

# LAYING THE FOUNDATIONS FOR A NEW THICK LEVEL SET METHOD

*Student:*  
A. Poot

*Project Supervisors:*  
F.P. van der Meer  
L.A.T. Mororo

*Assessment Committee:*  
F.P. van der Meer  
L.A.T. Mororo  
R.Y. Peters  
L.J. Sluys



# Abstract

In this thesis, the basis for a general implementation of the Thick Level Set method is presented. This basis combines the general applicability of the TLS\_V1 method by Moës et al. [1] with the improved fracture mechanics of the TLS\_V2 method by Lé et al [2]. In order to accomplish this, the topological skeleton needs to be found for an arbitrary configuration of the iso-0 curve, and mapped onto the mesh. Additionally, a discontinuity in the displacement field needs to be applied on the skeleton curve. Since the iso-0 curve that determines the skeleton curve can be arbitrary, the displacement jump also needs to be designed for an arbitrary skeleton curve.

For the determination of the location of the skeleton curve, this thesis relies on a combination of the shrinking ball method by Ma et al. [3] and Prim's algorithm [4]. The resulting skeleton curve is then mapped onto the mesh using a newly developed set of algorithms. Having mapped the skeleton curve onto the mesh, the displacement jump is then modeled using the phantom node method by Hansbo and Hansbo [5].

During verification of the model, it is shown that the skeleton curve could be found for virtually any acyclical iso-0 curve. If the iso-0 curve was not acyclical, a single segment would be missing from the skeleton curve. Furthermore, it is demonstrated that the phantom node method can be used to define the displacement jump on the skeleton curve. Lastly, a mesh refinement study has been performed, which compared the results of the proposed model for 5 different mesh sizes. It was found that, for the rail shear test that was used to perform the verification, the shape of the iso-0 curve can vary randomly when the mesh is refined. The load-displacement curves, however, do not show significant dependence on the mesh size.





# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Academic context . . . . .	1
1.2	Focus and scope . . . . .	2
1.3	Thesis outline . . . . .	2
<b>2</b>	<b>Theory</b>	<b>3</b>
2.1	The Thick Level Set method . . . . .	3
2.1.1	Computing the level set field . . . . .	3
2.1.2	Computing the displacement field . . . . .	5
2.1.3	Computing the front evolution . . . . .	6
2.1.4	Extending TLS_V1 into TLS_V2 . . . . .	8
2.2	The shrinking ball method . . . . .	12
2.2.1	The shrinking ball algorithm . . . . .	13
2.2.2	Noise reduction . . . . .	17
2.3	The phantom node method . . . . .	20
2.3.1	Element and node definitions . . . . .	20
2.3.2	Integration scheme . . . . .	22
<b>3</b>	<b>Implementation</b>	<b>23</b>
3.1	Skeletonizer . . . . .	24
3.1.1	makeLinestrings . . . . .	25
3.1.2	makeClosedLoops . . . . .	28
3.1.3	makeGroupedLoops . . . . .	28
3.1.4	makeAtomGraph . . . . .	31
3.1.5	makeCrackPattern . . . . .	33
3.1.6	makeElemGraph . . . . .	35
3.1.7	makeShortCuts . . . . .	37
3.1.8	makePhantomNodes . . . . .	39
3.2	PhantomNodeModel . . . . .	43

3.2.1	createPhantomNodes . . . . .	43
3.2.2	updateElements and getOriginalElements . . . . .	45
3.2.3	Additional modifications . . . . .	45
<b>4</b>	<b>Verification</b>	<b>47</b>
4.1	Skeletonizer . . . . .	47
4.2	PhantomNodeModel . . . . .	53
4.3	Robustness . . . . .	56
<b>5</b>	<b>Conclusion</b>	<b>59</b>
5.1	Aim and research questions . . . . .	59
5.2	Limitations and recommendations . . . . .	60
5.3	Contribution to the field . . . . .	61
<b>A</b>	<b>Program listings</b>	<b>a</b>
	<b>Bibliography</b>	<b>1</b>

# Figures

2.1	A demonstration of the results of the fast marching algorithm. . . . .	8
2.2	A visualization of the way CZM and TLS_V1 are combined into TLS_V2 . .	9
2.3	An example of a material damage function $D(\phi)$ and interfacial damage $d(\phi_s)$ function that fulfill all requirements given by Lé et al. . . . .	10
2.4	The displacement field of a 1D bar where the three different models have been applied. . . . .	11
2.5	An overview of the shrinking ball algorithm by Ma et al. . . . .	13
2.6	A demonstration of the effects of a noisy boundary on the skeleton curve . .	17
2.7	A comparison of the results of the shrinking ball algorithm for a noise-free and noisy boundary. . . . .	18
2.8	A demonstration of the moment at which the medial ball ‘jumps’ from one side of the point cloud to the other. . . . .	18
2.9	A visualization of the connectivity between two phantom elements and the original nodes and the phantom nodes. . . . .	20
2.10	A comparison of the handling of a simple junction in the crack pattern by removing the original element, and by taking a shortcut through the element.	21
2.11	A comparison of the handling of a junction that spans two elements by removing the original element, and by taking a shortcut through the element.	21
2.12	A visualization of the displacement jump and phantom nodes at the crack tip. . . . .	21
2.13	The subdomain triangulation of intersected elements and their integration points. . . . .	22
3.1	A general overview of the Thick Level Set model. . . . .	23
3.2	A demonstration of the differences between closed linestrings, open linestrings, disconnected linestrings and enclosed linestrings . . . . .	25
3.3	The atom locations and connections describing of the skeleton curve. . . . .	31
3.4	The atomGraph that would be created by the makeAtomGraph function. . . . .	32
3.5	The elemGraph that would be created by the makeElemGraph function, based on the atomGraph shown in figure 3.3 . . . . .	35
3.6	A comparison of the atomGraph and the elemGraph that is produced by the makeElemGraph function . . . . .	37

