University of New Mexico
UNM Digital Repository

Mechanical Engineering ETDs | Engineering ETDs

Fall 11-14-2023

Effect of Fiber Proximity on the Pullout Response: A Finite Element Study

Tyler Mitchell
University of New Mexico - Main Campus

Follow this and additional works at: https://digitalrepository.unm.edu/me_etds

Part of the Mechanical Engineering Commons

Recommended Citation

Mitchell, Tyler. "Effect of Fiber Proximity on the Pullout Response: A Finite Element Study." (2023). https://digitalrepository.unm.edu/me_etds/242

This Thesis is brought to you for free and open access by the Engineering ETDs at UNM Digital Repository. It has been accepted for inclusion in Mechanical Engineering ETDs by an authorized administrator of UNM Digital Repository. For more information, please contact disc@unm.edu.

Tyler Grant Mitchell

Candidate

Mechanical Engineering

Department

This thesis is approved, and it is acceptable in quality and form for publication:

Approved by the Thesis Committee:

Dr. Yu-Lin Shen , Chairperson

Dr. Heng Zuo

Mr. Niccoli Scalice

Effect of Fiber Proximity on the Pullout Response: A Finite Element Study

by

Tyler Grant Mitchell

B.S., University of New Mexico, 2022

THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science
Mechanical Engineering

The University of New Mexico

Albuquerque, New Mexico

December, 2023

Dedication

To my parents and sister for their unwavering belief in me.

Acknowledgments

I would like to thank my advisor, Dr. Yu-Lin Shen, for his guidance and expertise. I would also like to thank my other committee members Dr. Heng Zuo and Mr. Niccoli Scalice for their time.

Effect of Fiber Proximity on the Pullout Response: A Finite Element Study

by

Tyler Grant Mitchell

B.S., University of New Mexico, 2022

M.S., Mechanical Engineering, University of New Mexico, 2023

Abstract

With the advancement of computational capabilities, there is growing interest in developing intricate models for composite materials. This has been historically challenging due to the multifaceted interactions and failure mechanisms of these materials. Multiscale modeling holds promise for predicting the intricate dynamics of such materials. A foundational understanding of their microscale behaviors is critical, particularly the interfacial interactions between the fiber and matrix. This study delves into the influence of fiber volume fraction (FVF) on fiber pull-out tests, focusing on the impact of neighboring fibers across different geometries and the underlying cohesive properties.

Contents

List of Figures ix

List of Tables xiii

Glossary xiv

1 Introduction 1

2 Review of Related Literature 4

2.1 Observable State Variable Theories 5

2.1.1 Long Fiber Models 5

2.1.2 Short Fiber Models 6

2.1.3 Non-ISV Failure Theories 8

2.2 Internal State Variable Theories 9

3 Methodology 12

3.1 Materials 12

3.1.1 Matrix 13

3.1.2 Fiber 14

3.1.3 Cohesive elements 15

3.1.4 Viscosity 18

3.2 Model Setup 19

3.2.1 Boundary Conditions 19

3.2.2 Mesh Generation 21

3.2.3 Solver Configuration 21

4 Results **23**

4.1 Rule of Mixtures (ROM) 23

4.1.1 Without cohesion 25

4.1.2 With cohesion 26

4.2 Fiber Analysis 26

4.2.1 Single fiber 27

4.2.2 Single vs Double 31

4.2.3 Single vs Double vs Double without Cohesion 35

4.3 Cohesive Properties 35

4.3.1 Maximum Displacement 36

4.3.2 Maximum Traction 37

4.3.3 Thickness 41

4.4 Embedded Fiber 41

5 Conclusions 46

5.1 Limitations of the Study 47

5.2 Implications for Future Research 47

Appendices 48

A S13 Contour Plots 49

B Parametric Python Code 54

B.1 Parametric code for varying FVF 54

B.2 Module Code 67

References 98

List of Figures

2.1 Material directions as resulting from orthotropic assumption[13] . . 5

2.2 Tiled concentric cylinders as seen in the shear lag model 7

2.3 Single fiber pull-out testing methods (a) SFPO Type 1, (b) SFPO Type 2, single fiber fragmentation tests (SFFT)[20] 10

2.4 Typical load-displacement curve for pullout test[4] 11

3.1 Materials arrangement in the model 13

3.2 Matrix part 14

3.3 Fiber part 15

3.4 Cohesive part 16

3.5 Traction separation law graphical representation [14] 17

3.6 Different model instances 19

3.7 Model boundary conditions 20

4.1 Displacement of rule of mixtures models 25

4.2 Setup for single fiber case 28

4.3 Reaction force of single fiber case with different FVF 28
4.4 10% FVF at peak stress 29
4.5 60% FVF at peak stress 29
4.6 Single fiber contour plots for s33 - Peak Stress 29
4.7 10% FVF at failure 30
4.8 60% FVF at failure 30
4.9 Single fiber contour plots for s33 - Failure 30
4.10 Reaction force of double fiber case with different FVF 31
4.11 Double fiber vs single fiber for different FVF 32
4.12 Double fiber contour plots for s33 at peak stress 33
4.13 Double fiber contour plots for s33 at failure 34
4.14 Setup for double fiber with single cohesive zone 35
4.15 Reaction force of single fiber, double fiber, and double fiber without the second cohesive zone case 36
4.16 Change in load response for different FVF and different max displacements. 38
4.17 Change in load response for 10% and 60% FVF for different max traction values. 39
4.18 Change in load response for 10% and 60% FVF. 40
4.19 Reaction force of double fiber with varying cohesive zone thickness . . 41

4.20 Model setup for embedded fiber case. Matrix material is shown in red, cohesive in cream, and fiber in green. 42

4.21 Comparison of single fiber case and embedded fiber case 43

4.22 10% FVF at peak stress 44

4.23 60% FVF at peak stress 44

4.24 Embedded fiber contour plots for s33 at peak stress 44

4.25 10% FVF at failure 45

4.26 60% FVF at failure 45

4.27 Embedded fiber contour plots for s33 at failure 45

A.1 10% FVF at peak stress 50

A.2 60% FVF at peak stress 50

A.3 Single fiber contour plots for s13 at peak stress 50

A.4 10% FVF at failure 51

A.5 60% FVF at failure 51

A.6 Single fiber contour plots for s13 at failure 51

A.7 10% FVF at peak stress 52

A.8 60% FVF at peak stress 52

A.9 Double fiber contour plots for s13 at peak stress 52

A.10 10% FVF at failure 53

A.11 60% FVF at failure 53

A.12 Double fiber contour plots for s13 at failure 53

List of Tables

3.1 Material Color Table 13

3.2 Matrix Material Properties [5] 14

3.3 Carbon Material Properties [5] 14

3.4 Cohesive Material Properties [5] 15

3.5 Instance Table 20

3.6 Boundary Conditions 21

3.7 Step Data 22

Glossary

FVF	Fiber Volume Fraction
ROM	Rule of Mixtures
ISV	Internal State Variable
OSV	Observable State Variable
SFPO	Single Fiber Pull-out
CDM	Continuum Damage Mechanics
RVE	Representative Volume Element

Chapter 1

Introduction

Composites are hybrid materials formed by combining two or more distinct constituents, often with contrasting properties, to produce a material with enhanced characteristics. The classic examples include the combination of steel rebar and concrete or the integration of natural rocky aggregates with a cement matrix, as seen in Roman concrete. However, the focus of this work is on continuous fiber composites where the reinforcing fibers generally span the entire length of the material. These continuous fibers, often made of glass or carbon, are enveloped by a matrix material like a thermoset resin (e.g., epoxy) to create composites with tailored mechanical properties.

In a continuous fiber composite, the matrix serves as a binder that transfers loads between fibers, thereby augmenting the composite's overall strength and stiffness. Unlike metals, whose grains interact directly, fibers in composites don't inherently bond, thus necessitating the role of the matrix. The interaction between fibers and the matrix creates an "interface zone" where mechanical properties can be different from those of the individual components. This interface can be modeled using traction-separation laws that describe how the interfacial layer carries load before

eventually experiencing inelastic deformation and fracture.

Modeling the behavior of composites presents unique challenges due to their heterogeneous nature. Traditional approaches such as the Rule of Mixtures (ROM) model provide simple, albeit often inaccurate, estimations by averaging the properties of the constituents. These models fall short of capturing complex phenomena like interfacial stress or micromechanical fracture. Micromechanical models aim to address these limitations by explicitly considering interfacial properties, fiber fractures, and matrix cracking at a microscopic scale.

However, microscale models are not computationally feasible for analyzing entire laminated composites or composite parts. This dilemma necessitates the translation of microscale properties to mesoscale models that describe laminates (stacked layers of composites with varying fiber orientations). Classical Laminate Theory is frequently employed at the mesoscale, relying on assumptions that may not hold in all applications[13]. The transition from micro to mesoscale is even more critical in high-strain applications, for example, with deployable space structures, where standard assumptions can lead to inaccurate predictions.

Recent advances suggest that multiscale modeling and integrating accurate models at each scale can offer a more robust solution, especially for high-strain applications. This method allows for the detailed study of interface behavior at the microscale, which can then be used to inform mesoscale models that account for delamination and other complex phenomena in laminates.

The present work aims to investigate one critical element of multiscale modeling: the role of how the fiber volume fraction-the percentage of fiber in the total volume of the composite, and thus the fiber proximity-influences the interfacial behavior and, consequently, the composite's mechanical properties. This focus helps to bridge the gap between microscale and mesoscale models, thereby providing a more accurate

representation of composites under various loading conditions.

Chapter 2

Review of Related Literature

Many mechanical and failure models have been proposed in the study of composite materials. Generally, these theories are either Internal state variable (ISV) or observable state variable (OSV) theories (also called non-ISV theories)[9]. Non-ISV theories are interested in describing the general response of the composite under loading. They require a large amount of empirical data to support them and are interested in the macro-scale phenomena of the material without concern for the internal state of the system. Internal state variable theories are interested in modeling the evolution of a material temporally. They do this by tracking the history of the material with differential equations to help predict the behavior of the material in the future[12]. Thus previous loading effects (damage, age, phase transformations, etc.) affect the material's subsequent behavior. Because ISV models are more closely linked to the physics of the system they require less empirical data. However, non-ISV models are generally simpler than ISV models and can be applied to many different situations with a reasonable amount of accuracy.

2.1 Observable State Variable Theories

2.1.1 Long Fiber Models

One of the first theories developed to understand the mechanical characteristics of composites focused on long fiber models. These models operate on the assumption that the fibers are sufficiently long, making the behavior at the fiber ends negligible. Long fiber models describe composites in terms of two primary directions: the matrix direction (perpendicular to the fiber orientation) and the fiber direction (parallel to the fiber orientation). This characterization results in an orthotropic material (Fig. 2.1).

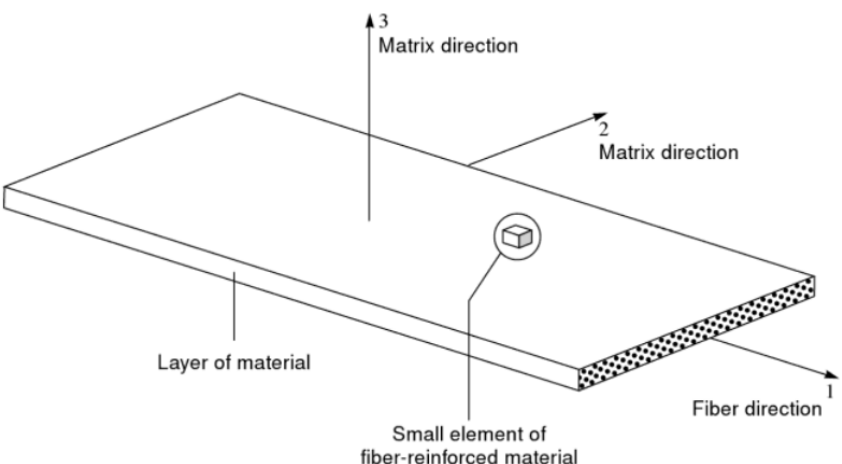


Figure 2.1: Material directions as resulting from orthotropic assumption[13]

Two prominent long fiber models are the iso-strain (Voigt) and iso-stress (Reuss) models. These are commonly referred to as "rule of mixture models" as they derive the composite's properties based on the weighted average of the individual components' properties. Voigt's model proposes that strains remain constant under loading, leading to identical displacements for both the matrix and the fiber. In contrast,

Reuss proposed that stresses are uniform throughout the composite material[10]. Both the Voigt and Reuss models assume seamless load transfer between the matrix and the fiber, without accounting for the interfacial properties of the composite. The Voigt model, typically applicable under axial loading, offers a theoretical upper bound on the composite's elastic properties. Meanwhile, the Reuss model is more pertinent under transverse loading and provides a lower bound[11]. By synthesizing insights from both models, it's feasible to estimate the Young's modulus in the longitudinal (E11) and transverse (E22) directions, as well as the shear modulus (G12) and the Poisson's ratio (V12)[10]. However, these models' accuracy wanes due to stress concentrations, especially near the fiber tips. Such discrepancies cause the iso-stress and iso-strain assumptions to falter, paving the way for the development of short fiber models.

2.1.2 Short Fiber Models

Compared to long fiber models, which primarily focused on the individual properties of the matrix and the fiber, short fiber models began to examine the effect of the interface. One of the initial models for short fibers depicted the composite as concentric cylinders, where the inner cylinder represented the fiber and the surrounding outer cylinder embodied the matrix. To account for the fiber volume fraction, these cylindrical representations were uniformly distributed or "tiled" across the designated area (see Fig. 2.2), maintaining a consistent ratio between the fiber and matrix radii[13].

Termed the "shear lag model", this approach aimed to depict the delay or "lag" in load transfer between the matrix and the fiber. Contrary to the instantaneous load-sharing presumed in long fiber models, the shear lag model illustrates the gradual distribution of stress, particularly evident at the matrix-fiber interface. Central to the development of this model, Cox proposed that while the matrix predominantly

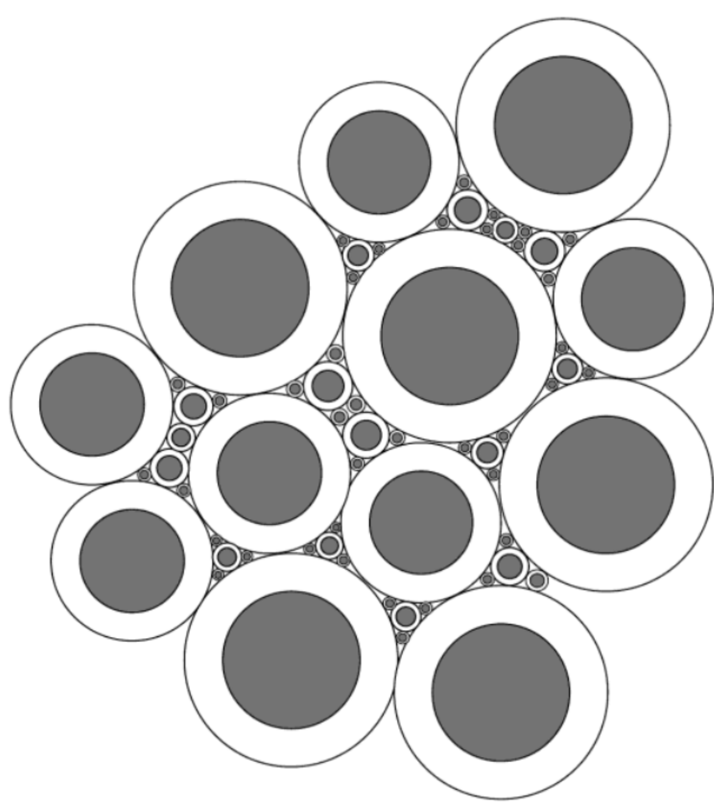


Figure 2.2: Tiled concentric cylinders as seen in the shear lag model

showcases homogeneous strain, discrepancies arise at the fiber interfaces. Here, the strain distribution isn't uniform, leading to the characteristic shear lag[3]. A pivotal, albeit simplifying, assumption of this model is the equivalence of transverse properties between the fiber and matrix, despite their differing axial properties. This presumption has implications for the model's accuracy and applicability, especially in scenarios where the transverse behaviors of the materials vary significantly.

Other approaches to the shear lag model include the Eshelby inclusion model, which models an analytical solution of an ellipsoidal inclusion in an infinite matrix [7], the Mori and Tanaka model, which expands on the Eshelby model with a mean-field theory to account for interactions among multiple inclusions[16]], the

work of Benveniste, which helps account for interphase zones between inclusions and the matrix[1], and work by Chou, who investigated the effects of biased fibers and the debonding at fiber ends on composite behavior [2]. With the advancements in computational power, simulating the microstructure response numerically to obtain material properties has become more accessible. For instance, finite element analyses of square and hexagonally arranged fibers can reveal how material properties change based on direction[13]. These computational advances have also paved the way for the development of intricate models that monitor the internal state of the system, a topic discussed in the subsequent section.

2.1.3 Non-ISV Failure Theories

Once material properties have been estimated, it is crucial to understand the potential failure mechanisms of the material. The initial and most straightforward approach is to use non-interactive criteria, wherein axial and shear stresses are considered independent of each other. Such failure criteria resemble yield theories in metals and often describe failure as the point when a maximum stress or strain threshold is reached [9].

On the other hand, interactive criteria take into account in-plane shear stress and strain to determine failure. A prominent example is the Tsai-Wu criterion, represented in Eq. 2.1[22]. This equation proposes that failure does not occur as long as the left side remains below unity, effectively weighing different contributions from shear and axial stresses. However, this model may not be suitable for materials exhibiting brittle fracture, such as carbon fiber/epoxy composites[8]. Moreover, since the coefficients of the Tsai-Wu criterion must be determined experimentally, it necessitates a significant amount of empirical data. To mitigate the need for extensive experimental data, Internal State Variable (ISV) models have been proposed.

$$F_1\sigma_1 + F_2\sigma_2 + F_6\tau_{12} + F_{11}\sigma_1^2 + F_{22}\sigma_2^2 + F_{66}\tau_{12}^2 + 2F_{12}\sigma_1\sigma_2 \leq 1 \tag{2.1}$$

Where:

- σ_1, σ_2 are the normal stresses.
- τ_{12} is the in-plane shear stress.
- $F_1, F_2, F_6, F_{11}, F_{22}, F_{66}$ and F_{12} are the Tsai-Wu material coefficients, determined through experimental testing.

2.2 Internal State Variable Theories

Given the limitations of non-physically based models, ISV (Internal State Variable) models have emerged. These models delve deeper into the interactions between materials and the underlying physics, striving to better understand both material properties and failure mechanisms. Instead of focusing on an averaged approximation of material behavior, ISV models consider the history of the material. This ensures that the preceding steps in the modeling process influence subsequent ones. A significant development in this field is through continuum damage mechanics (CDM), with pioneering work on composites being attributed to Talreja[21]. His model employs a vectorial approach to estimate microcracks within a representative volume element (RVE). This method doesn't merely quantify the extent of damage but also its direction. This directional insight is particularly beneficial for composite materials due to their anisotropic nature.

With the advancement in computational capabilities, more intricate numerical approaches have become feasible. Internal State Variable models can now be developed using computer simulations of RVEs. One critical point of potential failure at

the microscale of a composite isn't the matrix or fiber breakage, but the debonding of the interface between the matrix and the fiber. To investigate further the physical properties of this transition zone, Single Fiber Pull-Out (SFPO) tests have been conducted[4, 19, 18]. A diagram of what these tests look like can be seen in Fig. 2.3 as well as a common load-displacement curve found from these tests in Fig. 2.4.

