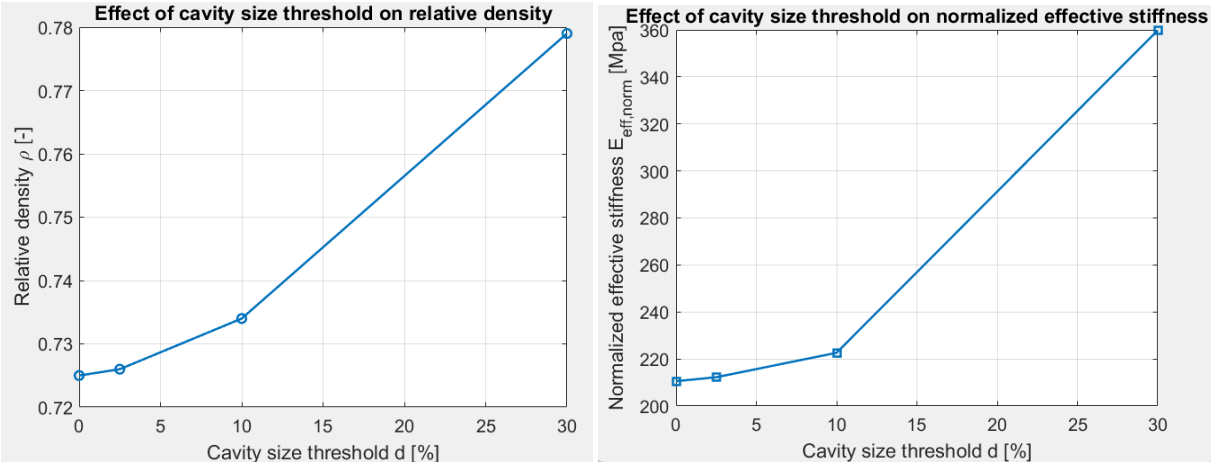


Figure 4.2.8: Plot of Relative Density (left) and Normalized Effective Stiffness (right) versus threshold Size of Filled Voids

First of all, the filling of certain voids obviously causes an increase in the relative density of the resulting structure, with the percentage increase relative to the original structure reported in the table below:

	Case 1 (original)	Case 2	Case 3	Case 4
d, voids size (%grid_res)	0	2.5%	10%	30%
Percentage increase in relative density	0	0.14%	1.24%	7.45%

Figure 4.2.9: Percentage Increase in relative density

Regarding the effective stiffness, although it is already normalized and thus accounts for the increase in relative density, it shows a percentage increase relative to the original value, reported below:

	Case 1 (original)	Case 2	Case 3	Case 4
d, voids size (%grid_res)	0	2.5%	10%	30%
Percentage increase in normalized effective stiffness	0	0.79%	5.7%	70%

Figure 4.2.10: Percentage increase in normalized effective stiffness

Based on these results, a value of *d* equal to 2.5 was chosen, as it allows maintaining a limited variation in the effective stiffness (around 1%) while, at the same time, a visual inspection of the structures (Figure 4.2.7: Final Structures after Filling Voids of Different Sizes) shows that this size is appropriate to fill only the small voids of interest without affecting those naturally resulting from the alternation of the lamellae.

4.3 Model Accuracy and Future Enhancements

It is important to note that the resolution of the binary domain, in both two-dimensional (2D) and three-dimensional (3D) cases, represents the fundamental parameter underlying the entire process. A significant increase in this resolution would improve every stage of the workflow: from the overall definition of the structure, which would exhibit a more detailed and physically consistent morphology, to the morphological characterization, benefitting from smoother and more accurate skeletons; from the voxelization process, which would more faithfully reproduce the geometry, to the finite element method (FEM) analysis, which would profit from a finer discretization and yield more precise and reliable results. These improvements, however, would come at the cost of a considerable increase in computational demand, particularly in the three-dimensional case, where complexity grows exponentially with spatial resolution.

Initially, as previously discussed, the work focused on the more general case of three-dimensional structures, completing the stages of structure generation and morphological characterization. However, due to limited computational resources (a laptop equipped with an Intel® Core™ i7-9750H CPU, 2.6 GHz, 6 cores, 16 GB RAM, and Windows 11, 64-bit), the analysis was subsequently restricted to the two-dimensional case. This decision was motivated by the need to complete the entire workflow, from structure generation and morphological characterization to finite element (FEM) analysis and neural network development, within the time constraints of the thesis, while maintaining an acceptable level of accuracy. Although this simplification reduced the geometrical complexity compared to the three-dimensional case, it still enabled an effective validation of the proposed approach, which can be readily extended to 3D structures when greater computational resources become available.