3.7	A visualization of the way the <code>makePhantoNodes</code> function iterates over a complicated crack pattern . . . . .	39
3.8	An example of the procedure by which the next intersection point is determined by the <code>getNextIntxn</code> function in listing 3.10 . . . . .	40
4.1	An overview of the measurements of the model for the rail shear test. . . . .	48
4.2	The skeleton curve at different time steps . . . . .	49
4.3	A plot of the load factor $\gamma$ as a function of $t$ . . . . .	50
4.4	A comparison of the <code>atomGraph</code> and the <code>elemGraph</code> . . . . .	51
4.5	A visualization of the <code>elemGraph</code> for two additional edge cases. . . . .	52
4.6	An overview of the measurements of the model for the CT test. . . . .	53
4.7	A comparison of the skeleton curve and the resulting deformed specimen during the CT test. . . . .	54
4.8	A closeup of the deformed mesh at time step $t = 100$ . . . . .	55
4.9	The relative runtime of the <code>Skeletonizer</code> procedure plotted for different mesh sizes . . . . .	56
4.10	The skeleton curve at $t = t_{crack}$ for different mesh sizes . . . . .	57
4.11	The load-displacement diagram for different mesh sizes . . . . .	58

# Tables

3.1	The crack pattern that corresponds to the atomGraph shown figure 3.4 . . . .	33
3.2	An overview of which variables visual overview of the global algorithm for the TLS_V1. . . . .	46
4.1	The mesh sizes for which a simulation has been run, and the corresponding time steps at which a full crack is formed. . . . .	56



# Listings

2.1	Pseudocode for the shrinking ball algorithm by Ma et al. . . . .	14
2.2	Pseudocode for the ShrinkingBall-function by Ma et al. . . . .	15
2.3	Pseudocode for the ComputeCircle-function by Ma et al. . . . .	16
2.4	Pseudocode for the enhanced ShrinkingBall-function by Peters. . . . .	19
3.1	Pseudocode for the makeLinestrings function. . . . .	26
3.2	Pseudocode for the addSegmentsForward function. . . . .	27
3.3	Pseudocode for the makeClosedLoops function. . . . .	29
3.4	Pseudocode for the makeGroupedLoops function. . . . .	30
3.5	Pseudocode for the makeAtomGraph function. . . . .	32
3.6	Pseudocode for the makeCrackPattern function. . . . .	34
3.7	Pseudocode for the makeElemGraph function. . . . .	36
3.8	Pseudocode for the makeShortCuts function. . . . .	38
3.9	Pseudocode for the makePhantomNodes function. . . . .	40
3.10	Pseudocode for the getNextIntxn function. . . . .	42
3.11	Pseudocode for the createPhantomNodes function. . . . .	44
A.1	Pseudocode for the getIso0Elems function. . . . .	a
A.2	Pseudocode for the addSegmentsBackward function. . . . .	b
A.3	Pseudocode for the getNextElements function. . . . .	c
A.4	Pseudocode for the getNodeNegPos and getNodePosNeg functions. . . . .	d
A.5	Pseudocode for the edgeInPattern function. . . . .	e
A.6	Pseudocode for the addAllSegments and addAllSegmentsExcept functions. . . . .	f
A.7	Pseudocode for the getAngle function. . . . .	g
A.8	Pseudocode for the updateElements and getOriginalElements functions. . . . .	h

# Chapter 1

## Introduction

### 1.1 Academic context

Over the last 10 years, a new damage model has been developed, called the Thick Level Set (TLS) method [1, 6, 7, 2]. This method distinguishes itself from other damage models by defining the material damage as a function of the so-called 'level set field', rather than a direct function of the local or regional mechanical properties. The TLS method is an enrichment of the Level Set (LS) method, which was initially proposed by Sethian [8, 9] and Osher [9, 10] as a general method to model propagating fronts with curvature-dependent speeds. A key advantage of the LS method is that it allows for merging and branching of fronts in a way that previously required ad-hoc solutions [10]. In 2002, Allaire et al. [11] first applied the LS method in the context of structural mechanics, more specifically as a method for structural optimization [12, 13]. Later, this LS method was used to model damage mechanics by Allaire et al. [14]. In this LS model, only a damaged and an undamaged zone are distinguished, which are separated by iso-0 curve.