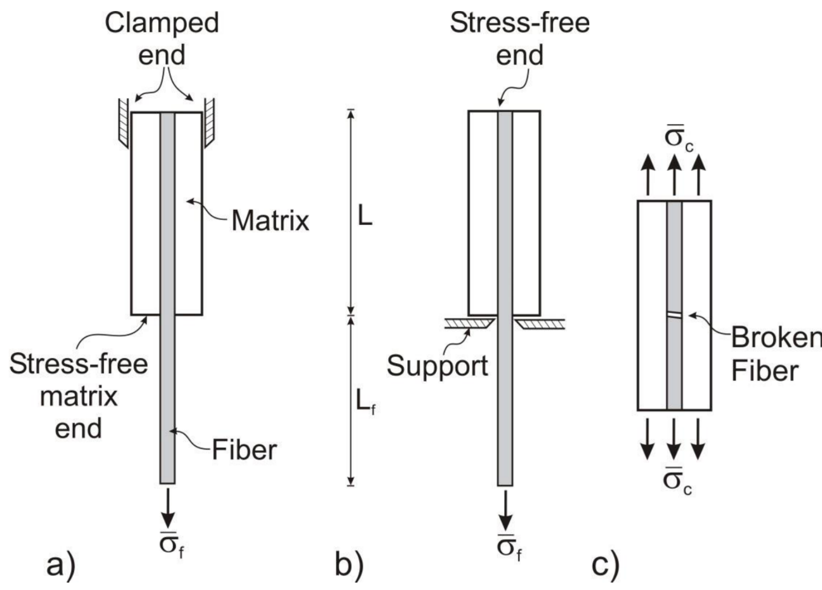


Figure 2.3: Single fiber pull-out testing methods (a) SFPO Type 1, (b) SFPO Type 2, single fiber fragmentation tests (SFFT)[20]

Although extensive testing exists in this domain, simulation results remain limited in scope. B.D. Ellis[6] explored the impact of varying fiber morphologies on pullout tests, while Jia[15] investigated the effects of thermal stresses during composite curing on pullout behavior. Notably, both these simulations and the majority of experimental tests focus on the extraction of a solitary fiber from its matrix, neglecting the influence of adjacent fibers. This study aims to shed light on the effects of adjacent fibers on pullout behavior. Should neighboring fibers significantly alter the pullout response, then conclusions drawn from SFPO tests might be inaccurate Moreover, without a comprehensive understanding of the interplay between neigh-

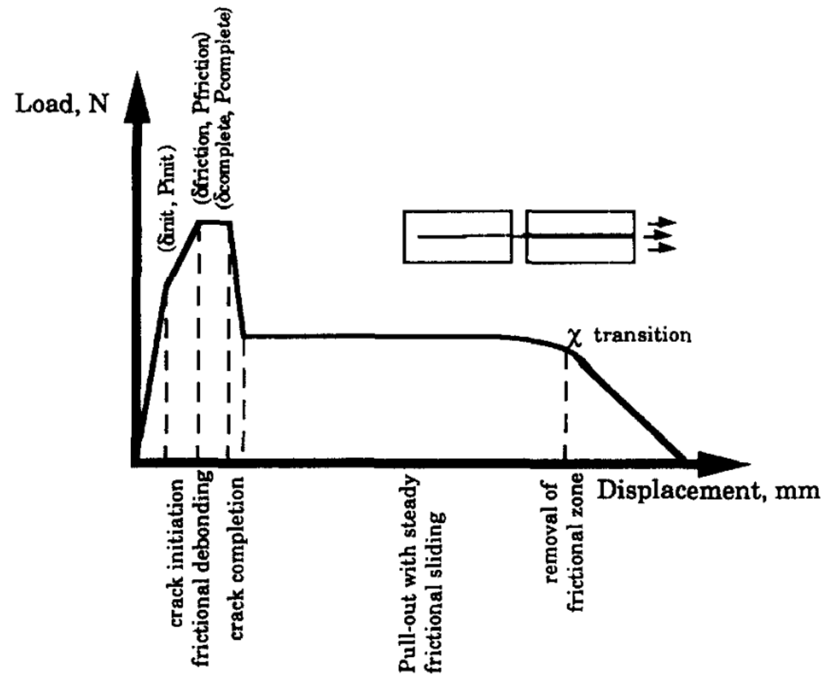


Figure 2.4: Typical load-displacement curve for pullout test[4]

boring fibers, interface zones, and pullout behavior, intricate multi-scale models risk relying on foundational assumptions that could skew their results.

Chapter 3

Methodology

To investigate the influence of neighboring fibers on pullout behavior, a series of models were created and analyzed using Abaqus standard. The primary objective of these studies was to understand the effect of fiber volume fraction (FVF) on the reaction force response of the pulled-out fiber. To achieve this, a representative volume element (RVE) was created employing a symmetric boundary condition. This chapter will detail the material properties and the specific setup of the model.

3.1 Materials

Composite materials, by their very nature, derive their mechanical properties from the properties of the different components, each contributing unique and vital properties to the final assembly. The behavior of such composites depends not only on the individual constituents but also on their interactions, interfaces, and orientations. Understanding the fundamental behavior of individual components, therefore, becomes important before undertaking any comprehensive analysis of the composite itself.

This section aims to describe the intrinsic properties of each material incorporated into the analysis. For each material, its characteristics are described. Furthermore, it will illustrate how each material is used and integrated into the composite structure. The arrangement and relationship of these components within the composite are visually depicted in Fig. 3.1.

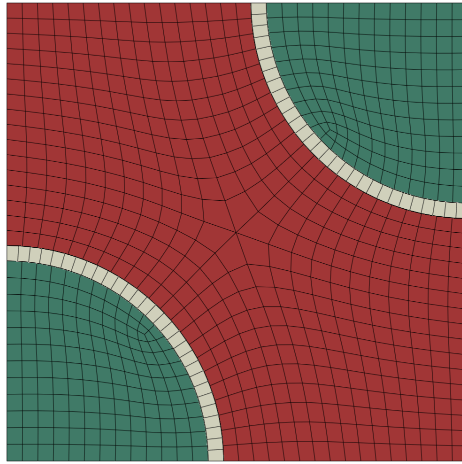


Figure 3.1: Materials arrangement in the model

Table 3.1: Material Color Table

Material	**Color**
COHESIVE	
CARBON_FIBER	
MATRIX	

3.1.1 Matrix

The matrix binds the fibers together and transfers stresses between them. The matrix in this case was modeled as purely elastic for the simplicity of the study. The properties can be found in Table 3.2. The material consists of two cylindrical cutouts where the fibers and cohesive zones are connected (Fig. 3.2).

Table 3.2: Matrix Material Properties [5]

Property	**Value**
E (Elastic Modulus)	3 GPa
Poisson's Ratio	0.3

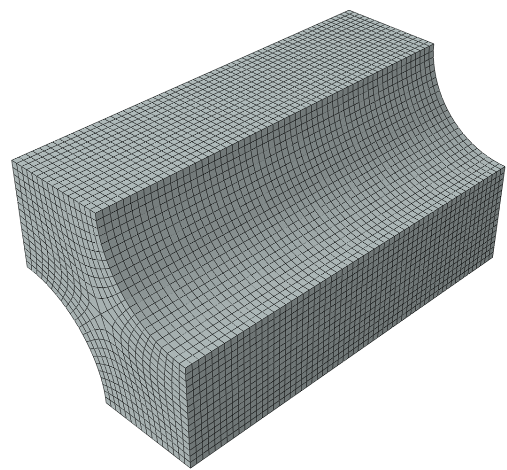


Figure 3.2: Matrix part

3.1.2 Fiber

Fibers provide the primary strength in composite materials. In this analysis, the fiber was modeled as carbon although a similar analysis could be done using glass or ceramic fibers. Since the focus of this study was on the interface zone, the fiber was modeled as purely elastic. The material properties of the fiber can be seen in table 3.3 and the geometry of the fiber can be seen in Fig. 3.3. It should be noted that the radius of the fiber is dependent on the fiber Volume fraction (FVF) of the model.

Table 3.3: Carbon Material Properties [5]

Property	**Value**
E (Elastic Modulus)	227 GPa
Poisson's Ratio	0.4

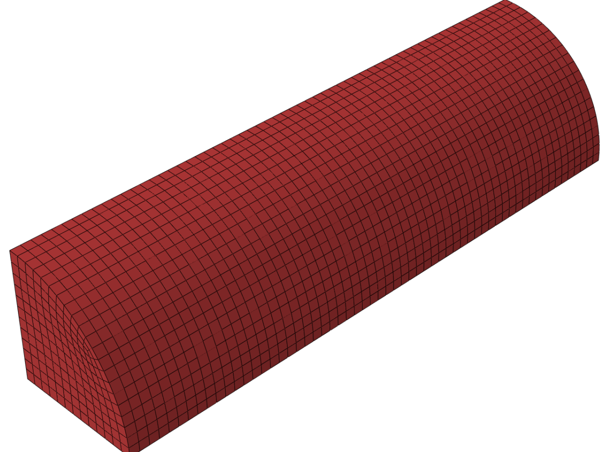


Figure 3.3: Fiber part

3.1.3 Cohesive elements

Cohesive elements are special traction-based elements in Abaqus. The elements were used to model the interface between the fiber and the matrix. The cohesive zone was modeled as a thin layer of elements between the matrix and the fiber (Fig. 3.4). Cohesive elements are useful in modeling this interface because they are allowed to fail (be deleted by the program) after a certain load and displacement is achieved. This causes the interface zone to weaken throughout loading and for crack propagation to occur.

Table 3.4: Cohesive Material Properties [5]

Property	**Value**
NS (Peak Traction)	10 MPa
E_{coh}	$1 \times 10^{11}\ N/mm^3$
$G1_{\text{ess}}$	$1 \times 10^{11}\ N/mm^3$
$G2_{\text{ett}}$	$1 \times 10^{11}\ N/mm^3$
Max Displacement	1
Viscosity	0.01

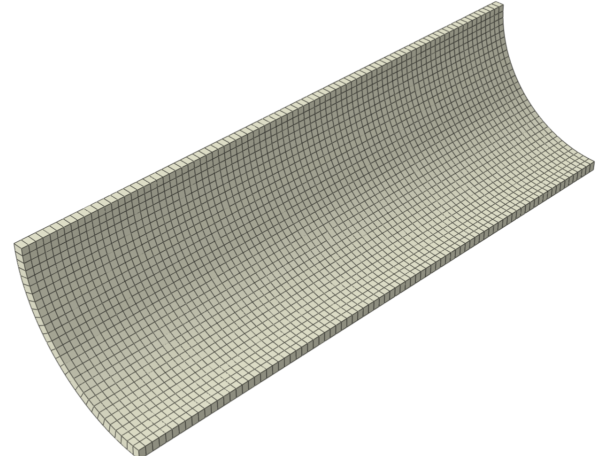


Figure 3.4: Cohesive part

Traction-Separation Law

Traction-separation laws play a pivotal role in characterizing the behavior of the interface zone between the matrix and fibers in composites, especially during debonding. This study employs such laws to model the cohesive elements representing the interface zone. The damage law, as represented graphically in Fig. 3.5, shows that no damage occurs until peak traction is reached at δ_0, after which damage progresses until δ_{sep}, signifying complete material failure.

For simplicity in this study a linear traction separation model was used as seen in Eq. 3.1 [14]. The equation has two inputs: a normal traction term and a shear term. The max function is used to ensure whichever of the terms holding a greater value is taken into account. Each of these terms is normalized by their critical stress value ensuring that when either reaches unity, damage starts to occur. The version used here contains a Macaulay bracket around the traction term to ensure no compressive stresses are accounted for.

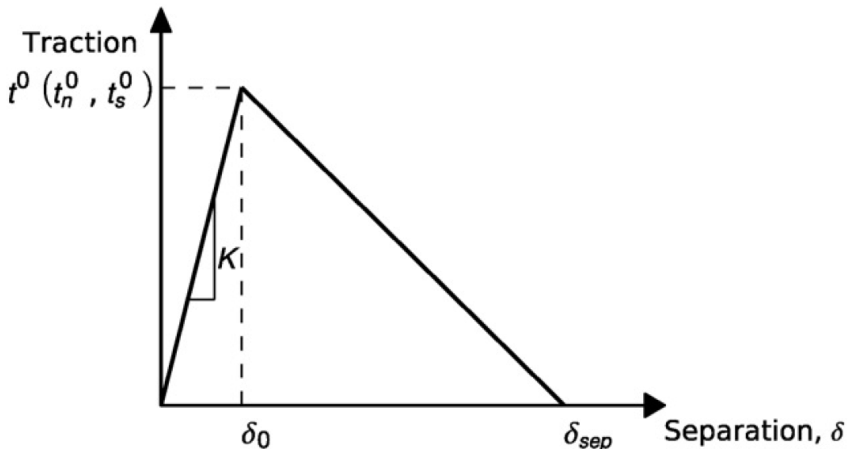


Figure 3.5: Traction separation law graphical representation [14]

$$\max\left\{\frac{\langle t_n\rangle}{t_n^0}, \frac{t_s}{t_s^0}\right\} = 1 \tag{3.1}$$

- t_n is the normal traction.
- t_n^0 is the critical tensile stress.
- t_s is the shear stress.
- t_s^0 is the critical shear stress.

An effective displacement can be equated using

$$\delta_m = \sqrt{\langle\delta_n\rangle^2 + \delta_s^2} \tag{3.2}$$

Where $\langle\delta_n\rangle$ is the tensile normal separation displacement and δ_s is the shear displacement. Given that, a damage variable can be computed by

$$D = \frac{\delta_m^f \left(\delta_m^{\max} - \delta_m^0\right)}{\delta_m^{\max} \left(\delta_m^f - \delta_m^0\right)} \tag{3.3}$$

Where:

- δ_m^f is the effective displacement at complete failure.
- δ_m^0 is the effective displacement at damage initiation.
- $\delta_m^{\max}$ is the maximum value of the effective displacement attained.

When D achieves unity the material has completely failed. In the case of cohesive elements in Abaqus, this means the element is deleted. The traction separation model is particularly useful for capturing the gradual degradation of interface properties up to the point of complete failure. However, it may not adequately describe the interface behavior under cyclic loading or when significant plasticity is involved at the interface.

3.1.4 Viscosity

It has been found that adding a viscosity parameter to the cohesive model reduces the likelihood of divergence during calculation. To help with convergence, a viscosity value of $\mu = 0.01$ was used for this work [5].

3.2 Model Setup

3.2.1 Boundary Conditions

There are 5 instances in the model. Two fiber instances, two cohesive, and one matrix. These instances are arranged in the model as a representative volume element (RVE) as shown in Fig. 3.6 and Table 3.5. Tie constraints were applied on four surfaces of the model. These surfaces are:

- The surface between the bottom fiber and its neighboring cohesive zone
- The surface between the bottom cohesive zone and the matrix
- The surface between the top cohesive zone and the matrix
- The surface between the top fiber and its neighboring cohesive zone

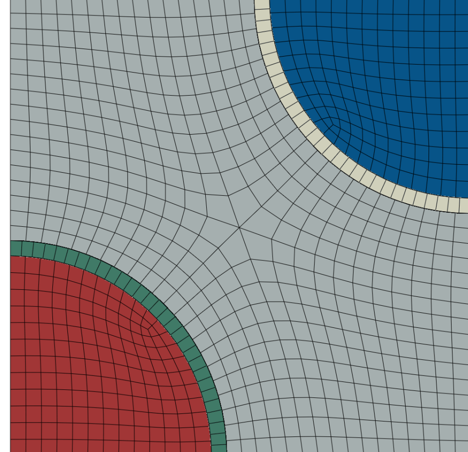


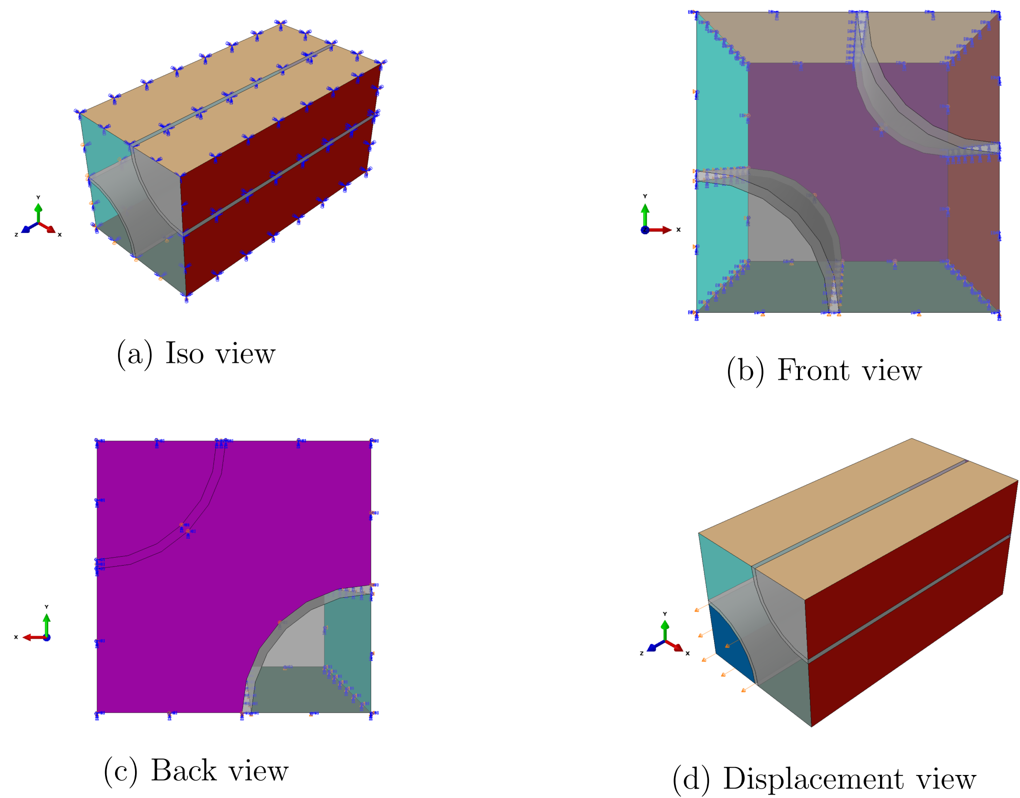
Figure 3.6: Different model instances

The boundary conditions of the model are presented in Table 3.7 and Fig. 3.7. The constraints were chosen to ensure the symmetry of the model remained intact

Table 3.5: Instance Table

Instance Name	**Color**	**# Elements**	**# Nodes**	**Element type (# elements)**
FIBER-2		9120	10736	C3D8R : (9120)
FIBER-1		9120	10736	C3D8R : (9120)
COH-1		2494	5220	COH3D8 : (2494)
COH-2		2494	5220	COH3D8 : (2494)
MATRIX-1		27720	31537	C3D8R : (27720)

while allowing for Poisson's effects to take place. One may notice that the bottom fiber of the model is unconstrained. This is to allow for the pullout behavior to be studied.



(a) Iso view

(b) Front view

(c) Back view

(d) Displacement view

Figure 3.7: Model boundary conditions

Table 3.6: Boundary Conditions

	Side	Description
	Left	Can extend along the yz-plane but no movement in the z direction.
	Bottom	Can extend in the xz-plane but no movement in the y direction.
	Back	Can extend in the xy-plane but no movement in the z direction.
	Top	Constrained to not rotate around the x, y, or z axis but can move laterally in any direction.
	Right	Constrained to not rotate around the x, y, or z axis but can move laterally in any direction.
	Displacement	Set displacement to pull the fiber out of the matrix.

3.2.2 Mesh Generation

Meshes were generated for each part individually. For the fiber and matrix parts, C3D8R elements were used whereas for the cohesive zones, a COH3D8 element was used (see Table 3.5). Although seeds and element sizes varied depending on the thickness of the cohesive elements, generally a seed of 0.35 was used for the cohesive layers and a seed of 0.5 was used for all other parts. This was found to give an accurate model without increasing runtime by too much.

3.2.3 Solver Configuration

A standard abaqus solver was used in the study. However, non-linear geometry was turned on to help with the convergence of the cohesive elements. A single step after the initial was used in the study. More information can be found in Table 3.7.

Table 3.7: Step Data

Step	**Value**
Increments completed	15
Minimum time increment	1e-05
Step time completed	1.0
Analysis type	Standard
Maximum time increment	1.0
Initial time increment	0.01
Matrix solver	DIRECT_SOLVER
Time Period	1.0
Maximum number of increments	100000

Chapter 4

Results

4.1 Rule of Mixtures (ROM)

A preliminary rule of mixtures model was created to validate the cohesive model. Initially, the model was run without any cohesive elements to verify the behavior of the fiber and matrix materials. Subsequently, cohesive elements were introduced to assess the effect of the cohesive material on the ROM calculation.

The Rule of Mixtures provides an estimate for the modulus of elasticity of a composite material, considering the contributions from its constituents, the fibers and the matrix. The modulus of elasticity of the composite, E_c, is given by:

$$E_c = f \cdot E_f + (1 - f) \cdot E_m \tag{4.1}$$

Where:

- E_f denotes the modulus of elasticity of the fibers.