With regard to the MPNN model, its accuracy could be further improved by expanding and diversifying the training dataset. Specifically, as shown in Section 2.2.4.3 – Code overview, the variability of the structures obtained in the current dataset is limited to only four possible orientations of the primary and hierarchical lamellae. Introducing a larger number of samples exhibiting greater morphological diversity, such as different lamellar orientations, input relative densities, or degrees of structural hierarchy, would allow the network to learn more effectively the complex nonlinear relationships between geometry and mechanical properties, thereby enhancing its generalization capability.

In the literature, several studies ([3], [4], [5]) have demonstrated that training neural networks for predicting the mechanical properties of primary spinodoid structures typically relies on datasets significantly larger than the one employed in the present work. When considering that such primary spinodoid structures are described in those datasets by only four parameters (the relative density and the three limiting angles ϑ), paired with the corresponding effective stiffness, the necessity for a larger dataset becomes even more evident in the case of more complex structures, such as those analyzed in this work. In the present framework, each sample is characterized by multiple vectors and matrices that describe its geometry and connectivity, which substantially increases the dimensionality and variability of the input space.

This comparison highlights how the extension of the current dataset, both in terms of size and morphological diversity, would likely yield substantial improvements in predictive accuracy, model robustness, and overall generalization capability, thereby further validating the potential of the proposed MPNN-based framework for complex structural systems.

5. CONCLUSIONS

This thesis has systematically addressed the study, generation, and mechanical characterization of hierarchical spinodoid metamaterials, with the overarching goal of exploring their potential in the design of advanced mechanical materials with tunable and adaptive properties. The research integrated concepts from stochastic geometry, numerical modeling, and machine learning into a unified framework capable of describing, simulating, and predicting the effective mechanical behavior of complex hierarchical structures.

In the first stage of the work, a numerical generation method for three-dimensional spinodoid structures was developed, based on the combination of static wave superposition and subsequent morphological processing of a binary domain. This method enabled the creation of diverse morphologies with controllable anisotropy and density distributions. The generated structures were then subjected to geometric and topological characterization to quantitatively describe their morphology and to extract key features representative of the underlying microstructure.

To perform this characterization, a skeletonization process was implemented, allowing for a detailed analysis of several structural descriptors, including connectivity, branch thickness, and branch length distributions. This stage provided a clear understanding of how the microstructural topology influences macroscopic mechanical behavior. The main challenges and critical issues encountered during the generation and characterization processes, such as disconnected regions, resolution-dependent artifacts, or irregular skeleton geometries, were identified and systematically addressed through tailored computational strategies and parameter adjustments.

The concept of structural hierarchy was then introduced, first applied to 3D structures and later extended to two-dimensional (2D) domains for computational efficiency, as justified in Section 4.3 – Model Accuracy and Future Enhancements. The introduction of secondary lamellae within primary spinodoid structures enabled the exploration of hierarchical effects on overall stiffness and load transfer. After a thorough morphological characterization of the hierarchical structures, their mechanical properties were evaluated through finite element analyses (FEM) conducted in Abaqus. These simulations were performed using fully automated procedures capable of extracting the effective stiffness directly from output files, without any manual intervention through the graphical interface.

The results of the FEM simulations, combined with morphological data, were used to construct a comprehensive reference database for the subsequent training of a Message Passing Neural Network (MPNN) developed entirely in MATLAB. The network successfully learned the relationship between local graph morphology and the effective stiffness of the material, achieving a coefficient of determination of $R^2 > 0.92$, thereby demonstrating the reliability and predictive capability of the proposed model. The close agreement between FEM-computed and MPNN-predicted stiffness values confirmed that graph-based representations can effectively encode complex topological and geometrical information, allowing the neural network to infer global mechanical behavior from local structural descriptors.

The integrated FEM–MPNN framework developed in this thesis constitutes a fully automated and generalizable workflow for the analysis and prediction of mechanical properties in spinodoid metamaterials. Beyond the specific application presented here, the

workflow has the potential to serve as a foundational tool for inverse design, enabling data-driven optimization of complex metamaterials without repeated costly FEM simulations.
To further generalize the automated process, future work could include the implementation of a universal conversion module capable of transforming input data from various formats (such as images, meshes, or other geometric representations) into a standardized binary format. Such a module would provide a unified entry point for all subsequent stages of the workflow, from morphological characterization and voxelization to FEM simulations and neural network training, thereby enhancing the flexibility and applicability of the framework across a wide range of material systems and structural configurations.
The results obtained in this work provide a solid foundation for the development of hierarchical design strategies at the three-dimensional scale and for the extension of data-driven models to predict additional mechanical properties, such as anisotropy, energy absorption, or fracture behavior. Furthermore, the demonstrated integration of physics-based modeling and artificial intelligence opens the door to real-time predictive tools for metamaterial optimization, where desired macroscopic properties can directly guide the generation of optimal microstructures.
The original contributions of this thesis can be summarized as follows:

- Development of a generation algorithm for hierarchical spinodoid structures based on controlled static-wave superposition, enabling continuous tuning of morphological parameters and anisotropy;

- Integration of stochastic generation, morphological characterization, FEM analysis, and deep learning into a single, automated MATLAB pipeline, utilizing Abaqus for simulation with complete automation and no graphical interface interaction;

- Implementation of a Message Passing Neural Network (MPNN) in MATLAB, including manual backpropagation and explicit gradient computation, specifically designed for predicting effective stiffness from graph-encoded structural data;

- Introduction of a graph-based representation of hierarchical spinodoid microstructures, capable of simultaneously capturing geometric and topological information to enable efficient learning of structure–property relationships;

- Demonstration of consistent predictive accuracy ($R^2 > 0.92$) in estimating effective stiffness directly from topological data, confirming the feasibility of machine-learning-driven mechanical prediction for complex hierarchical materials.

In conclusion, the work presented in this thesis represents a significant advancement in the computational study of hierarchical metamaterials. By uniting stochastic geometry, finite element analysis, and graph-based neural networks into a coherent and automated workflow, this research contributes to the emerging field of materials-by-design, offering a practical and scalable foundation for inverse design, multiscale modeling, and data-driven material optimization.

6. References

[1] C. Liu, Z. Gao, J. Chang, J. Zhao, S. Qiu, P. Yu and X. Zhang, "A lattice-mechanical metamaterial with tunable two-step deformation, tunable stiffness, tunable energy absorption and programmable properties," *Materials Research Express,* vol. 11, p. 125801, 2024.

[2] Y. Zheng, W. Qiu, X. Liu, Z. Huang and L. Xia, "Damage-tolerant mechanical metamaterials designed by fail-safe topology optimization," *Materials & Design,* vol. 246, no. 113546, 2025.

[3] S. Kumar, S. Tan and L. Zheng, "Inverse-designed spinodoid metamaterials," *npj Comput Mater,* vol. 6, no. 73, 2020.

[4] H. Wang, Y. Lyu, J. Jiang and H. Zhu, "Data-driven inverse design of novel spinodoid bone scaffolds with highly matched mechanical properties in three orthogonal directions," *Materials & Design,* vol. 251, no. 113697, 2025.

[5] Y. Liu, H. Wang, L. Yan, J. Huang and Y. Liang, "Mechanical properties of homogeneous and functionally graded spinodal structures," *International Journal of Mechanical Sciences,* vol. 269, no. 109043, 2024.

[6] M. Vafaeefar, K. M. Moerman and T. J. Vaughan, "Experimental and computational analysis of energy absorption characteristics of three biomimetic lattice structures under compression," *Journal of the Mechanical Behavior of Biomedical Materials,* vol. 151, no. 106328, 2024.

[7] L. Zheng, S. Kumar and D. M. Kochmann, "Data-driven topology optimization of spinodoid metamaterials with seamlessly tunable anisotropy," *Computer Methods in Applied Mechanics and Engineering,* vol. 383, no. 113894, 2021.

[8] S. Liu and P. Acar, "Generative Adversarial Networks for Inverse Design of Two-Dimensional Spinodoid Metamaterials," *AIAA Journal,* vol. 62, 2024.

[9] W. Deng, S. Kumar, A. Vallone, D. M. Kochmann and J. R. Greer, "AI-Enabled Materials Design of Non-Periodic 3D Architectures With Predictable Direction-Dependent Elastic Properties," *Adv. Mater.,* vol. 36, no. 2308149, 2024.

[10] G. X. Gu, M. Takaffoli and M. J. Buehler, "Hierarchically Enhanced Impact Resistance of Bioinspired Composites," *Adv. Mater.,* vol. 29, no. 1700060, 2017.

[11] Y. Zhu, Z. Chen, Y. Zhu, M. Gresil and Y. Tang, "Bio-inspired perturbated hierarchical mechanical metamaterial for energy absorption," *International Journal of Mechanical Sciences,* vol. 307, no. 110847, 2025.

[12] P. P. Meyer, C. Bonatti, T. Tancogne-Dejean and D. Mohr, "Graph-based metamaterials: Deep learning of structure-property relations," *Materials & Design,* vol. 223, no. 111175, 2022.

[13] M. Maurizi, C. Gao and F. Berto, "Predicting stress, strain and deformation fields in materials and structures with graph neural networks," *Scientific Reports,* vol. 12, no. 21834, 2024.

[14] M. Maurizi, C. Gao and F. Berto, "Inverse design of truss lattice materials with superior buckling resistance," *npj Computational Materials,* vol. 8, no. 247, 2022.

[15] X. Zheng, X. Zhang, T.-T. Chen and I. Watanabe, "Deep Learning in Mechanical Metamaterials: From Prediction and Generation to Inverse Design," *Adv. Mater.,* vol. 35, no. 2302530, 2023.