The main drawback of this LS method lies in the fact that there is no gradual degradation of the material properties possible. This might give accurate results for materials that exhibit semi-brittle failure behavior, but in case of a material that fails more gradually, this binary distinction will not suffice [1]. Furthermore, the LS model suffered from mesh dependencies including spurious localization [1, 7]. To resolve these issues, a critical length was introduced as a material property by Moës et al. [1] in 2010. Over this critical length, a gradual transition from undamaged to fully damaged occurs, which depends on the distance from the iso-0 curve. For this TLS model, a more general implementation was developed by Bernard et al. [6] and Van der Meer et al. [7], which allowed for arbitrary damage fronts, as well as generalization into 3D.

It should be noted that this initial TLS model (which will be referred to as TLS\_V1 from this point onwards) only allowed material damage<sup>1</sup>, though for many applications, an approach involving both material damage and interfacial damage<sup>2</sup> is required. To allow for the inclusion of interfacial damage in the TLS model, a new version of the TLS method (called TLS\_V2 from this point onwards) has been proposed by Lé et al. [2]. This model defines a displacement jump on the topological skeleton of the iso-0 curve. Similarly to the material damage in TLS\_V1, the interfacial damage on the skeleton curve is defined as a function of the level set field. In their proposal, a few test cases were used to demonstrate the improvements of the TLS\_V2 model, compared to Moës' initial proposal. However, all

---

<sup>1</sup>i.e. the gradual degradation of the elements' material properties

<sup>2</sup>i.e. the explicit modeling of a crack formation as a discontinuity in the displacement field through which energy dissipates



of these test cases made use of a trivial skeleton curve, either due to symmetry, or forced by the boundary conditions of the problem. A general implementation of TLS\_V2, similar to the one proposed by Bernard for TLS\_V1 has yet to be developed.

## 1.2 Focus and scope

This thesis aims to describe and develop a basis upon which a more generally applicable implementation of the TLS\_V2 model can be built. More specifically, it intends to answer the following research questions:

- How can the skeleton curve be defined for a given damage front?
- How can the skeleton curve be discretised for any given triangular 2D mesh?
- How can the displacement jump be defined using this discretised skeleton curve?

The reason that the main focus of this thesis concerns a 2D mesh consisting of triangular elements is twofold. Firstly, this is the most simple kind of mesh element, which limits the number of edge cases that need to be taken into account when creating a discretised version of the skeleton curve. Also, the definition of a skeleton curve is more straightforward for 2D than for 3D, as well as the implementation of the phantom node method. Secondly, this thesis aims to support ongoing research into the fracture mechanics of Fibre Reinforce Polymer (FRP) materials at TU Delft by Van der Meer, Mororó, Sluys et al. [15, 16, 7, 17, 18]. The current TLS\_V1-model used by the research group only functions for triangular elements, and due to time constraints it will not be possible to adapt the program for quadrilateral elements as well. For these reasons, the choice has been made to create and analyze an implementation of TLS\_V2 which will be validated for structures with a 2D triangular mesh.

## 1.3 Thesis outline

This thesis will be split into four main sections: First, the background theory required to develop an implementation of TLS\_V2 will be discussed in chapter 2. In section 2.1, a detailed description of TLS\_V1 will be given, mainly following the approaches used by Bernard et al. [6] and Meer et al. [7]. Afterwards, the modifications proposed by Le et al. [2] to arrive at the TLS\_V2 method will be explained. Additionally, the theory behind the shrinking ball method and the phantom node method will be discussed in sections 2.2 and 2.3, respectively. Both of these methods will be used later to create a basis for an implementation of TLS\_V2. Having layed out all required background knowledge, an implementation of a basis for TLS\_V2 will be proposed in chapter 3. The determination of the skeleton curve for any given configuration of the iso-0 curve will be discussed in section 3.1. Afterwards, in section 3.2, a method by which the displacement jump can be included in the model will be described. These proposed implementations will then be verified in chapter 4, from which a conclusion will be drawn in chapter 5.

# Chapter 2

## Theory

### 2.1 The Thick Level Set method

To properly understand TLS\_V2, it is important to first gain a thorough understanding of TLS\_V1, since the TLS\_V2 model heavily relies on TLS\_V1. For this reason, the TLS\_V1 model will first be completely described in sections 2.1.1 through 2.1.3, after which the modifications that need to be made in order to arrive at TLS\_V2 will be laid out in section 2.1.4.

Both the TLS\_V1 model and the TLS\_V2 model consist of three main procedures in each time step, which can be summarized as follows [1, 6, 19, 7]:

1. Compute the level set field
  - (a) Check for initiation if needed.
  - (b) Compute the updated level set field based on the front velocity.
2. Compute the mechanical equilibrium solution
  - (a) Assemble the linear system of equations for the mechanical problem
  - (b) Solve this system of equations to find the displacement field
3. Compute the evolution of the level set front
  - (a) Assemble the linear system of equations for the averaged energy release rate
  - (b) Solve this system of equations and compute the front velocity
  - (c) Extend this solution over the whole mesh

In the following sections, each of these steps will be discussed more elaborately.