- E_m represents the modulus of elasticity of the matrix.
- f is the volume fraction of the fibers.

Given:

$$E_f = 227 \times 10^{15} \, \mu Pa$$
$$E_m = 3 \times 10^{15} \, \mu Pa$$
$$f = 0.3$$

Substituting into the ROM equation yields:

$$E_c = 0.3 \cdot 227 \times 10^{15} \, \mu Pa + (1 - 0.3) \cdot 3 \times 10^{15} \, \mu Pa$$
$$E_c = 7.02 \times 10^{16} \, \mu Pa$$

For the models, a strain $\epsilon = 0.001\%$, was applied, chosen to ensure the model operates in a low-strain regime. Given the model length of $30\mu m$, the required displacement to induce this strain is:

$$d = 0.001\% \times 100 \times 30 = 0.003 \, \mu m \tag{4.2}$$

All the same boundary conditions were used as in other models except that the entire front face is displaced and the entire back face's plane is fixed in the axial direction (Fig. 4.1). For ease of modeling, the tie constraints between the cohesive layer and the other parts are used, but for the no cohesive case, the material is changed to matrix.

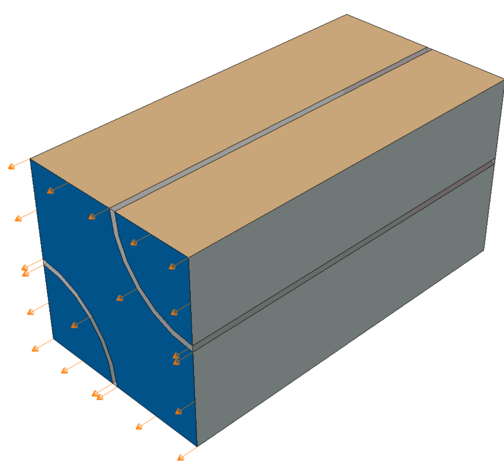


Figure 4.1: Displacement of rule of mixtures models

4.1.1 Without cohesion

For the case without cohesion, the maximum force was measured as $F_{\text{max}} = 1.57803 \times 10^{15}$ μN. To verify the model, the ratio of the force to the total cross-sectional area gives the stress:

$$\text{Stress} = \frac{F_{\text{max}}}{\text{Area}} = \frac{1.57803 \times 10^{15}\ \mu\text{N}}{225\ \mu\text{m}^2} = 7.013 \times 10^{12}\ \mu\text{Pa} \tag{4.3}$$

Further, using stress and strain, we can compute the modulus of elasticity E:

$$E_{\text{cal}} = \frac{\text{Stress}}{\text{Strain}} = \frac{7.013 \times 10^{13}\ \mu\text{Pa}}{0.0001} = 7.013 \times 10^{16}\ \mu\text{Pa} \tag{4.4}$$

Compared to the calculation the percent error is 0.1%. Thus the model does show verified results without the cohesive layer.

4.1.2 With cohesion

For the case with cohesion, the maximum force was measured as $F_{\text{max, cohesive}} = 1.57474 \times 10^{15}\ \mu\text{N}$. Going through a similar calculation as before:

$$\text{Stress} = \frac{F_{\text{max, cohesive}}}{\text{Area}} = \frac{1.57474 \times 10^{15}\ \mu\text{N}}{225\ \mu\text{m}^2} = 6.999 \times 10^{12}\ \mu\text{Pa} \tag{4.5}$$

Further, using stress and strain, we can compute the modulus of elasticity E:

$$E_{\text{cal, cohesive}} = \frac{\text{Stress}}{\text{Strain}} = \frac{6.999 \times 10^{12}\ \mu\text{Pa}}{0.0001} = 6.999 \times 10^{16}\ \mu\text{Pa} \tag{4.6}$$

Compared to the calculation, the cohesive model shows a percent error of 0.3%. This shows that even with cohesive elements the model acts according to theory. However, one point of interest is the $E_{\text{cal, cohesive}}$ does show a lower value than both cases. This checks with intuition since the cohesive elements create a weaker zone in the composite model.

4.2 Fiber Analysis

A parametric study was executed to explore the influence of varying fiber volume fraction (FVF) during a pullout test. The study involved generating models with distinct geometric attributes and assigning appropriate material properties. After defining geometric conditions, simulations were run using Python scripts, examples of which can be found in appendix B.1. Initially, the study focused on a matrix containing a single fiber. This was followed by a scenario with two fibers embedded in the matrix. To probe the impact of the cohesive zone, a dual-fiber setup was analyzed without the second cohesive layer.

The load-displacement graphs for all cases were obtained by summing up the reaction forces in the pulling direction (RF3) on the face of the pulled-out fiber. This gives the load-displacement curve that theoretically would be obtained by experiment (the amount of load needed to be applied on the fiber to pull it from the matrix).

4.2.1 Single fiber

The single fiber case consisted of just one fiber, cohesive zone, and matrix. This case was used as the control for the study since most pullout tests are done with no influence from neighboring fibers. If a difference was found between the single and double fiber cases, then the influence of FVF could be shown[1]. Figure 4.3 shows the results for these tests. As seen, the peak stress increases as the FVF increases due to an increase in load capability given from the greater area of the fiber. Contour plots for the single fiber cases for peak stress and failure are shown in Figs. 4.6 and 4.9, respectively.

[1] FVF are calculated for the double fiber case and listed here for comparison. True FVF for the single cases is half of what is shown although the fiber radii are the same between cases

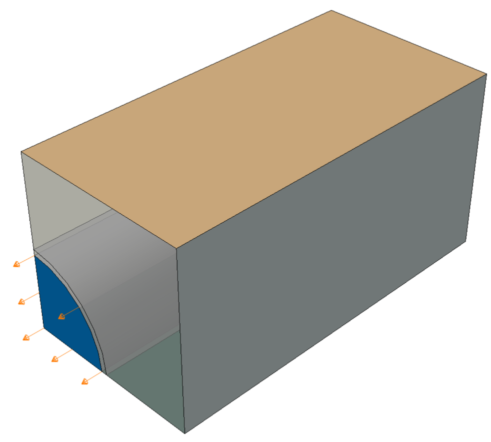


Figure 4.2: Setup for single fiber case

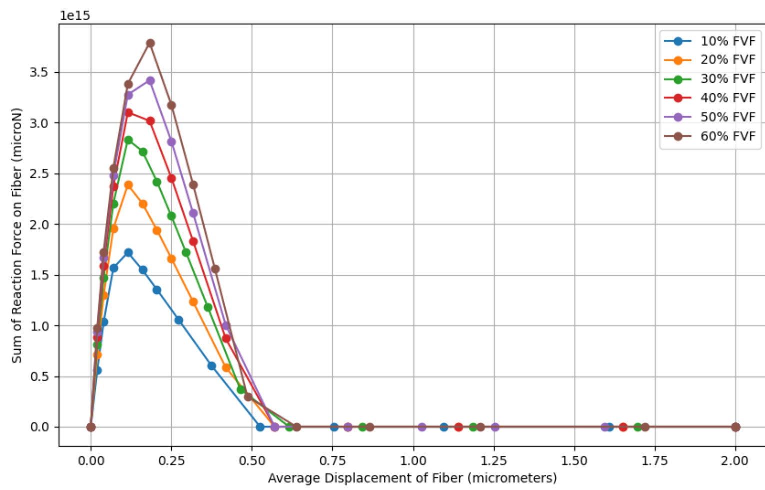


Figure 4.3: Reaction force of single fiber case with different FVF

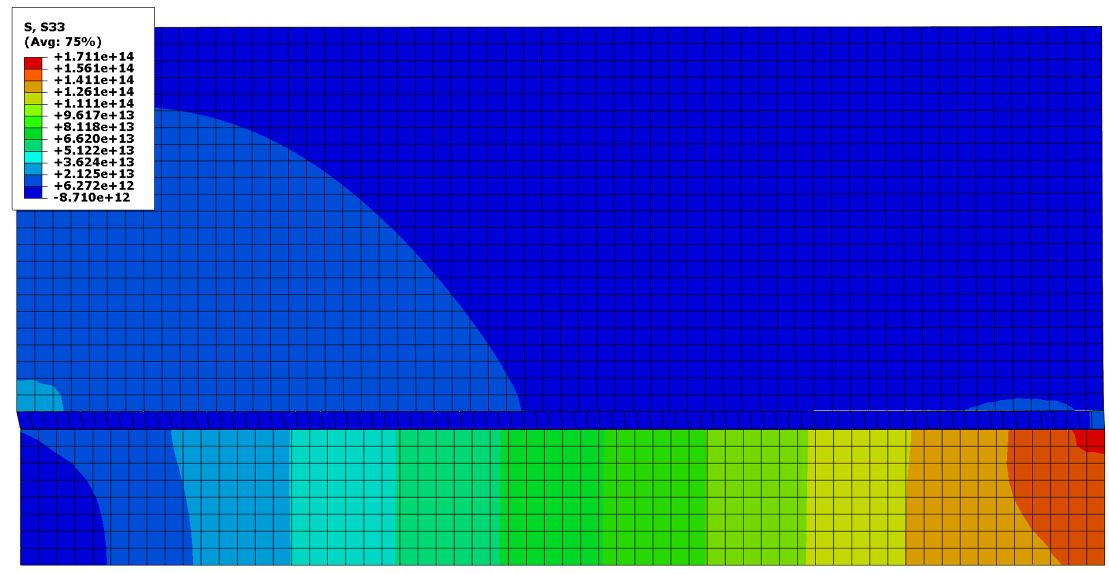


Figure 4.4: 10% FVF at peak stress

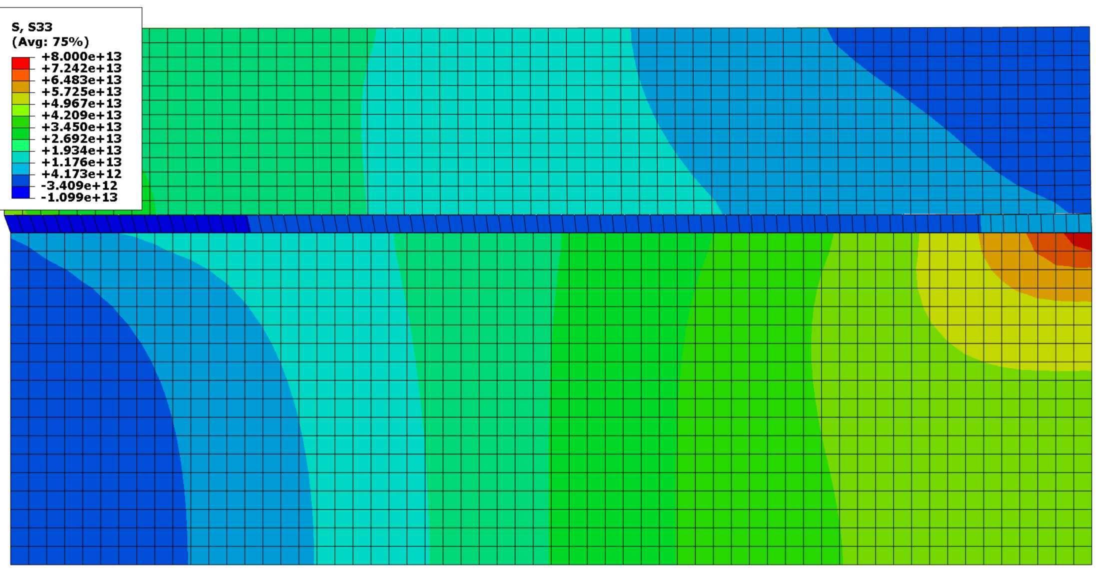


Figure 4.5: 60% FVF at peak stress

Figure 4.6: Single fiber contour plots for s33 - Peak Stress

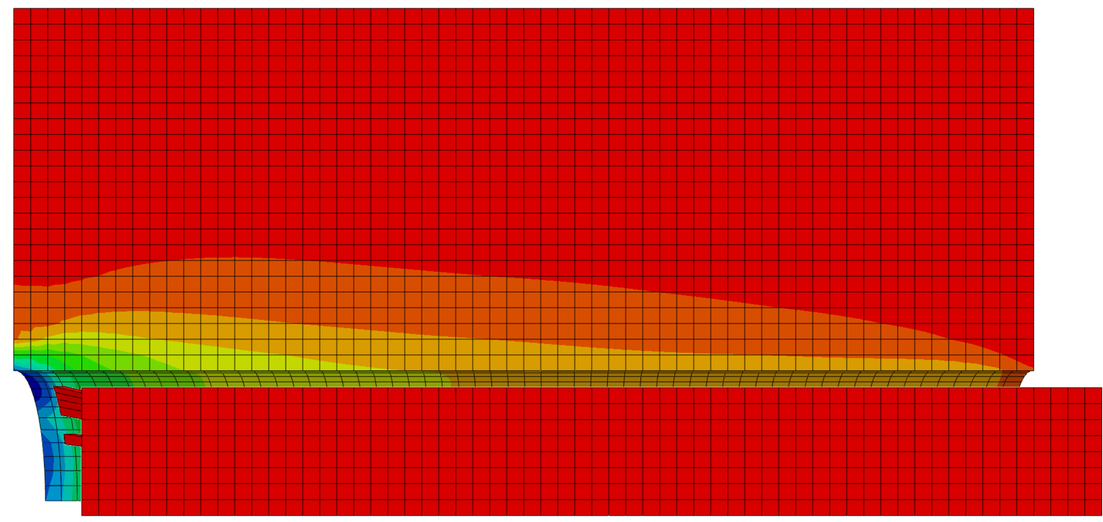


Figure 4.7: 10% FVF at failure

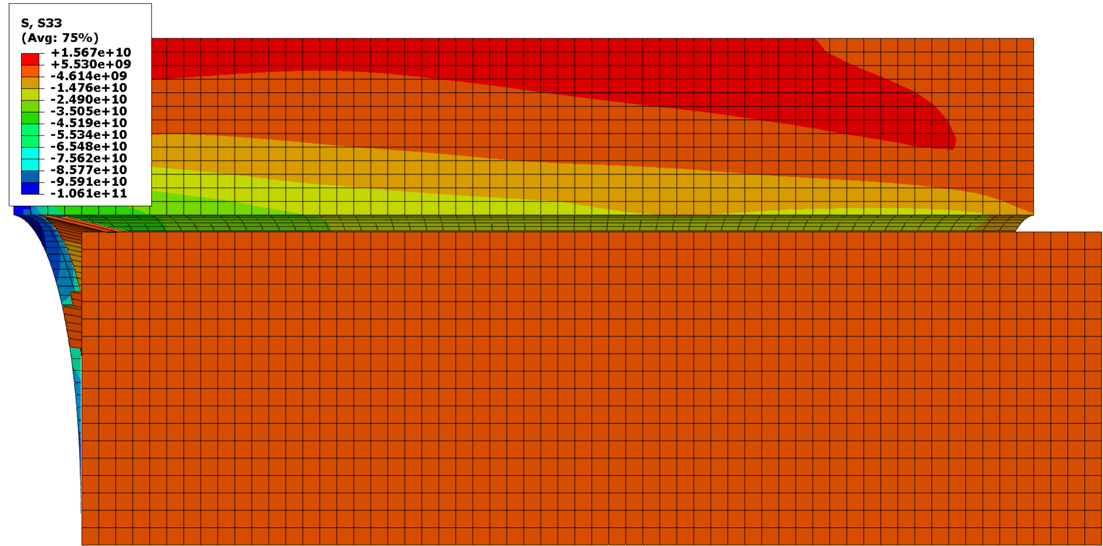


Figure 4.8: 60% FVF at failure

Figure 4.9: Single fiber contour plots for s33 - Failure

4.2.2 Single vs Double

The results from the double fiber case are shown in Fig. 4.10. As seen, the results are similar to the single case where the increase in FVF results in a higher peak reaction force. However, when graphed on the same chart as the single fiber cases differences emerge (Fig. 4.11). For the 10% FVF case, very little changes between the single and double fiber case are observed. However, as the FVF increases the difference becomes noticeable. Specifically, this difference becomes very apparent at 60% FVF where the single fiber case takes less load to be pulled out. This suggests that with high FVF cases, the pullout behavior does get affected by neighboring fibers, although it seems unaffected by low FVF. This finding casts doubt on how a SFPO measurement can accurately characterize the interfacial behavior in actual composite materials, where fibers are normally densely distributed. Contour plots for the double fiber cases for peak stress and failure are shown in Figs. 4.12 and 4.13, respectively.

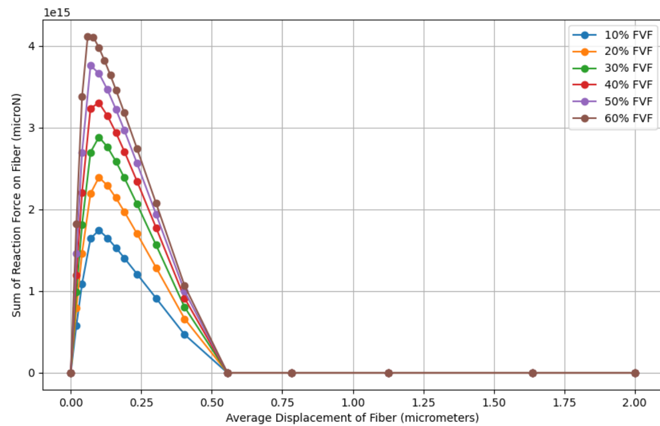


Figure 4.10: Reaction force of double fiber case with different FVF

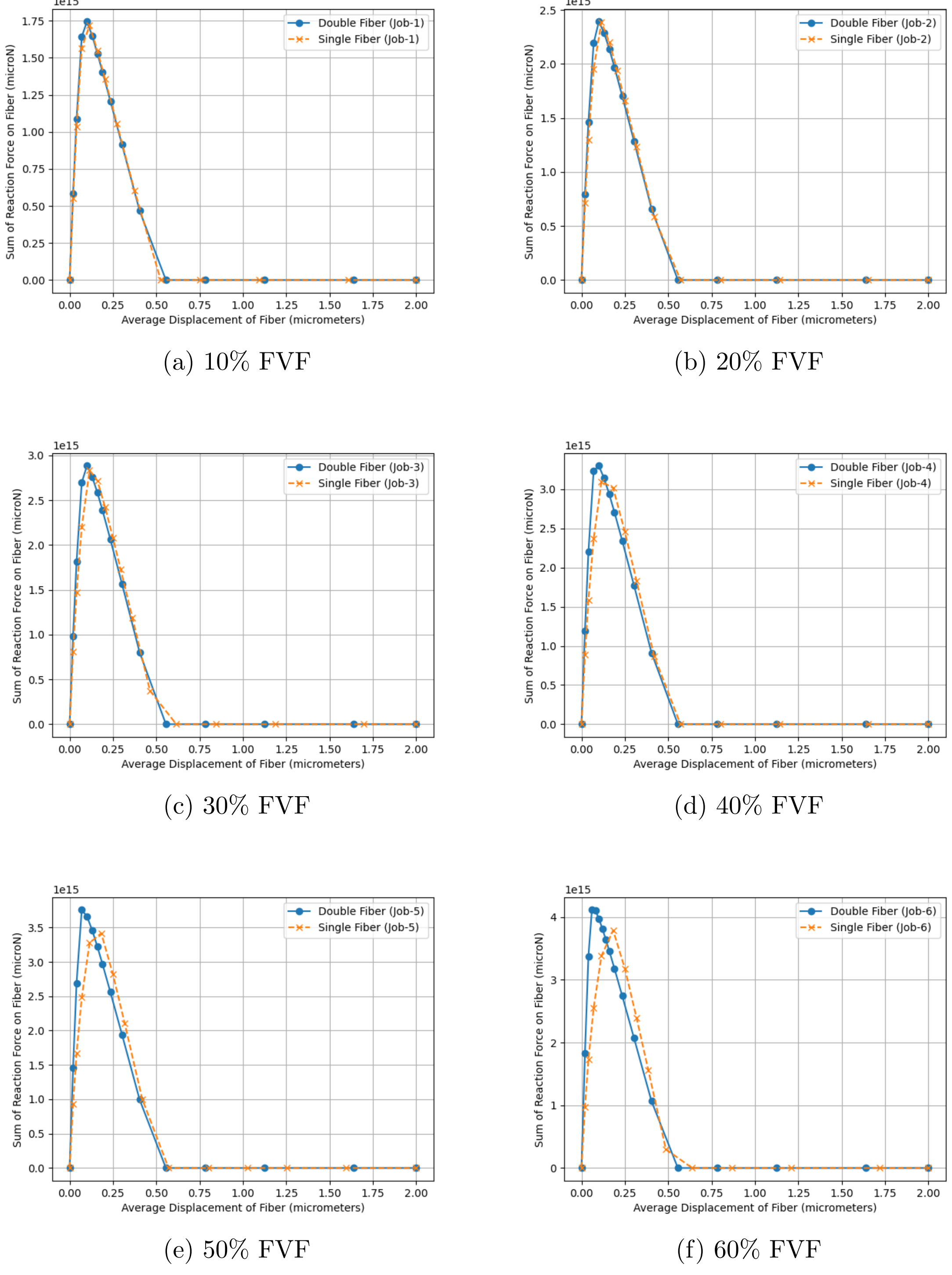
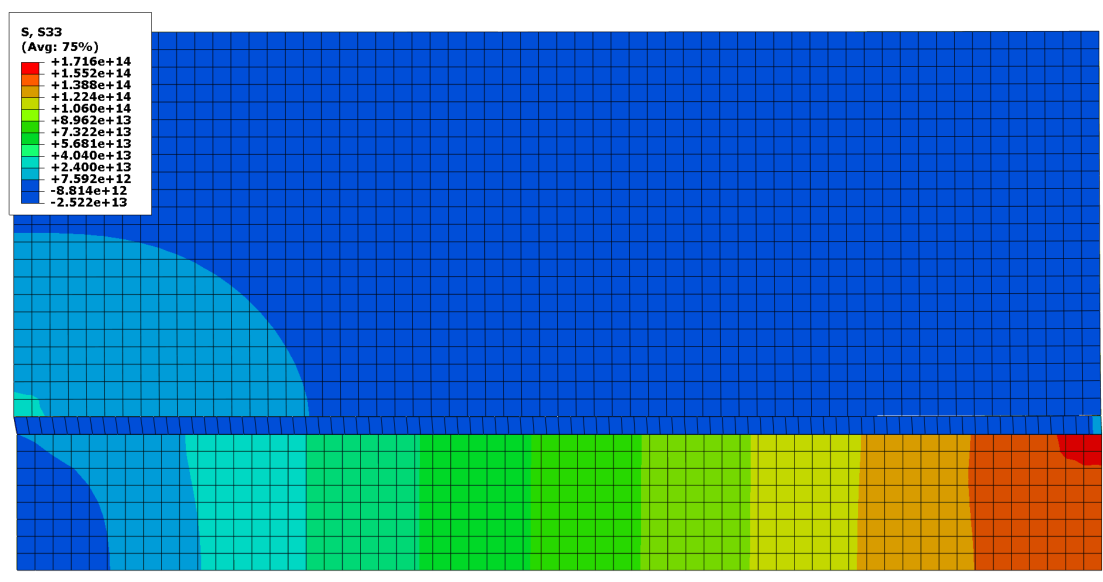
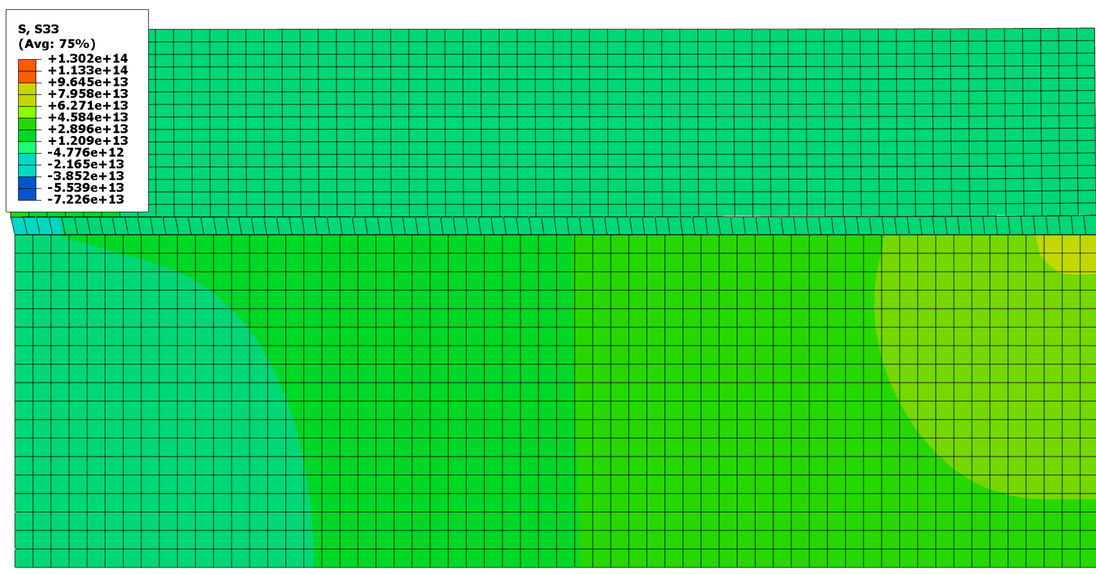


Figure 4.11: Double fiber vs single fiber for different FVF

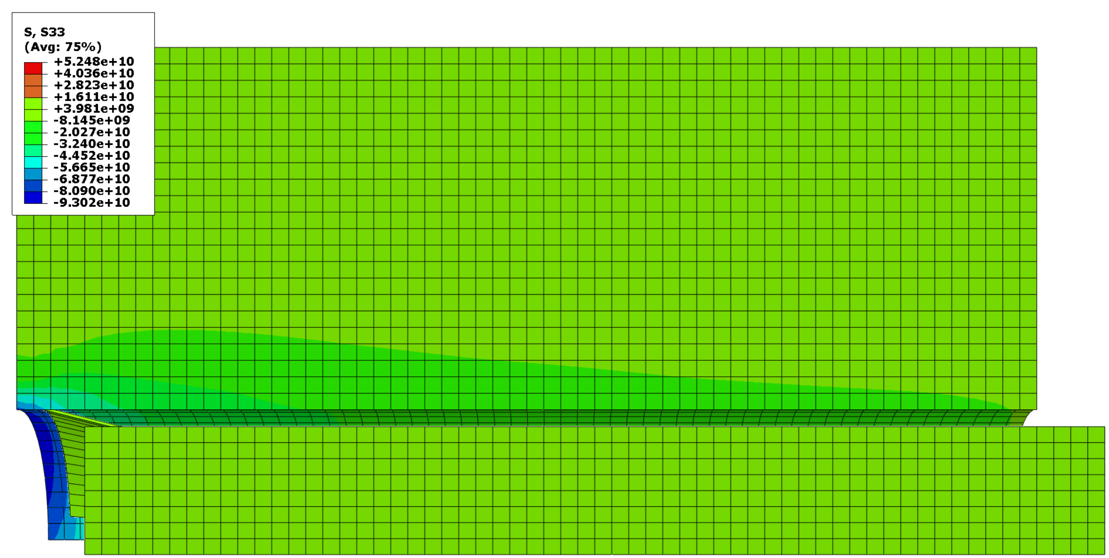


(a) 10% FVF at peak stress

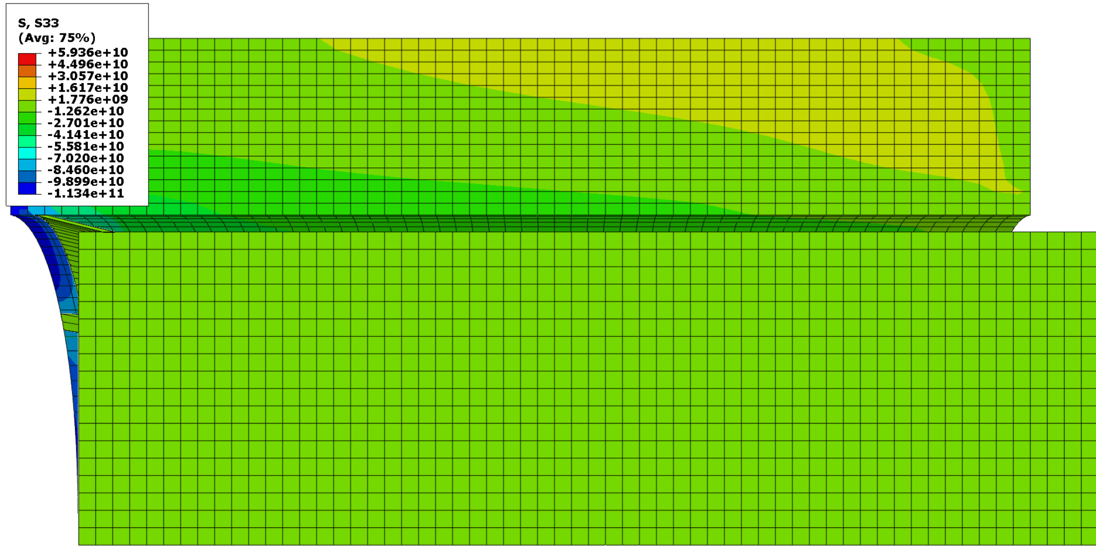


(b) 60% FVF at peak stress

Figure 4.12: Double fiber contour plots for s33 at peak stress



(a) 10% FVF at failure



(b) 60% FVF at failure

Figure 4.13: Double fiber contour plots for s33 at failure

4.2.3 Single vs Double vs Double without Cohesion

To test and see if the cohesive layer plays a part in the behavior of the pullout test with double fibers, a test with two fibers but only one cohesive zone was performed(Fig. 4.14). The results shown in Fig. 4.15 are surprising. The double fiber without the second cohesive zone shows a lower peak reaction force than even the single fiber case. This shows that the cohesive zone along with the extra fiber plays a role in the load transfer. However, as of the point of this writing, no physical explanation could be found. This seems to be a point of interest for further research and should be kept in mind in the future for similar cohesive zone models.

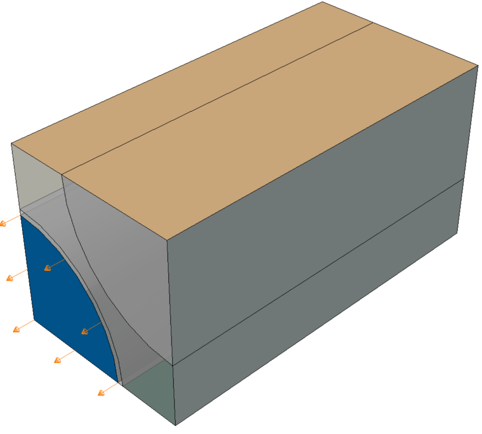


Figure 4.14: Setup for double fiber with single cohesive zone

4.3 Cohesive Properties

To test the effect, if any, of the cohesive material properties on the influence of FVF on the pullout behavior, several parametric studies were performed. The first entailed varying the maximum displacement parameter, the second looked at the max traction value, and the third investigated varying the cohesive thickness.

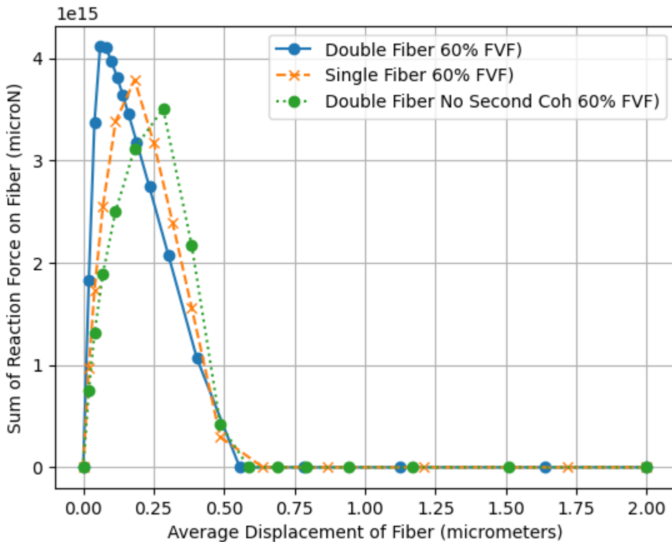


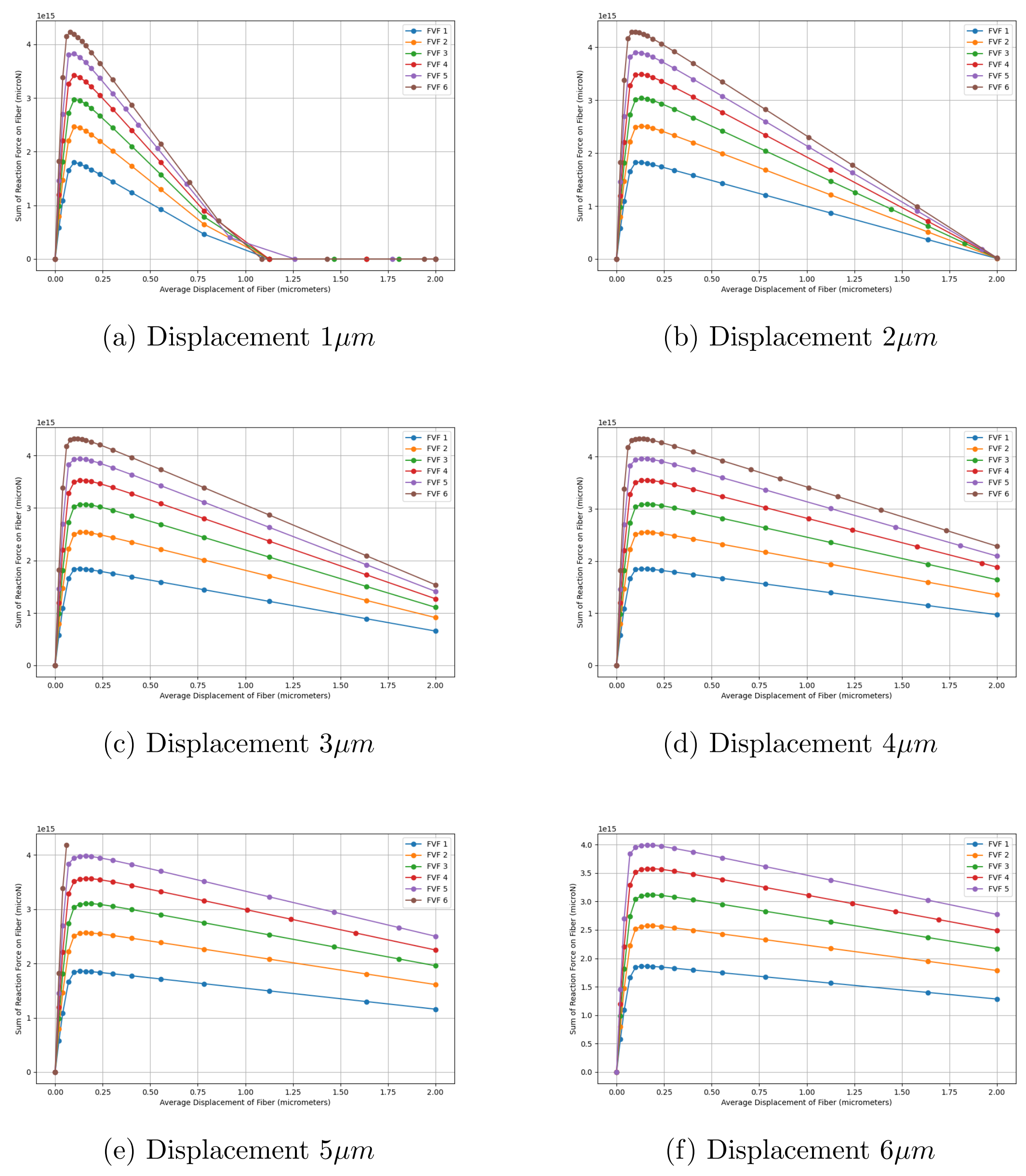
Figure 4.15: Reaction force of single fiber, double fiber, and double fiber without the second cohesive zone case

4.3.1 Maximum Displacement

The max displacement value at failure was varied from $1\mu m$ to $6\mu m$. The maximum applied pulling displacement on the fiber was held at a constant $2\mu m$. As seen in Fig. 4.16, the different FVF values had little effect on the shape of the force-displacement curve although it did vary the stress-carrying behavior after the initiation of interfacial damage. However, it does seem like the decreasing slope of the curve is greater in the 60% FVF case rather than the 10% FVF case. This makes sense given that the max load reached is higher so to get to the same displacement value the curve would have to be steeper.

4.3.2 Maximum Traction

The max traction value was varied from $1 \times 10^{12} \mu Pa$ to $6 \times 10^{12} \mu Pa$. This is based on various experimental values with the first providing a low-end estimate and the second providing a high-end estimate [17, 5]. The failure displacement values were kept constant at $1 \mu m$. The results are shown in Fig. 4.17. It can be seen that the shape of the graphs is greatly affected by the max traction value. However, there isn't much difference besides the height of the graph between the 10% and 60% FVF cases. As shown in Fig. 4.18, by increasing the max traction value the curve starts to "lag". This lag is more prominent with the 10% FVF case than the 60% FVF case.



(a) Displacement $1\mu m$

(b) Displacement $2\mu m$

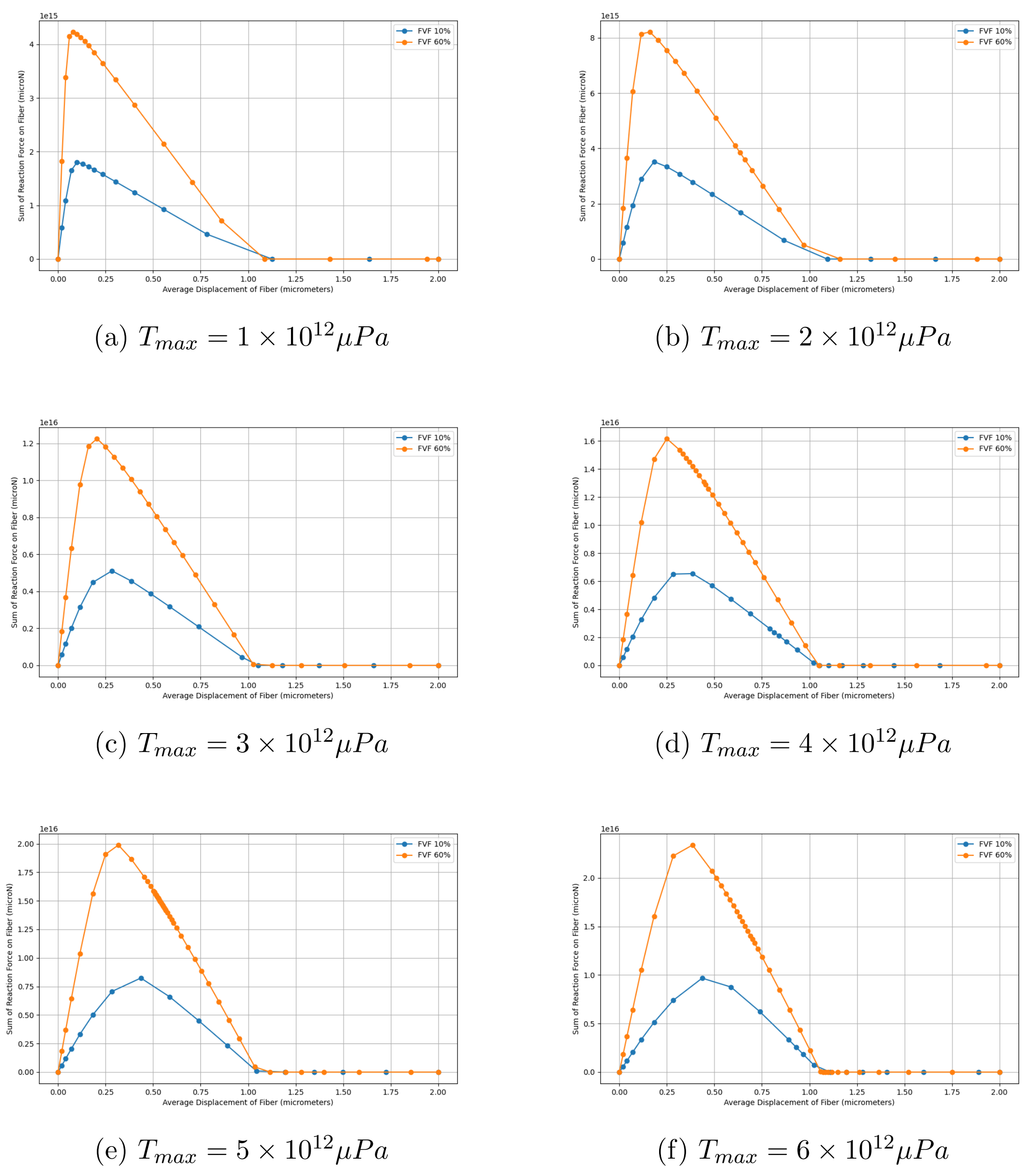
(c) Displacement $3\mu m$

(d) Displacement $4\mu m$

(e) Displacement $5\mu m$

(f) Displacement $6\mu m$

Figure 4.16: Change in load response for different FVF and different max displacements.