#### 2.1.1 Computing the level set field

The basis of the TLS method is given by the iso-0 curve, which is defined as the outer boundary of the damaged zone, and denoted as  $\Gamma_0$ . On the entire domain  $\Omega$ , a level set field  $\phi$  is defined, which is equal to 0 on the iso-0 curve. The gradient of the level set field has a magnitude equal to 1 over the whole domain. It can be fully described by

$$\begin{aligned} |\nabla\phi(x)| &= 1 \mid x \in \Omega \\ \phi(x) &= 0 \mid x \in \Gamma_0 \end{aligned} \tag{2.1}$$

This definition is equivalent to the signed distance function of  $\Gamma_0$  [1, 19]. In other words, the value of  $\phi$  is equal to the signed distance from the iso-0 curve at any point in the domain. In this thesis, a positive value of  $\phi$  corresponds to a location within the damaged zone, whereas a negative value corresponds to a location outside the damaged zone.

As an enhancement of the LS method by Allaire et al. [14], a critical length  $l_c$  is introduced in the TLS method<sup>1</sup>, which represents the distance from  $\Gamma_0$  at which the material is fully damaged. For the region between  $\phi = 0$  and  $\phi = l_c$ , a damage function  $F(\phi)$  is introduced. To ensure continuity, it must be that  $F(0) = 0$  and  $F(l_c) = 1$ . The material damage  $D(\phi)$  can thus be written as a function of  $\phi$  as follows

$$\begin{cases} D(\phi) = 0 & | \phi < 0 \\ D(\phi) = F(\phi) & | 0 \leq \phi \leq l_c \\ D(\phi) = 1 & | \phi > l_c \end{cases} \quad (2.2)$$

where  $F(\phi)$  is subjected to the following conditions

$$\begin{cases} F(\phi) = 0 & | \phi = 0 \\ F'(\phi) \geq 0 & | 0 \leq \phi \leq l_c \\ F(\phi) = 1 & | \phi = l_c \end{cases} \quad (2.3)$$

In principle, any function that fulfills the conditions put forth in equation 2.3 can be used as a damage function for the partially damaged zone, such as a parabolic damage profile [2], or one which is based on an arctangent [6, 7]. In fact, a discontinuous damage profile could also be used, provided that additional terms are included for any quantities that are based on the spatial derivative of  $D(\phi)$  [1].

Given the relations in equations 2.2 and 2.3, only the level set field  $\phi$  needs to be determined in order to find the value of  $D$  over the full domain  $\Omega$ . As explained in the introduction of section 2.1, the velocity of the front in its normal direction  $v_n$  is determined at the end of each time step. Accordingly, the level set field at time step  $t$  can be computed by using a forward Euler method, discretized in time, as follows [19]:

$$\phi_t = \phi_{t-1} + v_n \Delta t \quad (2.4)$$

Here,  $\Delta t$  refers to the time step size. It is possible to put an upper limit on the crack growth that is allowed by the model, for example by simply introducing a maximum front advance  $a_{\max}$  [6, 7], or by limiting the size of the time step  $\Delta t$ , and reducing the prescribed displacements accordingly [19].

Of course, this level set field update can only be used to extend the existing level set field, and cannot by itself initiate new cracks. To this end, a crack initiation procedure is necessary. Generally, a critical energy release rate  $Y_c$  is used to determine whether a new crack develops. This parameter can be based on the fracture energy  $G_c$  [1, 6, 19], the tensile strength  $f_t$  [7], or a combination of these two [7]. Regardless of the criterion that is used, when the criterion is met at a given location in the mesh, a sufficiently small circle<sup>2</sup> will be introduced to the iso-0 curve, and the level set field will be updated following equation 2.4. To do this, the fast marching algorithm by Sethian [8, 9] can be used, which will be explained in section 2.1.3.

Theoretically it is not necessary to run the fast marching algorithm when no new damage zones are initiated, since the level set field  $\phi$  and front velocity  $v_n$  are known over

<sup>1</sup>This critical length is the ‘thickness’ to which the term *Thick* Level Set method refers

<sup>2</sup>For 2D problems, a circle will be added to the level set field. For 3D problems, this will be a sphere.

the entire domain from the previous time step. However, due to instability of the forward Euler method, the resulting value of  $\phi$  might start to drift away from its true value. This drifting behavior, can be prevented by occasionally recomputing the level set field using the fast marching method [19].

### 2.1.2 Computing the displacement field

With  $\phi$ , and consequently  $D(\phi)$  fully known, the displacement field can be computed. The process of building up the global stiffness matrix  $\mathbf{K}$  and global external force vector  $\mathbf{f}_{ext}$ , and solving the following system of equations

$$\mathbf{K} \mathbf{u} = \mathbf{f}_{ext} \quad (2.5)$$

is well known for elastic materials.  $\mathbf{K}$  and  $\mathbf{f}_{ext}$  are built up on the elemental stiffness matrix  $\mathbf{K}^e$  and elemental external force vector  $\mathbf{f}_{ext}^e$ , which are defined according to

$$\begin{aligned} \mathbf{K}^e &= \int_{\Omega^e} \mathbf{B}^T \cdot \mathbf{D} \cdot \mathbf{B} d\Omega \\ \mathbf{f}^e &= \int_{\Omega^e} \mathbf{N}^T \cdot \mathbf{t} d\Omega \end{aligned} \quad (2.6)$$