(a) $T_{max} = 1 \times 10^{12} \mu Pa$

(b) $T_{max} = 2 \times 10^{12} \mu Pa$

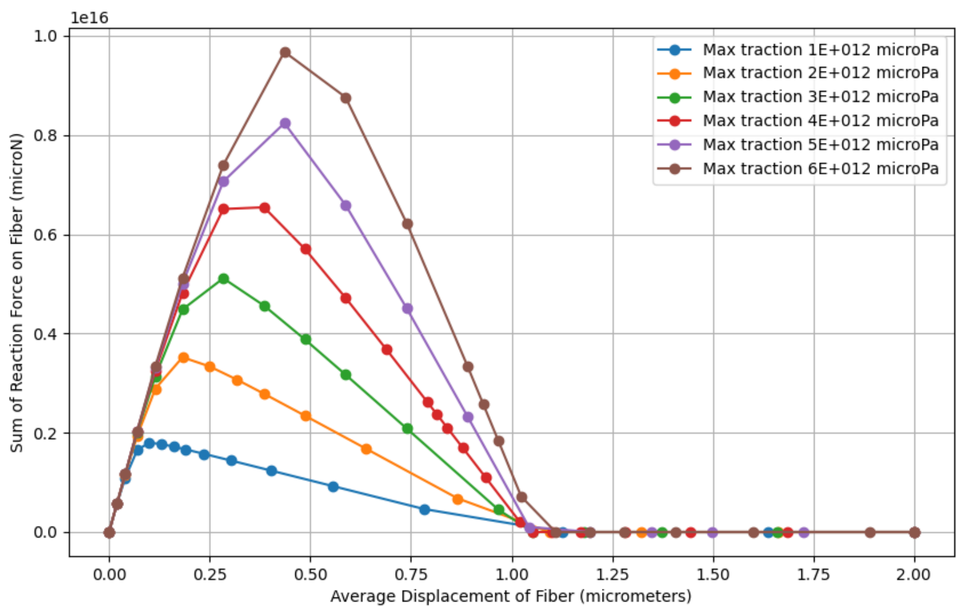
(c) $T_{max} = 3 \times 10^{12} \mu Pa$

(d) $T_{max} = 4 \times 10^{12} \mu Pa$

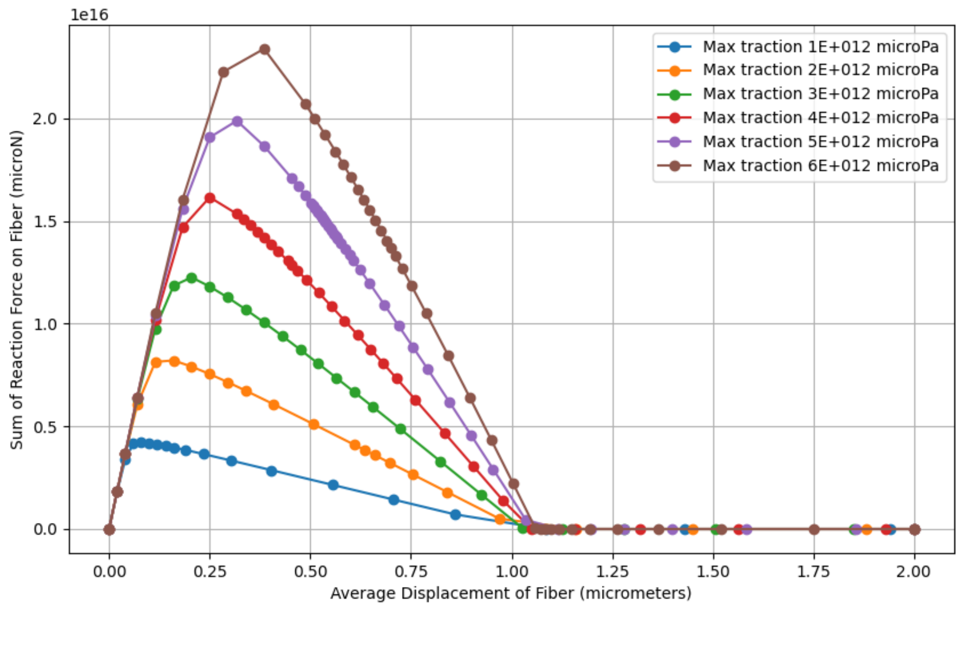
(e) $T_{max} = 5 \times 10^{12} \mu Pa$

(f) $T_{max} = 6 \times 10^{12} \mu Pa$

Figure 4.17: Change in load response for 10% and 60% FVF for different max traction values.



(a) 10% FVF



(b) 60% FVF

Figure 4.18: Change in load response for 10% and 60% FVF.

4.3.3 Thickness

The thickness of the cohesive layer varied from $0.1\mu m$ to $1.0\mu m$ as seen in Fig. 4.19. Varying the cohesive thickness had relatively little effect on the fiber response. This implies that, in modeling the interfacial behavior using the cohesive layer approach, the choice of the cohesive layer thickness generally does not play a critical role.

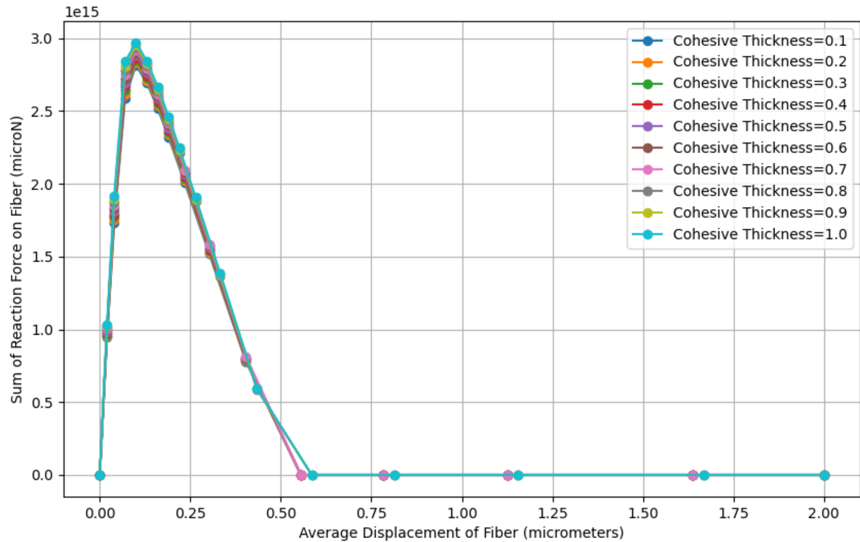
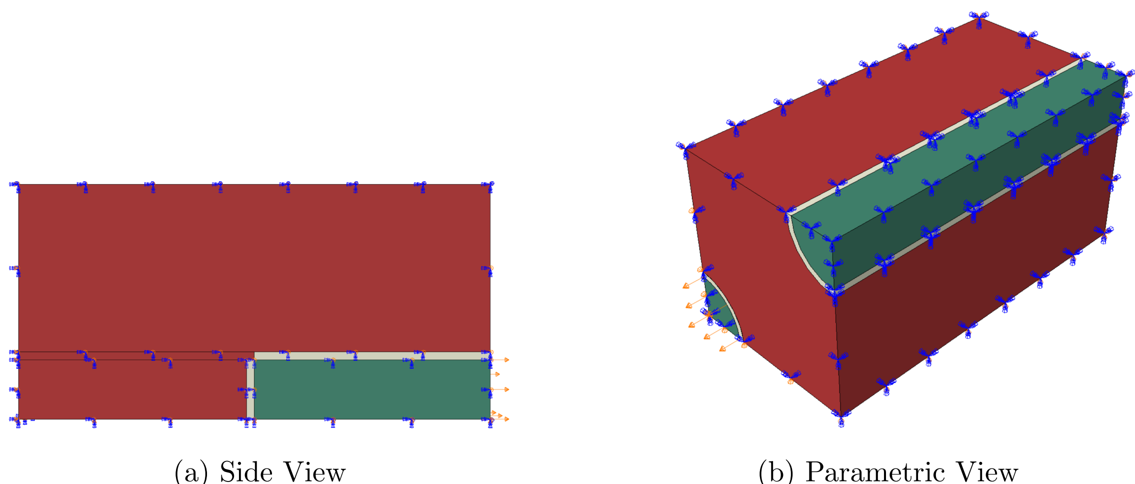


Figure 4.19: Reaction force of double fiber with varying cohesive zone thickness

4.4 Embedded Fiber

To gain a deeper understanding of how the placement of the first fiber influences the pullout behavior, an embedded fiber case was investigated as shown in Fig. 4.20. The first fiber was embedded within the matrix material, with half of it being fiber material and the remaining part matrix. A cohesive layer was sandwiched between each material. The entire back face was restrained from moving in the z-direction.



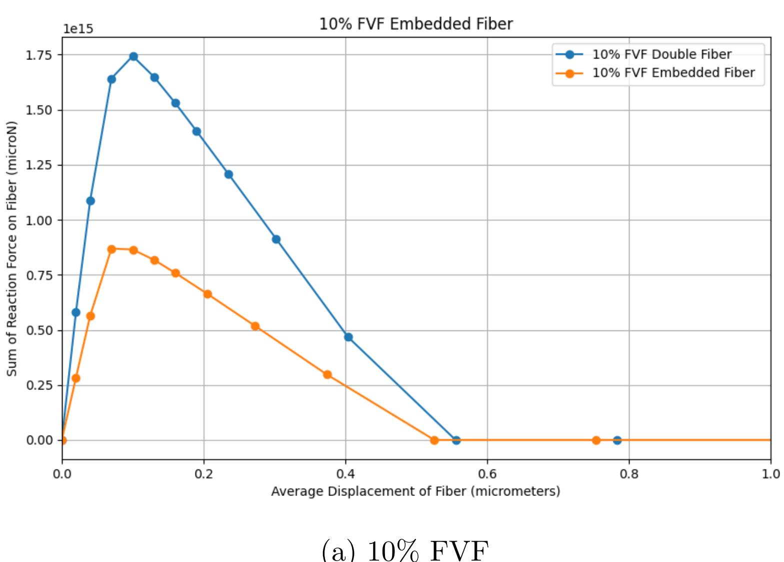
(a) Side View

(b) Parametric View

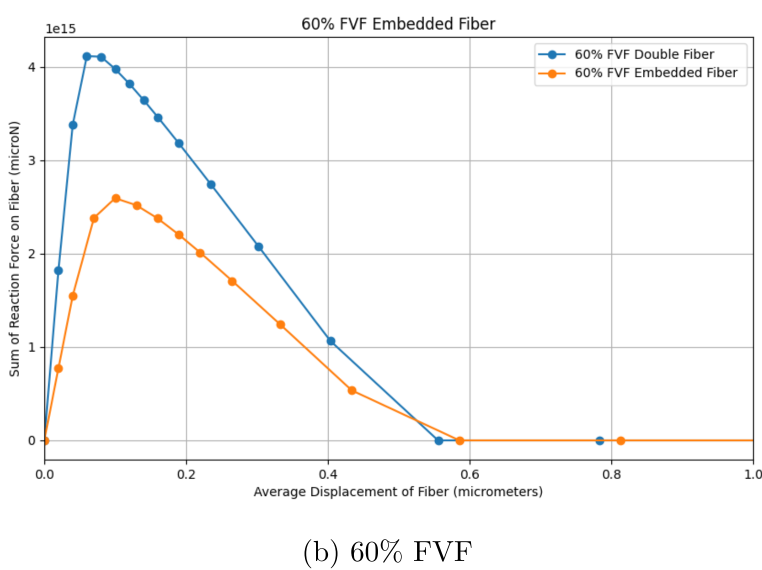
Figure 4.20: Model setup for embedded fiber case. Matrix material is shown in red, cohesive in cream, and fiber in green.

The embedded fiber case demonstrates two main distinctions when juxtaposed with the double fiber case (Fig. 4.21). Firstly, the peak traction in both the 10% and 60% FVF cases differ from their double fiber counterparts. This discrepancy can likely be attributed to the reduced number of cohesive elements interacting with the embedded fiber compared to the double fiber scenario. Secondly, the gradient leading to the peak traction in the embedded fiber cases is less pronounced than in the double fiber cases. This observation could also stem from the reduced involvement of cohesive elements. Interestingly, for the 60% FVF case, all the axial elements at the back were removed by the end of the simulation. In contrast, these elements remained intact in the 10% FVF scenario, as illustrated in Fig. 4.25. This pattern suggests a more pronounced load transfer to the axial elements in the high FVF scenario. A potential area of exploration could be to adjust the peak shear and axial traction values. While these were kept consistent in the current study, varying them might lead to distinct behaviors in axial versus radial elements. Modifying the positioning of the second fiber to create a skew-symmetric scenario (with half of the second fiber extending in the z-direction and the other half forming the matrix) could offer

insights into different load-bearing capacities and provide a closer representation of a chopped fiber scenario.