The strain energy per unit volume  $\Psi$  in the material can be written as a function of the strain vector  $\boldsymbol{\varepsilon}$  only, using

$$\Psi(\boldsymbol{\varepsilon}) = \frac{1}{2} \boldsymbol{\varepsilon} \cdot \mathbf{D} \cdot \boldsymbol{\varepsilon} \quad (2.7)$$

From this relation, a simple expression for the stress tensor  $\boldsymbol{\sigma}$  can be found:

$$\boldsymbol{\sigma} = \frac{\partial \Psi}{\partial \boldsymbol{\varepsilon}} = \mathbf{D} \cdot \boldsymbol{\varepsilon} \quad (2.8)$$

This formulation, however, only works for linear elastic materials. To include the non-linear effects of damage growth, a spectral decomposition is required, after which  $\Psi$  can be linked to the principal strain vector  $\bar{\boldsymbol{\varepsilon}}$  for isotropic materials using the following relation [1, 6]:

$$\Psi(\bar{\boldsymbol{\varepsilon}}, D) = \mu \bar{\boldsymbol{\varepsilon}} \cdot (\mathbf{I} - \bar{\boldsymbol{\alpha}} D) \cdot \bar{\boldsymbol{\varepsilon}} + \frac{1}{2} \lambda (1 - \alpha_v D) \varepsilon_v^2 \quad (2.9)$$

In this relation,  $\varepsilon_v$  represents the volumetric strain, which can be obtained by taking the trace of  $\bar{\boldsymbol{\varepsilon}}$ , and  $\mu$  and  $\lambda$  are the Lamé parameters, which can be obtained from  $\mathbf{D}$ .  $\mathbf{I}$  is the second-order identity tensor, and  $\bar{\boldsymbol{\alpha}}$  is a diagonal matrix, which accounts for the difference between compressive and tensile strains in damaged elements in each principal direction. The scalar  $\alpha_v$  fulfills the same function for the volumetric strain. If for both  $\bar{\boldsymbol{\alpha}}$  and  $\alpha_v$ , it is assumed that the damage does not affect the material behavior under compression, and that the damages has full effect under tension, it follows that [7]:

$$\begin{cases} \bar{\alpha}_{ii} = 0 & | \bar{\varepsilon}_i \leq 0 \\ \bar{\alpha}_{ii} = 1 & | \bar{\varepsilon}_i > 0 \\ \alpha_v = 0 & | \varepsilon_v \leq 0 \\ \alpha_v = 1 & | \varepsilon_v > 0 \end{cases} \quad (2.10)$$

From this relationship, the principal stress tensor  $\bar{\sigma}$  and energy release rate  $Y$  can be found using

$$\begin{cases} \bar{\sigma} = \frac{\partial \Psi}{\partial \bar{\varepsilon}} = 2\mu(\mathbf{I} - \bar{\alpha}D) \cdot \bar{\varepsilon} + \lambda(1 - \alpha_v D)\varepsilon_v \\ Y = \frac{\partial \Psi}{\partial D} = -\mu\bar{\varepsilon} \cdot \bar{\alpha} \cdot \bar{\varepsilon} - \frac{1}{2}\lambda\alpha_v\varepsilon_v^2 \end{cases} \quad (2.11)$$

For pure tension and pure compression, the relationships from equation 2.11 can be simplified to those found below [6]. Note that in this case, the conditions need to apply to all (diagonal) components of  $\bar{\varepsilon}$  in order for the relationship to hold.

$$\begin{cases} \bar{\sigma} = \frac{\partial \Psi}{\partial \bar{\varepsilon}} = 2\mu(1 - D)\mathbf{I} \cdot \bar{\varepsilon} + \lambda(1 - D)\varepsilon_v \\ Y = \frac{\partial \Psi}{\partial D} = \mu\bar{\varepsilon} \cdot \bar{\varepsilon} + \frac{1}{2}\lambda\varepsilon_v^2 \end{cases} \quad | \bar{\varepsilon} \geq 0 \quad (2.12)$$

$$\begin{cases} \bar{\sigma} = \frac{\partial \Psi}{\partial \bar{\varepsilon}} = 2\mu(\mathbf{I} - \bar{\alpha}D) \cdot \bar{\varepsilon} + \lambda(1 - \alpha_v D)\varepsilon_v \\ Y = \frac{\partial \Psi}{\partial D} = \mu\bar{\varepsilon} \cdot \bar{\alpha} \cdot \bar{\varepsilon} + \frac{1}{2}\lambda\alpha_v\varepsilon_v^2 \end{cases} \quad | \bar{\varepsilon} \leq 0 \quad (2.13)$$

Using the eigenvectors obtained during the spectral decomposition and the principal stresses  $\bar{\sigma}$ , the stresses in the original coordinate system  $\sigma$  can be retrieved. With the relationships between  $\phi$ ,  $D$ ,  $\varepsilon$  and  $\sigma$  now fully known, it is possible to determine  $\mathbf{K}^e$  and  $\mathbf{f}_{ext}^e$ , build up  $\mathbf{K}$  and  $\mathbf{f}_{ext}$ , and find the displacement field by solving the system of equations 2.5. It should be noted that so far, only the material damage  $D$  has been taken into account. How these expressions need to be modified in order to include the interfacial damage  $d$  will be explained in section 2.1.4.

### 2.1.3 Computing the front evolution

For elastic materials, the total amount of energy that has dissipated from the system,  $E(\mathbf{u}, \phi)$  is given by the difference between the work exerted onto the system and the potential energy that is still stored in the system in the form of strain energy. This relationship can be written as

$$E(\mathbf{u}, \phi) = \int_{\Gamma_N} \mathbf{f} \cdot \mathbf{u} d\Gamma - \int_{\Omega} \Psi(\varepsilon(\mathbf{u}), D(\phi)) d\Omega \quad (2.14)$$

A small movement of the level set front  $\delta\phi$  produces a small amount of dissipated energy  $\delta E$  according to

$$\begin{aligned} \delta E(\mathbf{u}, \phi) &= E(\mathbf{u}, \phi + \delta\phi) - E(\mathbf{u}, \phi) \\ &= - \int_{\Omega} \Psi(\varepsilon(\mathbf{u}), D(\phi + \delta\phi)) d\Omega + \int_{\Omega} \Psi(\varepsilon(\mathbf{u}), D(\phi)) d\Omega \\ &= - \int_{\Omega} \delta\Psi(\varepsilon(\mathbf{u}), D(\phi)) d\Omega \\ &= - \int_{\Omega} \frac{\partial \Psi}{\partial \varepsilon} \delta\varepsilon + \frac{\partial \Psi}{\partial D} \frac{\partial D}{\partial \phi} \delta\phi d\Omega \\ &= - \int_{\Omega} Y D'(\phi) \delta\phi d\Omega \end{aligned} \quad (2.15)$$

By changing the coordinate system into a curvilinear system, as was done by Bernard et al. [6], the term  $d\Omega$  is changed to  $\left(1 - \frac{\phi}{\rho}\right) d\phi ds$ , which yields the following expression for  $\delta E$

$$\delta E(\phi) = - \int_{\Gamma_0} \int_0^l Y(\phi, s) D'(\phi) \left(1 - \frac{\phi}{\rho}\right) d\phi \delta\phi(s) ds = - \int_{\Gamma_0} g(s) \delta\phi(s) ds \quad (2.16)$$

In this expression,  $g(s)$  is the configurational force along the level set front  $\Gamma_0$ . Bernard et al. noted, however, that for damage initiation,  $l$  approaches 0, and thus, the configurational force also approaches 0 [6]. To resolve this issue, an averaged energy release rate  $\bar{Y}$  is introduced, which is defined such that

$$g(s) = \int_0^l Y(\phi, s) D'(\phi) \left(1 - \frac{\phi}{\rho(s)}\right) d\phi = \int_0^l \bar{Y}(s) D'(\phi) \left(1 - \frac{\phi}{\rho(s)}\right) d\phi \quad (2.17)$$

Following Van der Meer et al. [7], equation 2.17 can be approximated by applying the Galerkin method and weakly enforcing that  $\bar{Y}$  is constant over  $\phi$  by including Lagrange multipliers that demand that  $\nabla \bar{Y} \cdot \nabla \phi = 0$ . This yields the following system of equations

$$\begin{bmatrix} \mathbf{K} & \mathbf{L} \\ \mathbf{L} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \bar{\mathbf{Y}} \\ \mathbf{l} \end{bmatrix} = \begin{bmatrix} \mathbf{f}^Y \\ \mathbf{0} \end{bmatrix} \quad (2.18)$$

in which  $\mathbf{K}$ ,  $\mathbf{L}$  and  $\mathbf{f}^Y$  are given by [7, 20, 17]:

$$\begin{aligned} K_{ij} &= \int_{\Omega_d} D'(\phi) N_i N_j + \frac{\kappa h^2}{l_c} \frac{\partial N_i}{\partial x_k} \frac{\partial N_j}{\partial x_k} d\Omega \\ L_{ij} &= \int_{\Omega_d} l_c \left( \frac{\partial N_i}{\partial x_k} \frac{\partial \phi}{\partial x_k} \right) \left( \frac{\partial N_j}{\partial x_k} \frac{\partial \phi}{\partial x_k} \right) d\Omega \\ f_i^Y &= \int_{\Omega_d} N_i D'(\phi) Y d\Omega \end{aligned} \quad (2.19)$$

Here,  $N_i$  is the shape functions which corresponds to node  $i$ .  $\kappa$  is a stabilization parameter and  $h$  refers to the typical element size in the domain  $\Omega_d$ , which is the damaged area of the material.  $x_k$  refers to the coordinates in the  $k^{\text{th}}$  direction.

This value of  $\bar{Y}$  has to be compared to a parameter  $Y_c$ , which represents the resistance of the material to damage growth. To properly compare  $\bar{Y}$  and  $Y_c$ , it is necessary to determine its averaged value over  $\phi$ ,  $\bar{Y}_c$ , which can be done by following the same procedure as for  $\bar{Y}$  [6, 7]. However, in case this material property is uniform over the material, it is not necessary to perform this calculation, since it is obvious that  $\bar{Y}_c = Y_c$  [7, 17].