(a) 10% FVF



(b) 60% FVF

Figure 4.21: Comparison of single fiber case and embedded fiber case

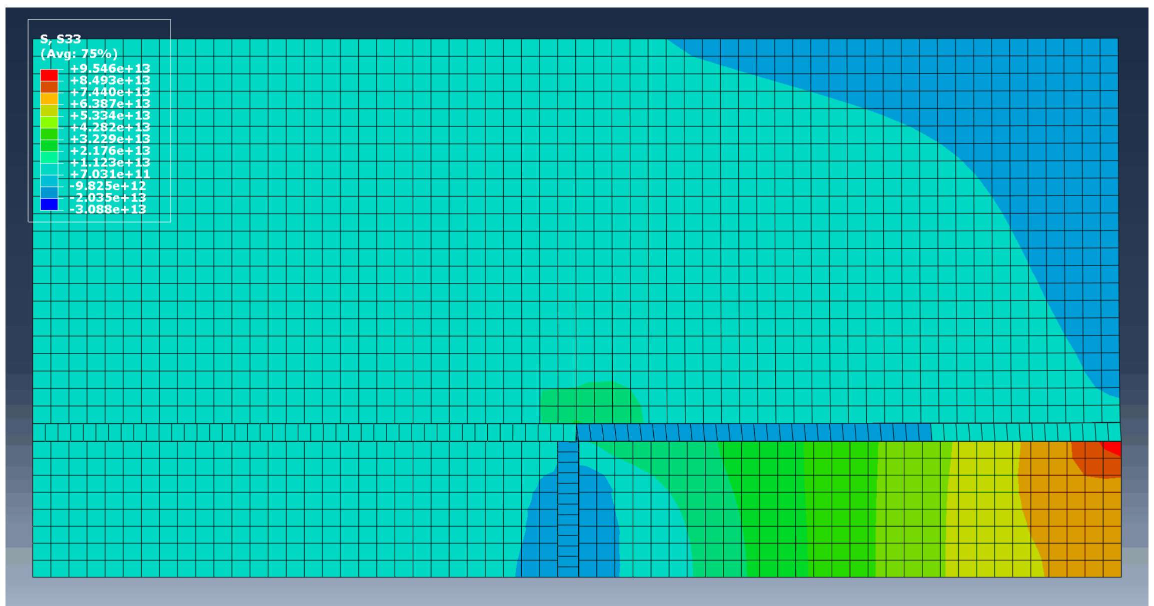


Figure 4.22: 10% FVF at peak stress

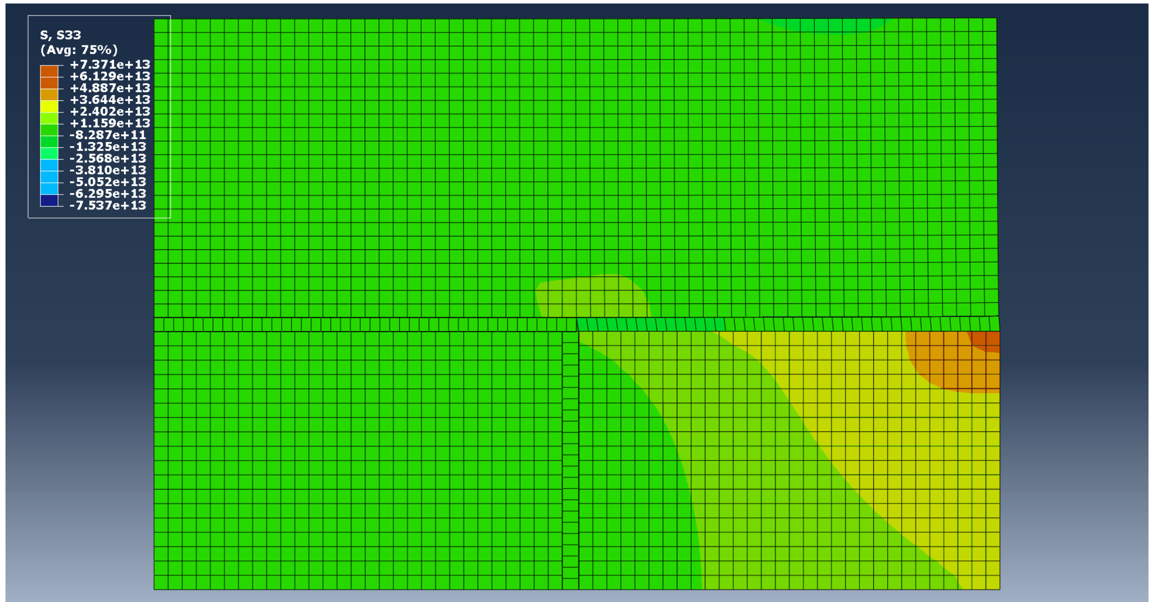


Figure 4.23: 60% FVF at peak stress

Figure 4.24: Embedded fiber contour plots for s33 at peak stress

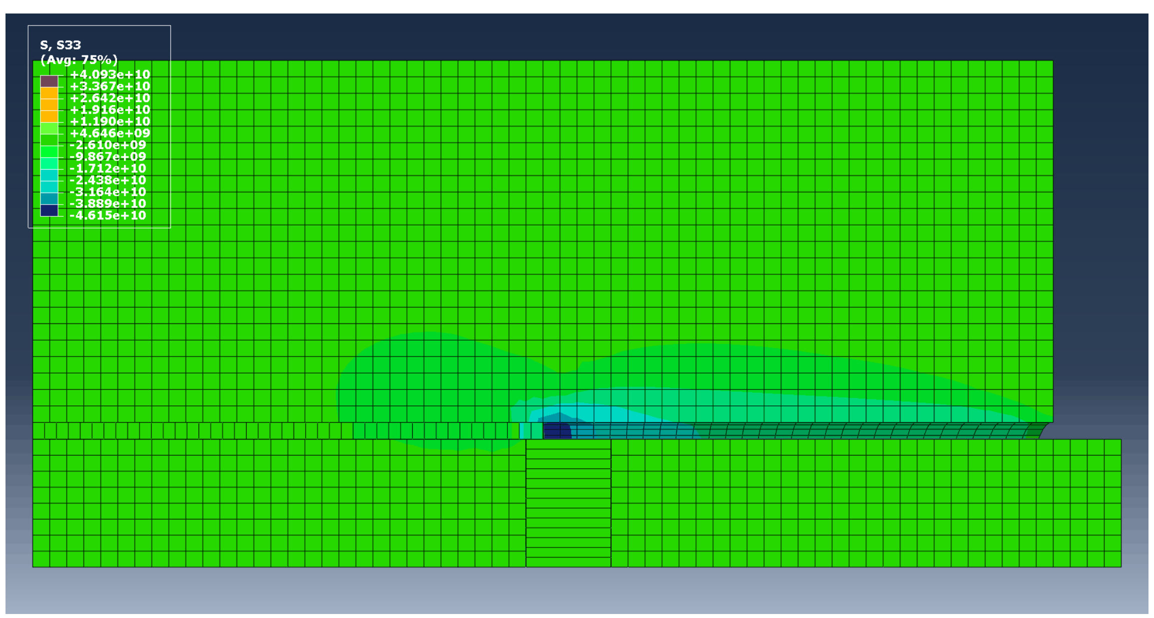


Figure 4.25: 10% FVF at failure

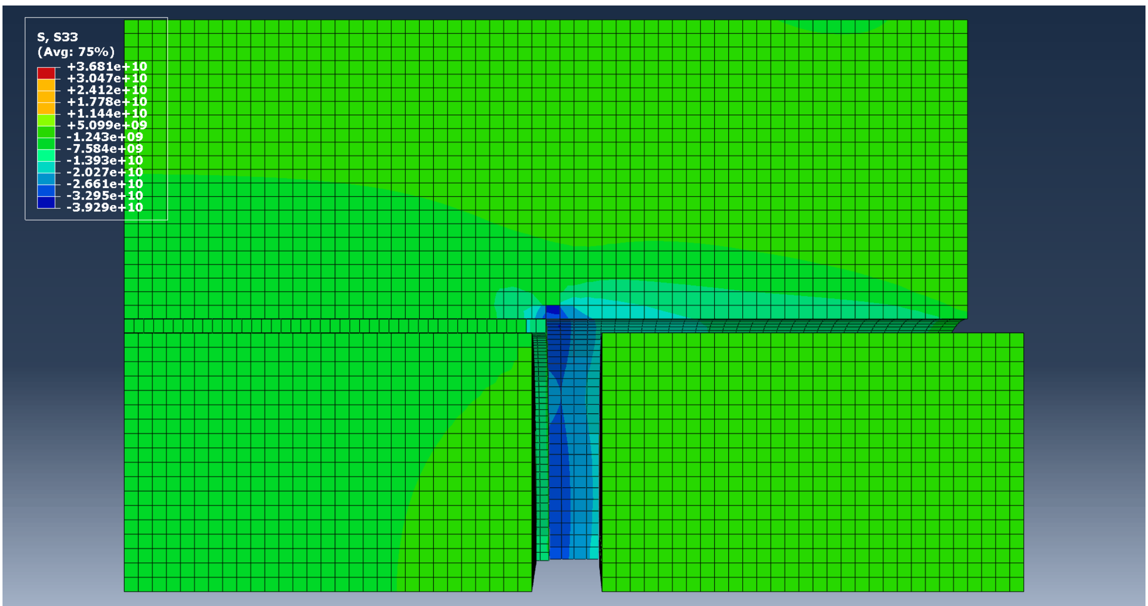


Figure 4.26: 60% FVF at failure

Figure 4.27: Embedded fiber contour plots for s33 at failure

Chapter 5

Conclusions

A fiber pull-out test was simulated using the Abaqus finite element analysis software. The test was focused on how different fiber volume fractions (FVF) influenced the pullout behavior. It was shown that on low FVF numbers (10%) the influence of the second fiber was minimal. However, with high FVF (60%) the results show that the second fiber has a substantial effect. This suggests that for composite materials with high FVF, pull-out test behavior is changed and needs to be taken into account. This could suggest that the results of interface properties on the pullout behavior could be inaccurate for these materials.

It was also shown that the addition of a second cohesive layer in the double fiber case does play a significant role in the pullout response. The addition of the cohesive layer seems to allow for more load-carrying ability. Perhaps the addition of the second layer causes the model to behave slightly ductile although further tests to understand this phenomenon would need to be performed.

Cohesive material properties were varied to understand how each property influenced the pullout behavior. It was shown that the maximum displacement influenced the model as expected with no real influence by different FVF. However, the max

traction parameter did seem to influence the behavior between FVF although not drastically. The thickness didn't seem to have an effect on the model. Thus, the cohesive properties although having an influence on the behavior of the model, are not influenced by different FVFs. This means that further tests can keep the cohesive properties constant and focus on changes in geometry.

5.1 Limitations of the Study

Both the fiber and matrix were modeled as elastic materials. Thus no plastic deformation was possible. The symmetric boundary conditions also assumed an array of fibers being pulled rather than a single fiber. A non-symmetric case would provide useful insight into a true single-fiber pullout behavior with neighboring fibers.

5.2 Implications for Future Research

Future work could aim to better characterize why the second cohesive layer helps strengthen the fiber's bond in the matrix. It would also be useful to understand how symmetry in the model influences the pullout behavior. Further look into the embedded fiber case including a skew-symmetric and varying cohesive material properties would be of interest to better understand how offset fibers affect the behavior. Material testing could also be explored to confirm numerical results. A better comparison between material test data and modeling data could show the limitation of a modeling approach, something critical in understanding a multi-scale approach.

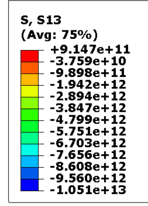
Appendices

A. S13 Contour Plots 49

B. Parametric Python Code 54

Appendix A

S13 Contour Plots



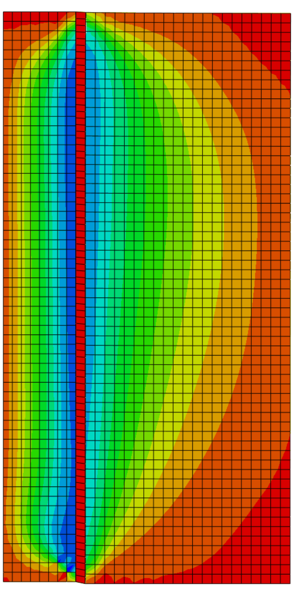
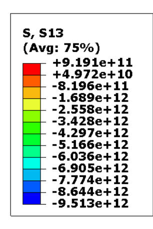


Figure A.1: 10% FVF at peak stress



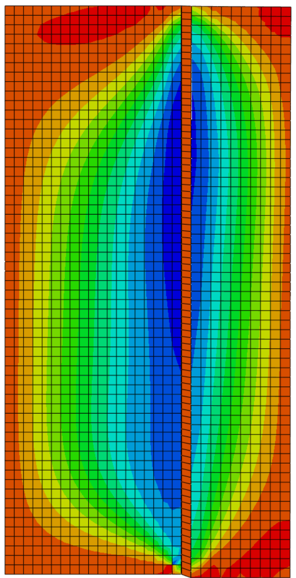
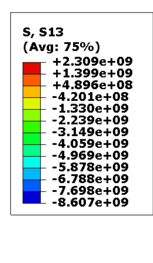


Figure A.2: 60% FVF at peak stress

Figure A.3: Single fiber contour plots for s13 at peak stress



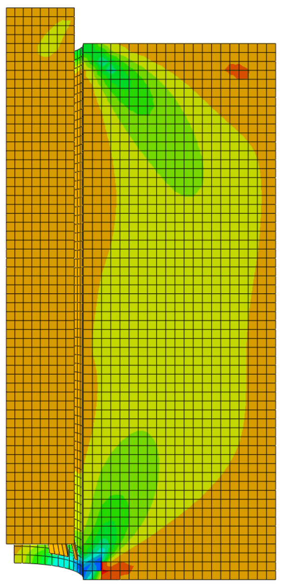
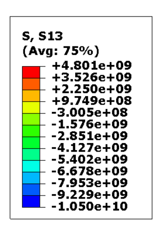


Figure A.4: 10% FVF at failure



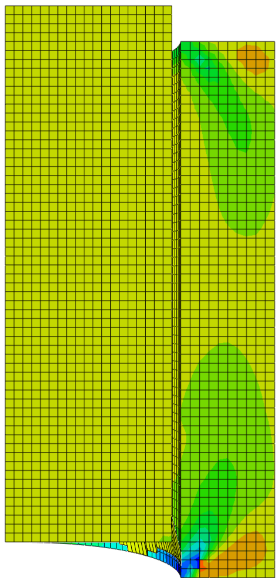
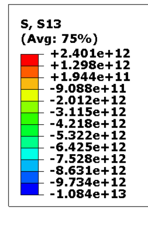


Figure A.5: 60% FVF at failure

Figure A.6: Single fiber contour plots for s13 at failure



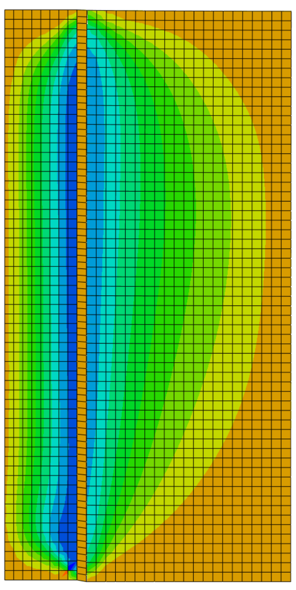
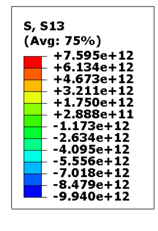


Figure A.7: 10% FVF at peak stress



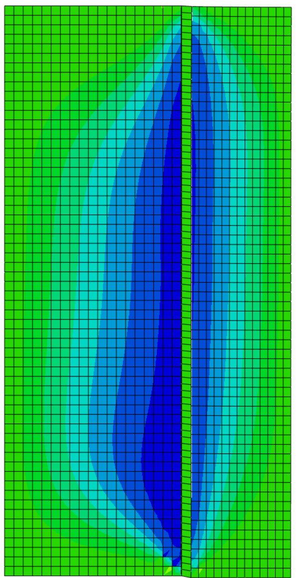


Figure A.8: 60% FVF at peak stress

Figure A.9: Double fiber contour plots for s13 at peak stress

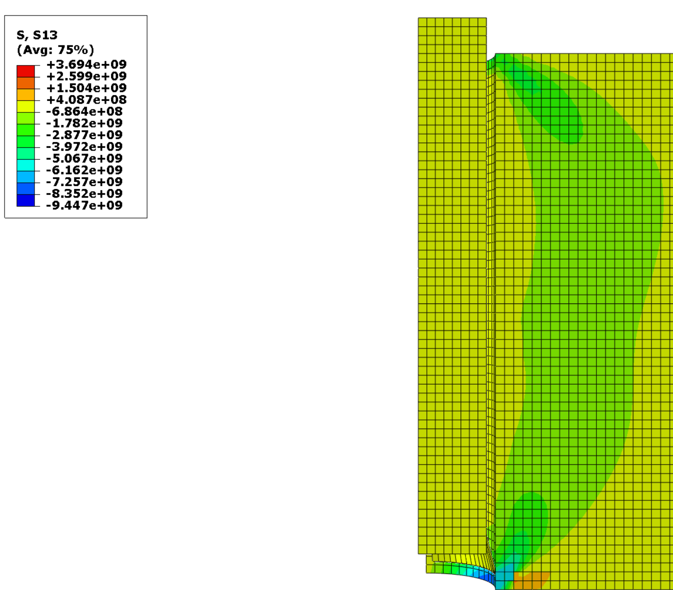


Figure A.10: 10% FVF at failure

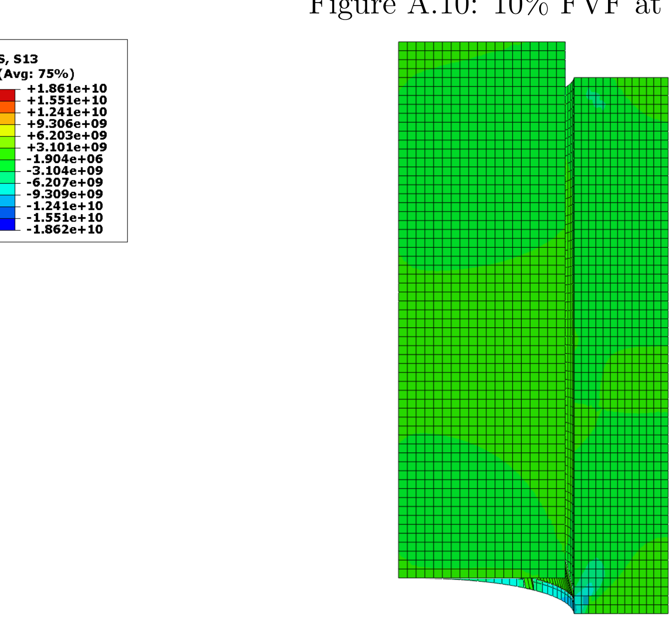


Figure A.11: 60% FVF at failure

Figure A.12: Double fiber contour plots for s13 at failure

Appendix B

Parametric Python Code

B.1 Parametric code for varying FVF

```
from math import sqrt, pi
from my_module import *
import sys
sys.path.append('path_to_files')


def compute_radius_from_fvf(fvf, edge_length):
    return sqrt(2 * fvf * edge_length**2 / pi)


def main():
    # Constants and Parameters
    EDGE_LENGTH = 15
    COH_THICKNESS = 0.5
    MODEL_LENGTH = 30
```

```
BASE_MODEL_NAME = 'Model-'
JOB_BASE_NAME = 'Job-'

# Varying the FVF
fvfs = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6]
# fvfs = [0.1]

for index, fvf in enumerate(fvfs, 1):
    # Compute the fiber radius based on the current FVF
    FIBER_RADIUS = compute_radius_from_fvf(fvf,
       EDGE_LENGTH)

    # Update the model name based on the current FVF index
    MODEL_NAME = BASE_MODEL_NAME + str(index)

    # Model Creation
    create_new_model(MODEL_NAME)
    create_model_geometry(MODEL_NAME, FIBER_RADIUS,
       COH_THICKNESS,
                              EDGE_LENGTH, MODEL_LENGTH)
    create_material_properties(MODEL_NAME)
    assign_sections(MODEL_NAME)
    define_steps(MODEL_NAME)
    create_instances_and_apply_transformations(MODEL_NAME,
                                                 EDGE_LENGTH
                                                     ,
                                                     FIBER_RADIUS
                                                     ,
                                                     COH_THICKNESS
                                                     )
```

```
        define_interactions(MODEL_NAME)
        combine_face_sets(MODEL_NAME)
        define_boundary_conditions(MODEL_NAME)
        mesh_creation(MODEL_NAME, FIBER_RADIUS, COH_THICKNESS,
                      coh_seed_fine=0.35, coh_seed_course=
                         COH_THICKNESS, fiber_seed=0.5)

        # Submitting a job with an updated name
        JOB_NAME = JOB_BASE_NAME + str(index)
        job_submission(MODEL_NAME, JOB_NAME)


def create_model_geometry(model_name, fiber_radius,
   coh_thickness, edge_length, model_length):
    # Create Fiber
    create_fiber_model(fiber_radius, model_length, model_name)

    # Create Coh
    create_coh_model(fiber_radius, coh_thickness, model_length
       , model_name)

    # Create Matrix
    create_matrix_model(fiber_radius, coh_thickness,
                        edge_length, model_length, model_name)

    create_planes('coh', model_name=model_name)
    create_planes('matrix', model_name=model_name)
    create_planes('fiber', model_name=model_name)

    create_datum('coh', model_name=model_name)
```

```
    create_datum('matrix', model_name=model_name)
    create_datum('fiber', model_name=model_name)

    # Surfaces for interface and each model component
    create_model_surfaces(model_name, fiber_radius,
       coh_thickness,
                          edge_length, model_length)


def create_fiber_model(fiber_radius, model_length, model_name)
   :
    x1 = fiber_radius
    y1 = x1
    create_fiber(x1, y1, model_length, model_name=model_name)


def create_coh_model(fiber_radius, coh_thickness, model_length
   , model_name):
    x1 = fiber_radius
    y1 = x1
    x2 = x1 + coh_thickness
    y2 = x2
    create_coh(x1, y1, x2, y2, coh_thickness,
               model_length, model_name=model_name)


def create_matrix_model(fiber_radius, coh_thickness,
   edge_length, model_length, model_name):
    x2 = fiber_radius + coh_thickness
    y2 = x2
```

```
    x3 = edge_length
    y3 = edge_length
    create_matrix(x2, y2, x3, y3, model_length, model_name=
       model_name)


def create_model_surfaces(model_name, fiber_radius,
   coh_thickness, edge_length, model_length):
    # Surfaces for coh and fiber interface
    x1 = fiber_radius
    y1 = x1
    surface_x1, surface_y1 = diagonal_scale(x1, y1)

    create_surface_at_coordinate(surface_x1, surface_y1,
       model_length/2,
                                 model_name, 'coh', '
                                    face_set_coh_fiber', '
                                    coh_fiber_face')

    create_surface_at_coordinate(surface_x1, surface_y1,
       model_length/2, model_name,
                                 'fiber', 'face_set_fiber_coh'
                                    , 'fiber_coh_face')

    # Surfaces for fiber
    fiber_x, fiber_y = x1/2, y1/2
    create_surface_at_coordinate(fiber_x, 0, model_length/2,
       model_name,
                                 'fiber', 'face_set_xz_fiber',
                                    'fiber_xz_face')
```

```
create_surface_at_coordinate(0, fiber_y, model_length/2,
   model_name,
                              'fiber', 'face_set_yz_fiber',
                                  'fiber_yz_face')
back_x, back_y = diagonal_scale(fiber_x, fiber_y)
create_surface_at_coordinate(back_x, back_y, 0, model_name
   , 'fiber',
                              'face_set_xy_back_fiber', '
                                fiber_xy_back_face')
create_surface_at_coordinate(back_x, back_y, model_length,
    model_name, 'fiber',
                              'face_set_xy_front_fiber', '
                                fiber_xy_front_face')

# Surfaces for coh
coh_x, coh_y = x1 + coh_thickness/2, y1 + coh_thickness/2
create_surface_at_coordinate(coh_x, 0, model_length/2,
   model_name,
                              'coh', 'face_set_xz_coh', '
                                coh_xz_face')
create_surface_at_coordinate(0, coh_y, model_length/2,
   model_name,
                              'coh', 'face_set_yz_coh', '
                                coh_yz_face')
back_coh_x, back_coh_y = diagonal_scale(coh_x, coh_y)
create_surface_at_coordinate(back_coh_x, back_coh_y, 0,
   model_name, 'coh',
                              'face_set_xy_back_coh', '
                                coh_xy_back_face')
create_surface_at_coordinate(back_coh_x, back_coh_y,
```

```
    model_length, model_name,
                                            'coh', 'face_set_xy_front_coh
                                              ', 'coh_xy_front_face')

# Surfaces for matrix
matrix_x, matrix_y = edge_length-0.01, edge_length-0.01
create_surfaces(matrix_x, 0, model_length/2, model_name, '
    matrix',
                  'face_set_xz_bottom_matrix', '
                    matrix_xz_bottom_face')
create_surfaces(0, matrix_y, model_length/2, model_name, '
    matrix',
                  'face_set_yz_left_matrix', '
                    matrix_yz_left_face')
create_surfaces(edge_length-matrix_x, edge_length,
    model_length/2, model_name,
                  'matrix', 'face_set_xz_top_matrix', '
                    matrix_xz_top_face')
create_surfaces(edge_length, edge_length-matrix_y,
    model_length/2, model_name,
                  'matrix', 'face_set_yz_right_matrix', '
                    matrix_yz_right_face')
create_surfaces(matrix_x, edge_length-matrix_y, 0,
    model_name, 'matrix',
                  'face_set_xy_back_matrix', '
                    matrix_xy_back_face')
create_surfaces(matrix_x, edge_length-matrix_y,
    model_length, model_name,
                  'matrix', 'face_set_xy_front_matrix', '
                    matrix_xy_front_face')
```

```
    # Surfaces for coh and matrix interface
    x2 = x1 + coh_thickness
    y2 = x2
    surface_x2, surface_y2 = diagonal_scale(x2, y2)
    create_surfaces(surface_x2, surface_y2, model_length/2,
       model_name,
                    'coh', 'face_set_coh_matrix', '
                       coh_matrix_face')
    create_surfaces(surface_x2, surface_y2, model_length/2,
       model_name,
                    'matrix', 'face_set_bottom_matrix_coh', '
                       bottom_matrix_coh_face')

    x3 = edge_length
    y3 = edge_length
    diagonal_length = sqrt(x3**2 + y3**2)
    surface_x3, surface_y3 = diagonal_scale(
        diagonal_length - x2, diagonal_length - y2)
    create_surfaces(surface_x3, surface_y3, model_length/2,
       model_name,
                    'matrix', 'face_set_top_matrix_coh', '
                       top_matrix_coh_face')


def create_material_properties(model_name):
    E_CARBON = 2.27e+17
    POISSON_RATIO_CARBON = 0.4
    material_carbon(E_CARBON, POISSON_RATIO_CARBON, model_name
       =model_name)
```

```
    NS_NORMAL_ONLY = 10000000000000
    E_COH = 1E+015
    G1_ESS = 1E+015
    G2_ETT = 1E+015
    MAX_DISPLACEMENT = 1
    VISCOSITY = 0.01
    material_coh(NS_NORMAL_ONLY, NS_NORMAL_ONLY,
       NS_NORMAL_ONLY,
                 MAX_DISPLACEMENT, VISCOSITY, E_COH, G1_ESS,
                    G2_ETT, model_name=model_name)

    E_MATRIX = 3e+15
    POISSON_RATIO_MATRIX = 0.3
    material_matrix(E_MATRIX, POISSON_RATIO_MATRIX, model_name
       =model_name)


def assign_sections(model_name):
    create_and_assign_section(model_name, 'coh', 'coh', 'coh')
    create_and_assign_section(model_name, 'matrix', 'matrix',
       'matrix')
    create_and_assign_section(model_name, 'fiber', '
       carbon_fiber', 'fiber')


def define_steps(model_name):
    create_step(initial_increment=0.01, model_name=model_name)
    variables = ('S', 'PE', 'PEEQ', 'PEMAG', 'LE', 'U',
                 'RF', 'CF', 'CSTRESS', 'CDISP', 'STATUS')
```

```
    set_field_output_requests(model_name, 'F-Output-1',
        variables)


def create_instances_and_apply_transformations(model_name,
    edge_length, fiber_radius, coh_thickness):
    # Creating instances
    instance_matrix(model_name=model_name)
    instance_coh(model_name=model_name)
    instance_fiber(model_name=model_name)
    instance_coh(model_name=model_name, instance_name='coh-2')
    instance_fiber(model_name=model_name, instance_name='fiber
        -2')

    # Set Positions for Translation and Rotation
    x1 = fiber_radius
    x2 = x1 + coh_thickness
    y1 = 0.0
    y2 = edge_length
    z1 = 0.0
    z2 = 0.0

    # Instance Translations and Rotations for Coh and Fiber
    translate_instance('coh-1', 0.0, y1, z1, model_name=
        model_name)
    translate_instance('fiber-1', 0.0, y1, z1, model_name=
        model_name)
    translate_instance('coh-2', edge_length - 2*x2,
                       y2, z2, model_name=model_name)
    rotate_instance('coh-2', -180.0, 0, 0, -1, edge_length -
```

```
                        x2, y2, z2, model_name=model_name)
    translate_instance('fiber-2', edge_length - 2*x1,
                           y2, z2, model_name=model_name)
    rotate_instance('fiber-2', -180.0, 0, 0, -1, edge_length -
                        x1, y2, z2, model_name=model_name)

    # Function calls for the given surfaces
    create_tie('fiber-1', 'fiber_coh_face', 'coh-1',
                'coh_fiber_face', model_name=model_name)
    create_tie('matrix-1', 'bottom_matrix_coh_face',
                'coh-1', 'coh_matrix_face', model_name=
                   model_name)
    create_tie('fiber-2', 'fiber_coh_face', 'coh-2',
                'coh_fiber_face', model_name=model_name)
    create_tie('matrix-1', 'top_matrix_coh_face', 'coh-2',
                'coh_matrix_face', model_name=model_name)


def define_interactions(model_name):
    create_tie('fiber-1', 'fiber_coh_face', 'coh-1',
                'coh_fiber_face', model_name=model_name)
    create_tie('matrix-1', 'bottom_matrix_coh_face',
                'coh-1', 'coh_matrix_face', model_name=
                   model_name)
    create_tie('fiber-2', 'fiber_coh_face', 'coh-2',
                'coh_fiber_face', model_name=model_name)
    create_tie('matrix-1', 'top_matrix_coh_face', 'coh-2',
                'coh_matrix_face', model_name=model_name)
```

```
def combine_face_sets(model_name):
    # Right Face Set
    combine_sets(model_name, 'right_combined_face',
                 ('fiber-2', 'face_set_yz_fiber'),
                 ('matrix-1', 'face_set_yz_right_matrix'))

    # Top Face Set
    combine_sets(model_name, 'top_combined_face',
                 ('fiber-2', 'face_set_xz_fiber'),
                 ('matrix-1', 'face_set_xz_top_matrix'))

    # Left Face Set
    combine_sets(model_name, 'left_combined_face',
                 ('fiber-1', 'face_set_yz_fiber'),
                 ('matrix-1', 'face_set_yz_left_matrix'))

    # Bottom Face Set
    combine_sets(model_name, 'bottom_combined_face',
                 ('fiber-1', 'face_set_xz_fiber'),
                 ('matrix-1', 'face_set_xz_bottom_matrix'))

    # Back Face Set
    combine_sets(model_name, 'back_combined_face',
                 ('fiber-2', 'face_set_xy_back_fiber'),
                 ('coh-2', 'face_set_xy_back_coh'),
                 ('matrix-1', 'face_set_xy_back_matrix'))


def define_boundary_conditions(model_name):
    create_ysymm_bc(model_name, 'Step-1', 'Bottom_face',
```

```
                    'bottom_combined_face')

    create_xsymm_bc(model_name, 'Step-1', 'Left_face', '
       left_combined_face')

    create_no_rotate_bc(model_name, 'Step-1', 'top_face', '
       top_combined_face')
    create_no_rotate_bc(model_name, 'Step-1',
                        'right_face', 'right_combined_face')

    # Function call:
    zsymm_bc(model_name, 'Step-1', 'back_ZsymmBC', '
       back_combined_face')

    create_no_rotate_bc(model_name, 'Step-1', 'displacement',
                        'fiber-1.face_set_xy_front_fiber')

    # Function call:
    apply_displacement_bc(model_name, 'Step-1', 'displacement'
       ,
                          'fiber-1', 'face_set_xy_front_fiber'
                             , u3=2.0)


def mesh_creation(model_name, fiber_radius, coh_thickness,
   coh_seed_fine, coh_seed_course, fiber_seed):
    matrix_seed = fiber_seed

    create_mesh(model_name, 'fiber', 'fiber', fiber_seed)
    create_mesh(model_name, 'matrix', 'matrix', matrix_seed)
```

```
    create_mesh(model_name, 'coh', 'coh', coh_seed_fine)

    # surface_names = ('coh_matrix_face', 'coh_fiber_face')
    # create_coh_mesh(model_name, "coh", coh_seed_fine,
       surface_names, ["coh_xz_face", "coh_yz_face"],
       coh_seed_course)


def job_submission(model_name, name_of_run):
    create_job(model_name, job_name=name_of_run)
    # submit_job(model_name, name_of_run)


if __name__ == '__main__':
    main()
```