In order to ensure adequate crack growth, the load in the next time step is linked to the averaged energy release rate via a load factor  $\gamma_{t+1}$  [7]. This load factor is used to scale the results from the unit load that is applied to their actual values. It is constructed in such a way that the following condition is satisfied

$$\gamma_{t+1}^2 \left( \frac{\bar{Y}}{\bar{Y}_c} \right)_{\max} = 1 \quad (2.20)$$

Here, the ratio between  $\bar{Y}$  and  $\bar{Y}_c$  has been maximized over the nodes. The load factor  $\gamma_{t+1}$  then ensures that at least in one node, the resistance of the material to crack growth is reached. Following Van der Meer et al. [7], the level set increment  $a$  for each node  $i$  is given by

$$a_i = \max \left\langle \frac{a_{\max}}{c-1} \left( \frac{c \gamma_{t+1}^2 \bar{Y}_i}{\bar{Y}_{ci}} - 1 \right), 0 \right\rangle \quad (2.21)$$

The spreading parameter  $c$  is intended to influence the spread of movement. It has to be larger than 1 to ensure that  $\bar{Y}_c$  is met for at least one node, though it should be noted that values of  $c$  that are very close to 1 will lead to a very slow simulation, since only one node would be incorporated into the damage zone for each time step. To avoid damage regression, a minimum value of 0 has been included in the calculation of  $a_i$ .

For any point in the damaged zone  $\Omega_d$ , the level set increment  $a$  is now known, which also yields the front movement velocity  $v$  via the relation  $a(s) = v(s) \Delta t$  [7]. To update the level set field in the next time step, the velocity  $v$  needs to be known over the whole mesh, as shown in equation 2.4. For this purpose, the ‘fast marching method’ by Sethian [8, 9] can be used, which is able to map the velocity values of the damage zone onto the entire mesh [1, 19], as shown in figure 2.1.

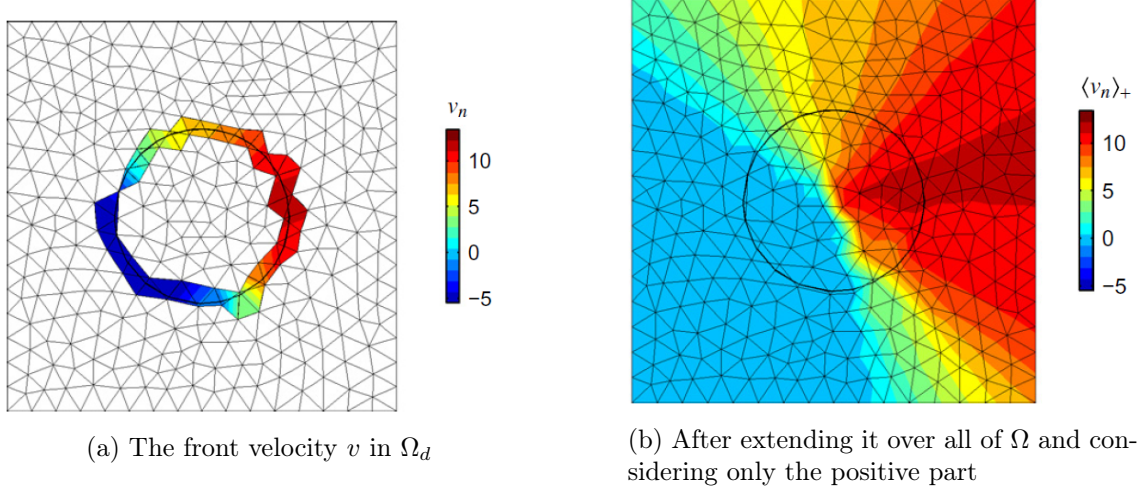


Figure 2.1: A demonstration of the results of the fast marching algorithm. [19]

Essentially, the fast marching method works by determining repeatedly which node would be reached first for a given front velocity. A narrow band of nodes that are taken into consideration is determined based on the connectivity around the level set front. If out of the nodes that are taken into consideration, the node that would be reached earliest is found, the extended velocity at that node is known. The node is then removed from the set of nodes that is taken into consideration, and all surrounding nodes that have not been treated are added to this set. The process is then repeated until no nodes are left [8, 9].

Since only a relatively small part of the mesh is considered each time, the procedure can be executed very quickly [9, 21]. Due to its low cost of operation, it can be used at every time step to recompute the level set field based on the iso-0 curve [19], as mentioned in section 2.1.1.

#### 2.1.4 Extending TLS\_V1 into TLS\_V2

In the TLS model described so far, material damage was only modeled as a gradual process in which the stiffness of the damaged material slowly reduces to 0 as the damage increases. Although the displacement jump representing the crack formation is modeled implicitly when no stiffness is left in the material, this limits the flexibility of the crack model. Two disadvantages mentioned by Lé et al. [2] of the TLS\_V1 model are the fact that very large strains are needed to model the displacement jump, and that it does not allow for interfacial models such as traction forces on the cracked surface. To improve the model, they propose to combine the TLS\_V1 model with the Cohesive Zone Model (CZM), which produces the TLS\_V2 model.