B.2 Module Code

```
from part import *
from material import *
from section import *
from assembly import *
from step import *
from interaction import *
from load import *
from optimization import *
from job import *
from sketch import *
```

```
from visualization import *
from connectorBehavior import *
from math import *
import mesh
from abaqusConstants import *


def create_new_model(model_name):
    # Check if model already exists
    if model_name in mdb.models.keys():
        return

    # Create a new model
    mdb.Model(modelType=STANDARD_EXPLICIT, name=model_name)


def create_fiber(x1, y1, model_length, model_name='Model-1',
   part_name='fiber'):
    mdb.models[model_name].ConstrainedSketch(
        name='__profile__', sheetSize=200.0)
    mdb.models[model_name].sketches['__profile__'].
       ArcByCenterEnds(center=(
        0.0, 0.0), direction=COUNTERCLOCKWISE, point1=(x1,
           0.0), point2=(0.0, y1))
    mdb.models[model_name].sketches['__profile__'].Line(point1
       =(0.0, 0.0), point2=(
        x1, 0.0))
    mdb.models[model_name].sketches['__profile__'].
       HorizontalConstraint(
        addUndoState=False, entity=mdb.models[model_name].
```

```
            sketches['__profile__'].geometry[3])
    mdb.models[model_name].sketches['__profile__'].Line(point1
       =(0.0, y1), point2=(
        0.0, 0.0))
    mdb.models[model_name].sketches['__profile__'].
       VerticalConstraint(
        addUndoState=False, entity=mdb.models[model_name].
            sketches['__profile__'].geometry[4])
    mdb.models[model_name].sketches['__profile__'].
       PerpendicularConstraint(
        addUndoState=False, entity1=mdb.models[model_name].
            sketches['__profile__'].geometry[2], entity2=mdb.
            models[model_name].sketches['__profile__'].geometry
            [4])
    mdb.models[model_name].Part(dimensionality=THREE_D,
                                name=part_name, type=
                                   DEFORMABLE_BODY)
    mdb.models[model_name].parts[part_name].BaseSolidExtrude(
        depth=model_length, sketch=mdb.models[model_name].
            sketches['__profile__'])
    del mdb.models[model_name].sketches['__profile__']


def create_coh(x1, y1, x2, y2, coh_thickness, model_length,
   model_name='Model-1', part_name='coh'):
    mdb.models[model_name].ConstrainedSketch(
        name='__profile__', sheetSize=200.0)
    mdb.models[model_name].sketches['__profile__'].
       ArcByCenterEnds(center=(
        0.0, 0.0), direction=COUNTERCLOCKWISE, point1=(x1,
```

```
        0.0), point2=(0.0, y1))
mdb.models[model_name].sketches['__profile__'].Line(point1
    =(x1, 0.0), point2=(
     x2, 0.0))
mdb.models[model_name].sketches['__profile__'].
    HorizontalConstraint(
     addUndoState=False, entity=mdb.models[model_name].
        sketches['__profile__'].geometry[3])
mdb.models[model_name].sketches['__profile__'].
    PerpendicularConstraint(
     addUndoState=False, entity1=mdb.models[model_name].
        sketches['__profile__'].geometry[2], entity2=mdb.
        models[model_name].sketches['__profile__'].geometry
        [3])
mdb.models[model_name].sketches['__profile__'].
    ArcByCenterEnds(center=(
     0.0, 0.0), direction=COUNTERCLOCKWISE, point1=(x2,
        0.0), point2=(0.0, y2))
mdb.models[model_name].sketches['__profile__'].Line(point1
    =(0.0, y1), point2=(
     0.0, y2))
mdb.models[model_name].sketches['__profile__'].
    VerticalConstraint(
     addUndoState=False, entity=mdb.models[model_name].
        sketches['__profile__'].geometry[5])
mdb.models[model_name].sketches['__profile__'].
    PerpendicularConstraint(
     addUndoState=False, entity1=mdb.models[model_name].
        sketches['__profile__'].geometry[2], entity2=mdb.
        models[model_name].sketches['__profile__'].geometry
```

```
        [5])
    mdb.models[model_name].Part(dimensionality=THREE_D,
                                name=part_name, type=
                                   DEFORMABLE_BODY)
    mdb.models[model_name].parts[part_name].BaseSolidExtrude(
        depth=model_length, sketch=mdb.models[model_name].
           sketches['__profile__'])
    del mdb.models[model_name].sketches['__profile__']


def create_matrix(x2, y2, x3, y3, model_length, model_name='
   Model-1', part_name='matrix'):
    mdb.models[model_name].ConstrainedSketch(
        name='__profile__', sheetSize=200.0)
    mdb.models[model_name].sketches['__profile__'].
       ArcByCenterEnds(center=(
        0.0, 0.0), direction=COUNTERCLOCKWISE, point1=(x2,
           0.0), point2=(0.0, y2))
    mdb.models[model_name].sketches['__profile__'].Line(point1
       =(x2, 0.0), point2=(
        x3, 0.0))
    mdb.models[model_name].sketches['__profile__'].
       HorizontalConstraint(
        addUndoState=False, entity=mdb.models[model_name].
           sketches['__profile__'].geometry[3])
    mdb.models[model_name].sketches['__profile__'].
       PerpendicularConstraint(
        addUndoState=False, entity1=mdb.models[model_name].
           sketches['__profile__'].geometry[2], entity2=mdb.
           models[model_name].sketches['__profile__'].geometry
```

```
        [3])
mdb.models[model_name].sketches['__profile__'].Line(point1
    =(0.0, y2), point2=(
     0.0, y3))
mdb.models[model_name].sketches['__profile__'].
    VerticalConstraint(
     addUndoState=False, entity=mdb.models[model_name].
        sketches['__profile__'].geometry[4])
mdb.models[model_name].sketches['__profile__'].
    PerpendicularConstraint(
     addUndoState=False, entity1=mdb.models[model_name].
        sketches['__profile__'].geometry[2], entity2=mdb.
        models[model_name].sketches['__profile__'].geometry
        [4])
mdb.models[model_name].sketches['__profile__'].
    ConstructionLine(point1=(x3,
```

```
mdb.models[model_name].sketches['__profile__'].
    CoincidentConstraint(
     addUndoState=False, entity1=mdb.models[model_name].
        sketches['__profile__'].vertices[3], entity2=mdb.
```

```
            models[model_name].sketches['__profile__'].geometry
            [5])
    mdb.models[model_name].sketches['__profile__'].
        CoincidentConstraint(
         addUndoState=False, entity1=mdb.models[model_name].
            sketches['__profile__'].vertices[4], entity2=mdb.
            models[model_name].sketches['__profile__'].geometry
            [5])
    mdb.models[model_name].sketches['__profile__'].copyMirror(
        mirrorLine=mdb.models[model_name].sketches['__profile__
        '].geometry[5], objectList=(
         mdb.models[model_name].sketches['__profile__'].
            geometry[2],
         mdb.models[model_name].sketches['__profile__'].
            geometry[3],
         mdb.models[model_name].sketches['__profile__'].
            geometry[4],
         mdb.models[model_name].sketches['__profile__'].
            geometry[5]))
    mdb.models[model_name].Part(dimensionality=THREE_D,
                                name=part_name, type=
                                   DEFORMABLE_BODY)
    mdb.models[model_name].parts[part_name].BaseSolidExtrude(
         depth=model_length, sketch=mdb.models[model_name].
            sketches['__profile__'])
    del mdb.models[model_name].sketches['__profile__']


def create_matrix_with_corner(x2, y2, x3, y3, y4, model_length
   , model_name='Model-1', part_name='matrix'):
```

```
mdb.models[model_name].ConstrainedSketch(
    name='__profile__', sheetSize=200.0)
mdb.models[model_name].sketches['__profile__'].
   ArcByCenterEnds(
    center=(0.0, 0.0), direction=COUNTERCLOCKWISE, point1
       =(x2, 0.0), point2=(0.0, y2))

# Horizontal line from the end of the arc to x3
mdb.models[model_name].sketches['__profile__'].Line(
    point1=(x2, 0.0), point2=(x3, 0.0))
mdb.models[model_name].sketches['__profile__'].
   HorizontalConstraint(
    addUndoState=False, entity=mdb.models[model_name].
       sketches['__profile__'].geometry[3])
mdb.models[model_name].sketches['__profile__'].
   PerpendicularConstraint(
    addUndoState=False, entity1=mdb.models[model_name].
       sketches['__profile__'].geometry[2], entity2=mdb.
       models[model_name].sketches['__profile__'].geometry
       [3])

# Vertical line from x3 to y3
mdb.models[model_name].sketches['__profile__'].Line(
    point1=(x3, 0.0), point2=(x3, y3))
mdb.models[model_name].sketches['__profile__'].
   VerticalConstraint(
    addUndoState=False, entity=mdb.models[model_name].
       sketches['__profile__'].geometry[4])
mdb.models[model_name].sketches['__profile__'].
   PerpendicularConstraint(
```

```
        addUndoState=False, entity1=mdb.models[model_name].
            sketches['__profile__'].geometry[3], entity2=mdb.
            models[model_name].sketches['__profile__'].geometry
            [4])

# Horizontal line from y3 to y4
mdb.models[model_name].sketches['__profile__'].Line(
        point1=(x3, y3), point2=(0.0, y3))
mdb.models[model_name].sketches['__profile__'].
    HorizontalConstraint(
        addUndoState=False, entity=mdb.models[model_name].
            sketches['__profile__'].geometry[5])
mdb.models[model_name].sketches['__profile__'].
    PerpendicularConstraint(
        addUndoState=False, entity1=mdb.models[model_name].
            sketches['__profile__'].geometry[4], entity2=mdb.
            models[model_name].sketches['__profile__'].geometry
            [5])

# Vertical line to complete the shape
mdb.models[model_name].sketches['__profile__'].Line(
        point1=(0.0, y3), point2=(0.0, y2))
mdb.models[model_name].sketches['__profile__'].
    VerticalConstraint(
        addUndoState=False, entity=mdb.models[model_name].
            sketches['__profile__'].geometry[6])
mdb.models[model_name].sketches['__profile__'].
    PerpendicularConstraint(
        addUndoState=False, entity1=mdb.models[model_name].
            sketches['__profile__'].geometry[5], entity2=mdb.
```

```
        models[model_name].sketches['__profile__'].geometry
        [6])

    # Create the part
    mdb.models[model_name].Part(
        dimensionality=THREE_D, name=part_name, type=
            DEFORMABLE_BODY)
    mdb.models[model_name].parts[part_name].BaseSolidExtrude(
        depth=model_length, sketch=mdb.models[model_name].
            sketches['__profile__'])

    # Delete the sketch
    del mdb.models[model_name].sketches['__profile__']


def create_planes(part_name, model_name='Model-1'):
    mdb.models[model_name].parts[part_name].
       DatumPlaneByPrincipalPlane(offset=0.0,
                                                                pr




    mdb.models[model_name].parts[part_name].
       DatumPlaneByPrincipalPlane(offset=0.0,
                                                                pr
```

```
    mdb.models[model_name].parts[part_name].
       DatumPlaneByPrincipalPlane(offset=0.0,
                                                                      pr

def create_datum(part_name, model_name='Model-1'):
    mdb.models[model_name].parts[part_name].
       DatumCsysByThreePoints(coordSysType=CARTESIAN, line1=(
        1.0, 0.0, 0.0), line2=(0.0, 1.0, 0.0), name='Datum
           csys-1', origin=(0.0, 0.0, 0.0))


def create_surface_at_coordinate(x, y, z, model, part,
   set_name, surface_name):
    """
    Finds a face at specified (x, y, z) coordinate in a given
       model and part in Abaqus,
    then creates a surface using that face.

    Parameters:
    - x, y, z: coordinates to find the face
    - model: name of the model
    - part: name of the part
    - surface_name: name for the created surface
    """
```

```
    # Access the specified part in the model
    p = mdb.models[model].parts[part]
    f = p.faces

    myFace = f.findAt((x, y, z),)

    face_tuple = (f[myFace.index:myFace.index+1], )

    # Create a set using the found face
    p.Set(faces=face_tuple, name=set_name)

    # Create a surface using the face from the set
    p.Surface(name=surface_name, side1Faces=face_tuple)


def create_surfaces(x, y, z, model, material, set_name,
   face_name):
    create_surface_at_coordinate(x, y, z, model, material,
        set_name, face_name)



def diagonal_scale(value1, value2):
    return 0.707106781187 * value1, 0.707106781187 * value2



def material_carbon(E, Poisson_ratio, model_name='Model-1'):
    mdb.models[model_name].Material(name='carbon_fiber')
    mdb.models[model_name].materials['carbon_fiber'].Elastic(
        table=((E, Poisson_ratio),))
```

```
def material_coh(NS_normal_only, NS_first_direction,
    NS_second_direction, max_displacement, viscosity, E_Enn,
    G1_Ess, G2_Ett, model_name='Model-1'):
     mdb.models[model_name].Material(name='coh')
     mdb.models[model_name].materials['coh'].
        QuadsDamageInitiation(table=((NS_normal_only,
```

```
    mdb.models[model_name].materials['coh'].
        quadsDamageInitiation.DamageEvolution(
         table=((max_displacement, ), ), type=DISPLACEMENT)
    mdb.models[model_name].materials['coh'].
        quadsDamageInitiation.DamageStabilizationCohesive(
         cohesiveCoeff=viscosity)
    mdb.models[model_name].materials['coh'].Elastic(
         table=((E_Enn, G1_Ess, G2_Ett),), type=TRACTION)


def material_matrix(E, Poisson_ratio, model_name='Model-1'):
    mdb.models[model_name].Material(name='matrix')
    mdb.models[model_name].materials['matrix'].Elastic(
```

```
        table=((E, Poisson_ratio), ))


def create_and_assign_section(model_name, part_name,
   material_name, section_name):
    """
    Create a section in the model and assign it to the
       specified part.

    Parameters:
    - model_name: Name of the model.
    - part_name: Name of the part to assign the section.
    - material_name: Name of the material used in the section.
    - section_name: Name of the section.
    """
    p = mdb.models[model_name].parts[part_name]

    if material_name == 'coh':
        mdb.models[model_name].CohesiveSection(material=
           material_name, name=section_name,
                                               outOfPlaneThickness
                                                  =None,
                                                  response=
                                                  TRACTION_SEPARATION
                                                  )
    else:
        mdb.models[model_name].HomogeneousSolidSection(
            material=material_name, name=section_name,
               thickness=None)
```

```
    cells = p.cells[:]
    pickedRegions = (cells, )
    p.Set(cells=cells, name='Set-{}'.format(part_name))
    p.SectionAssignment(offset=0.0, offsetField='', offsetType
        =MIDDLE_SURFACE,
                        region=p.sets['Set-{}'.format(
                            part_name)
                                    ], sectionName=
                                        section_name,
                        thicknessAssignment=FROM_SECTION)


def instance_matrix(instance_name='matrix-1', model_name='
    Model-1'):
    mdb.models[model_name].rootAssembly.DatumCsysByDefault(
        CARTESIAN)
    mdb.models[model_name].rootAssembly.Instance(dependent=ON,
         name=instance_name,
                                                 part=mdb.
                                                     models[
                                                     model_name
                                                     ].parts['
                                                     matrix'])


def instance_coh(instance_name='coh-1', coh1x=0.0, coh1y=0.0,
    coh1z=0.0, model_name='Model-1'):
    mdb.models[model_name].rootAssembly.Instance(
        dependent=ON, name=instance_name, part=mdb.models[
            model_name].parts['coh'])
```

```
    mdb.models[model_name].rootAssembly.instances[
       instance_name].translate(
        vector=(coh1x, coh1y, coh1z))


def instance_fiber(instance_name='fiber-1', fiber1x=0.0,
   fiber1y=0.0, fiber1z=0.0, model_name='Model-1'):
    mdb.models[model_name].rootAssembly.Instance(
        dependent=ON, name=instance_name, part=mdb.models[
           model_name].parts['fiber'])
    mdb.models[model_name].rootAssembly.instances[
       instance_name].translate(
        vector=(fiber1x, fiber1y, fiber1z))


def translate_instance(instance_name, x, y=0.0, z=0.0,
   model_name='Model-1'):
    mdb.models[model_name].rootAssembly.translate(
        instanceList=(instance_name, ), vector=(x, y, z))


def rotate_instance(instance_name, angle, axisDirectionX,
   axisDirectionY, axisDirectionZ, axisPointX, axisPointY,
   axisPointZ, model_name='Model-1'):
    mdb.models[model_name].rootAssembly.rotate(angle=angle,
       axisDirection=(axisDirectionX, axisDirectionY,
       axisDirectionZ),
                                                axisPoint=(
                                                   axisPointX,
                                                   axisPointY,
```

```
                                                                axisPointZ),

                                                                instanceList
                                                                =(
                                                                instance_name
                                                                , ))


def create_tie(master_instance, master_surface, slave_instance
    , slave_surface, model_name='Model-1'):
    tie_name = "{}.{}".format(slave_instance, slave_surface)

    mdb.models[model_name].Tie(adjust=ON,
                               master=mdb.models[model_name].
                                   rootAssembly.instances[
                                   master_instance].surfaces[
                                   master_surface],
                               name=tie_name,
                               positionToleranceMethod=
                                   COMPUTED,
                               slave=mdb.models[model_name].
                                   rootAssembly.instances[
                                   slave_instance].surfaces[
                                   slave_surface],
                               thickness=ON,
                               tieRotations=ON)


def create_step(step_name='Step-1', initial_increment=0.1,
                max_num_increments=100000, nonlinear_geometry=
```

```
                        True, previous_step_name='Initial',
                        model_name='Model-1'):
    mdb.models[model_name].StaticStep(initialInc=
        initial_increment,
                                      maxNumInc=
                                          max_num_increments,
                                      name=step_name,
                                      nlgeom=
                                          nonlinear_geometry,
                                      previous=
                                          previous_step_name)


def set_field_output_requests(model_name, request_name,
    variables):
    """
    Set values for field output requests in Abaqus.

    Args:
    - model_name (str): Name of the model in mdb.
    - request_name (str): Name of the field output request.
    - variables (tuple): Tuple containing the variables for
        field output.
    """
    # Create default Field Output Request, if it doesn't exist
    if 'F-Output-1' not in mdb.models[model_name].
        fieldOutputRequests.keys():
         mdb.models[model_name].FieldOutputRequest(name='F-
             Output-1',
                                                  variables=('
```

```
                                                                        S', 'U',
                                                                        'RF'))

    # Create default History Output Request, if it doesn't
        exist
    if 'H-Output-1' not in mdb.models[model_name].
        historyOutputRequests.keys():
         mdb.models[model_name].HistoryOutputRequest(name='H-
            Output-1',
                                                                       variables
                                                                          =('U1',
                                                                           'U2',
                                                                          'U3'))

    mdb.models[model_name].fieldOutputRequests[request_name].
        setValues(
         variables=variables)


def combine_sets(model_name, combined_set_name, *set_tuples):
    """
    Combines multiple sets into a single set in Abaqus.

    Parameters:
    - model_name: name of the model
    - combined_set_name: name of the resulting combined set
    - set_tuples: tuples containing (instance_name, set_name)
        for all sets to be combined
    """
```

```
    assembly = mdb.models[model_name].rootAssembly
    sets_to_combine = [assembly.allInstances[inst_name].sets[
        set_name]
                       for inst_name, set_name in set_tuples]

    assembly.SetByBoolean(name=combined_set_name, sets=tuple(
        sets_to_combine))


def create_ysymm_bc(model_name, step_name, bc_name, set_name):
    """
    Creates a Y-symmetric boundary condition for a specified
        set.

    Parameters:
    - model_name: name of the model
    - step_name: name of the step for which the BC is defined
    - bc_name: name of the boundary condition
    - set_name: name of the set to which the BC is applied
    """

    mdb.models[model_name].YsymmBC(createStepName=step_name,
                                   localCsys=None,
                                   name=bc_name,
                                   region=mdb.models[
                                       model_name].rootAssembly
                                       .sets[set_name])



def create_xsymm_bc(model_name, step_name, bc_name, set_name):
```

```
    """
    Creates an X-symmetric boundary condition for a specified
       set.

    Parameters:
    - model_name: name of the model
    - step_name: name of the step for which the BC is defined
    - bc_name: name of the boundary condition
    - set_name: name of the set to which the BC is applied
    """

    mdb.models[model_name].XsymmBC(createStepName=step_name,
                                   localCsys=None,
                                   name=bc_name,
                                   region=mdb.models[
                                      model_name].rootAssembly
                                      .sets[set_name])


def create_no_rotate_bc(model_name, step_name, bc_name,
   set_name, ur1_value=0.0, ur2_value=0.0, ur3_value=0.0):
    """
    Creates a displacement boundary condition with specified
       rotational displacements.

    Parameters:
    - model_name: name of the model
    - step_name: name of the step for which the BC is defined
    - bc_name: name of the boundary condition
    - set_name: name of the set to which the BC is applied
```

```
    - ur1_value, ur2_value, ur3_value: Rotational
       displacements around x, y, z respectively
    """

    mdb.models[model_name].DisplacementBC(amplitude=UNSET,
                                          createStepName=
                                             step_name,
                                          distributionType=
                                             UNIFORM,
                                          fieldName='',
                                          fixed=OFF,
                                          localCsys=None,
                                          name=bc_name,
                                          region=mdb.models[
                                             model_name].
                                             rootAssembly.sets
                                             [set_name],
                                          u1=UNSET,
                                          u2=UNSET,
                                          u3=UNSET,
                                          ur1=ur1_value,
                                          ur2=ur2_value,
                                          ur3=ur3_value)


def zsymm_bc(model_name, step_name, bc_name, set_name):
    """
    Apply Z-symmetric boundary condition to a specified set.

    Parameters:
```

```
    - model_name: Name of the model
    - step_name: Name of the step in which BC should be
       applied
    - bc_name: Name for the boundary condition
    - set_name: Name of the set to which BC should be applied
    """

    mdb.models[model_name].ZsymmBC(createStepName=step_name,
                                   localCsys=None,
                                   name=bc_name,
                                   region=mdb.models[
                                      model_name].rootAssembly
                                      .sets[set_name])


def apply_displacement_bc(model_name, step_name, bc_name,
   instance_name, set_name, u1=UNSET, u2=UNSET, u3=UNSET, ur1=
   UNSET, ur2=UNSET, ur3=UNSET):
    """
    Apply Displacement boundary condition to a specified set
       in a specified instance.

    Parameters:
    - model_name: Name of the model
    - step_name: Name of the step in which BC should be
       applied
    - bc_name: Name for the boundary condition
    - instance_name: Name of the instance where the set is
       located
    - set_name: Name of the set to which BC should be applied
```

```
    - u1, u2, u3: Displacements in the X, Y, and Z directions
       respectively
    - ur1, ur2, ur3: Rotational displacements about X, Y, and
       Z axes respectively
    """

    region = mdb.models[model_name].rootAssembly.instances[
       instance_name].sets[set_name]

    mdb.models[model_name].DisplacementBC(amplitude=UNSET,
                                          createStepName=
                                             step_name,
                                          distributionType=
                                             UNIFORM,
                                          fieldName='',
                                          fixed=OFF,
                                          localCsys=None,
                                          name=bc_name,
                                          region=region,
                                          u1=u1, u2=u2, u3=u3,
                                              ur1=ur1, ur2=ur2
                                             , ur3=ur3)


def get_edges_from_surface(part, surface_name):
    """
    Get edges associated with the specified surface in the
       given part.

    Parameters:
```

```
    - part: Reference to the ABAQUS part.
    - surface_name: Name of the surface in the part.

    Returns:
    - A list of edges associated with the surface.
    """
    face_tuple = part.surfaces[surface_name].faces
    edge_list = []
    for face in face_tuple:
        for edge_index in face.getEdges():
            edge = part.edges[edge_index]
            edge_list.append(edge)
    return edge_list


def get_edges_from_surfaces(part, surface_names):
    """
    Get edges associated with a list of specified surfaces in
       the given part.

    Parameters:
    - part: Reference to the ABAQUS part.
    - surface_names: List of surface names in the part.

    Returns:
    - A list of edges associated with the surfaces.
    """
    edge_list = []
    for surface_name in surface_names:
        edge_list.extend(get_edges_from_surface(part,
```

```
                surface_name))
    return edge_list


def create_mesh(model_name, part_name, material_name, size,
   special_seed_size=None):
    p = mdb.models[model_name].parts[part_name]

    # Seed the entire part with default size first
    p.seedPart(size=size, deviationFactor=0.1, minSizeFactor
       =0.1)

    # Determine the appropriate element types based on the
       material name
    if material_name == 'fiber':
        elem_types = (mesh.ElemType(elemCode=C3D8R,
           elemLibrary=STANDARD, secondOrderAccuracy=OFF,
                                   kinematicSplit=
                                      AVERAGE_STRAIN,
                                      hourglassControl=
                                      DEFAULT,
                                   distortionControl=DEFAULT)
                                      ,
                      mesh.ElemType(elemCode=C3D6, elemLibrary
                         =STANDARD),
                      mesh.ElemType(elemCode=C3D4, elemLibrary
                         =STANDARD))
        if special_seed_size:
            special_surfaces = ['fiber_coh_face']
            special_edges = get_edges_from_surfaces(p,
```

```
                special_surfaces)
            p.seedEdgeBySize(constraint=FINER, deviationFactor
                =0.1,
                             edges=special_edges,
                                minSizeFactor=0.1, size=
                                special_seed_size)
    elif material_name == 'matrix':
        elem_types = (mesh.ElemType(elemCode=C3D8R,
            elemLibrary=STANDARD),
                      mesh.ElemType(elemCode=C3D6, elemLibrary
                         =STANDARD),
                      mesh.ElemType(elemCode=C3D4, elemLibrary
                         =STANDARD))
        if special_seed_size:
            special_surfaces = [
                'bottom_matrix_coh_face', 'top_matrix_coh_face
                   ']
            special_edges = get_edges_from_surfaces(p,
                special_surfaces)
            p.seedEdgeBySize(constraint=FINER, deviationFactor
                =0.1,
                             edges=special_edges,
                                minSizeFactor=0.1, size=
                                special_seed_size)
    elif material_name == 'coh':
        elem_types = (mesh.ElemType(elemCode=COH3D8,
            elemLibrary=STANDARD),
                      mesh.ElemType(elemCode=COH3D6,
                         elemLibrary=STANDARD),
                      mesh.ElemType(elemCode=UNKNOWN_TET,
```

```
                                elemLibrary=STANDARD))

    # Set the element type
    cells = p.cells[:]
    pickedRegions = (cells, )
    p.setElementType(regions=pickedRegions, elemTypes=
        elem_types)

    # Generate the mesh
    p.generateMesh()


def create_coh_mesh(model_name, part_name, size, surface_names
    , special_surfaces, special_size):
    p = mdb.models[model_name].parts[part_name]

    # Access the edges of the provided surfaces using the
        helper function
    edges_main = get_edges_from_surfaces(p, surface_names)
    edges_special = get_edges_from_surfaces(p,
        special_surfaces)

    # Mesh the edges
    p.seedEdgeBySize(constraint=FINER, deviationFactor=0.1,
                     edges=edges_main, minSizeFactor=0.1, size
                        =size)
    p.seedEdgeBySize(constraint=FINER, deviationFactor=0.1,
                     edges=edges_special, minSizeFactor=0.1,
                        size=special_size)
```

```
    # Set element types specific for 'coh' material
    elem_types = (mesh.ElemType(elemCode=COH3D8, elemLibrary=
        STANDARD),
                  mesh.ElemType(elemCode=COH3D6, elemLibrary=
                      STANDARD),
                  mesh.ElemType(elemCode=UNKNOWN_TET,
                      elemLibrary=STANDARD))

    cells = p.cells[:]
    pickedRegions = (cells, )

    p.setElementType(regions=pickedRegions, elemTypes=
        elem_types)
    p.seedPart(size=size, deviationFactor=0.1, minSizeFactor
        =0.1)
    p.generateMesh()


def create_job(model_name, job_name='Job-1', description='',
   memory=90,
               numCpus=2, numDomains=2, numGPUs=1):
    """
    Create a job in Abaqus.

    Parameters:
    - model_name: Name of the model.
    - job_name: Name of the job. Default is 'Job-1'.
    - description: Description for the job. Default is empty
        string.
    - memory: Memory allocation. Default is 90.
```

```
    - numCpus: Number of CPUs. Default is 2.
    - numDomains: Number of domains. Default is 2.
    - numGPUs: Number of GPUs. Default is 1.
    """

    # Ensure the model assembly is updated
    mdb.models[model_name].rootAssembly.regenerate()

    # Create the job
    mdb.Job(atTime=None, contactPrint=OFF, description=
       description, echoPrint=OFF,
            explicitPrecision=SINGLE, getMemoryFromAnalysis=
               True, historyPrint=OFF,
            memory=memory, memoryUnits=PERCENTAGE, model=
               model_name, modelPrint=OFF,
            multiprocessingMode=DEFAULT, name=job_name,
               nodalOutputPrecision=SINGLE,
            numCpus=numCpus, numDomains=numDomains, numGPUs=
               numGPUs, queue=None,
            resultsFormat=ODB, scratch='', type=ANALYSIS,
               userSubroutine='',
            waitHours=0, waitMinutes=0)


def submit_job(model_name, job_name):
    """
    Regenerate the root assembly of the given model and submit
        the specified job.
    Then wait for the job to complete before returning.
```

```
    Parameters:
    - model_name: Name of the model.
    - job_name: Name of the job to submit.
    """
    mdb.models[model_name].rootAssembly.regenerate()
    mdb.jobs[job_name].submit(consistencyChecking=OFF)
    mdb.jobs[job_name].waitForCompletion()


# This block is executed only when this script is run directly
    (not when it's imported elsewhere)
if __name__ == "__main__":
    print("Running my_module.py standalone")
    print(function1())
    print(function2())
```

References

[1] Y Benveniste. A new approach to the application of Mori-Tanaka's theory in composite materials. *Mechanics of Materials*, 6(2):147–157, 1987.

[2] Tsu Wei Chou, Seiichi Nomura, and Minoru Taya. A Self-Consistent Approach to the Elastic Stiffness of Short-Fiber Composites. *Journal of Composite Materials*, 14(3):178–188, 7 1980.

[3] H. L. Cox. The elasticity and strength of paper and other fibrous materials. *British Journal of Applied Physics*, 3(3):72–79, 1952.

[4] lene Di Francis and Thomas C Ward. The single-fibre pull-out test. 1: Review and interpretation. *Composites: Part A*, 27:591–612, 1996.

[5] Bassem El Zoghbi. Modelling of Failure Mechanism in Unidirectional Carbon Fiber-Reinforced Polyamide Composites Using Cohesive Zone Model. *International Journal of Composite Materials*, 9(1):16–23, 2019.

[6] B. D. Ellis, D. L. McDowell, and M. Zhou. Simulation of single fiber pullout response with account of fiber morphology. *Cement and Concrete Composites*, 48:42–52, 2014.

[7] J. D. Eshelby. The determination of the elastic field of an ellipsoidal inclusion, and related problems. In *Collected Works of J. D. Eshelby*, pages 209–229. 2007.

[8] I I Gol'denblat and V A Kopnov. Strength of glass-reinforced plastics in the complex stress state. *Polymer Mechanics*, 1(2):54–59, 1965.

[9] Ge He, Yucheng Liu, T. E. Lacy, and M. F. Horstemeyer. A historical review of the traditional methods and the internal state variable theory for modeling composite materials. *Mechanics of Advanced Materials and Structures*, 29(18):2617–2638, 2022.

[10] Carl T Herakovich. Mechanics of composites: A historical review, 2012.

[11] R. Hill. The elastic behaviour of a crystalline aggregate. *Proceedings of the Physical Society. Section A*, 65(5):349–354, 1952.

[12] Mark F. Horstemeyer and Douglas J. Bammann. Historical review of internal state variable theory for inelasticity. *International Journal of Plasticity*, 26(9):1310–1334, 2010.

[13] Michael Hyer. *Stress Analysis of Fiber-Reinforced Composite Materials*. 1998.

[14] Ryan D. Jamison and Yu Lin Shen. Delamination analysis of metal–ceramic multilayer coatings subject to nanoindentation. *Surface and Coatings Technology*, 303:3–11, 2016.

[15] Yuanyuan Jia, Wenyi Yan, and Hong Yuan Liu. Carbon fibre pullout under the influence of residual thermal stresses in polymer matrix composites. *Computational Materials Science*, 62:79–86, 2012.

[16] T. Mori and K. Tanaka. Average stress in matrix and average elastic energy of materials with misfitting inclusions. *Acta Metallurgica*, 21(5):571–574, 1973.

[17] D J Mortell, D A Tanner, and C T Mccarthy. A virtual experimental approach to microscale composites testing. 2017.

[18] M. J. Pitkethly and J B Doble. Characterizing the fibre/matrix interface of carbon fibre-reinforced composites using a single fibre pull-out test. *Composites*, 21(5):389–395, 1990.

[19] Yiping Qiu and Peter Schwartz. A new method for study of the fiber-matrix interface in composites: Single fiber pull-out from a microcomposite. *Journal of Adhesion Science and Technology*, 5(9):741–756, 1991.

[20] B F Sørensen and H Lilholt. Fiber pull-out test and single fiber fragmentation test - Analysis and modelling. In *IOP Conference Series: Materials Science and Engineering*, volume 139, 2016.

[21] Ramesh Talreja. Stiffness properties of composite laminates with matrix cracking and interior delamination. *Engineering Fracture Mechanics*, 25(5-6):751–762, 1986.

[22] Stephen W Tsai and Edward M. Wu. A General Theory of Strength for Anisotropic Materials. *Journal of Composite Materials*, 5(1):58–80, 1971.