In figure 2.2, the three models are compared. As shown, the TLS\_V2 model incorporates both the level set-based damage progression from the TLS\_V1 model and the crack formation from the CZM. Note that the CZM does not affect the initial damage

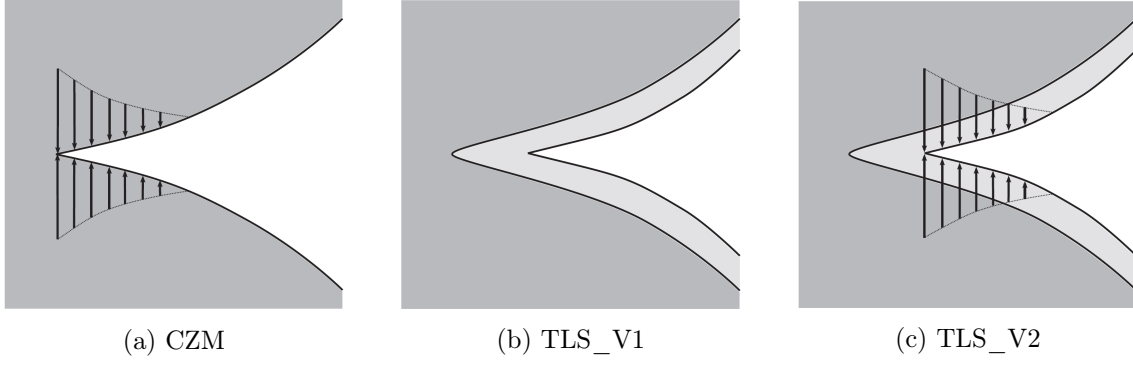


Figure 2.2: A visualization of the way CZM and TLS\_V1 are combined into TLS\_V2 [2].

model, and is only incorporated when the level set has reached a certain critical value  $\phi^*$ .

For the CZM, the amount of energy dissipated due to the crack formation  $\psi$  can be written as a function of the displacement jump  $w$  and interfacial damage  $d$ . Though several functions describing this relationship exist, the following is given by Lé et al [2], and will also be used here as an example:

$$\psi(w, d) = \frac{1}{2} k \left( \frac{1}{d} - 1 \right) w^2 \quad (2.22)$$

Using the same approach as in equation 2.11, the cohesive traction  $t(d, w)$  and cohesive energy release rate  $y(d, w)$  can be found via the following relations:

$$\begin{aligned} t(d, w) &= \frac{\partial \psi}{\partial w} = k \left( \frac{1}{d} - 1 \right) w \\ y(d, w) &= -\frac{\partial \psi}{\partial d} = \frac{1}{2} k \left( \frac{1}{d} \right)^2 w^2 \end{aligned} \quad (2.23)$$

In order to combine the damage model from TLS\_V1 and the CZM, equation 2.2 needs to be modified in such a way that the material damage  $D(\phi)$  no longer reaches 1 at  $\phi = l_c$ . This ensures that the material does not lose all its stiffness, and thus avoids infinite strains. The remaining energy is dissipated through the interfacial damage  $d(\phi_s)$ . The modified set of equations for  $D(\phi)$  is given below:

$$\begin{cases} D(\phi) = 0 & | \phi < 0 \\ D(\phi) = F(\phi) & | 0 \leq \phi \leq l_c \\ D(\phi) = F(l_c) < 1 & | \phi > l_c \end{cases} \quad (2.24)$$

where  $F(\phi)$  is subjected to the following conditions

$$\begin{cases} F(\phi) = 0 & | \phi = 0 \\ F'(\phi) \geq 0 & | 0 \leq \phi \leq l_c \\ F(\phi) < 1 & | \phi = l_c \end{cases} \quad (2.25)$$

Similarly to how the material damage  $D(\phi)$  is defined as a function of the level set field  $\phi$ , the interfacial damage  $d(\phi_s)$  is defined as a function of the level set field on the skeleton curve  $\phi_s$ . To allow for the full formation of a crack,  $d(\phi_s)$  has to be defined in such a way that it does reach 1 at  $\phi_s = l_c$ . On the other hand, since the introduction of the discontinuity should only occur after a significant amount of damage has developed,  $d(\phi_s)$



should be equal to 0 for  $\phi_s \leq \phi_s^*$  rather than for  $\phi_s \leq 0$ . The following sets of equations can be used to describe the relation between  $d(\phi_s)$  and  $\phi_s$ :

$$\begin{cases} d(\phi_s) = 0 & | \phi_s < \phi_s^* \\ d(\phi_s) = f(\phi_s) & | \phi_s^* \leq \phi_s \leq l_c \\ d(\phi_s) = 1 & | \phi_s > l_c \end{cases} \quad (2.26)$$

where the damage function  $f(\phi_s)$  is subjected to the following conditions

$$\begin{cases} f(\phi_s) = 0 & | \phi_s = \phi_s^* \\ f'(\phi_s) \geq 0 & | \phi_s^* \leq \phi_s \leq l_c \\ f(\phi_s) = 1 & | \phi_s = l_c \end{cases} \quad (2.27)$$

In figure 2.3, an example of a material damage function  $D(\phi)$  that fulfills the conditions from equations 2.24 and 2.25 and an interfacial damage function  $d(\phi_s)$  that fulfills the conditions from equations 2.26 and 2.27 is given.

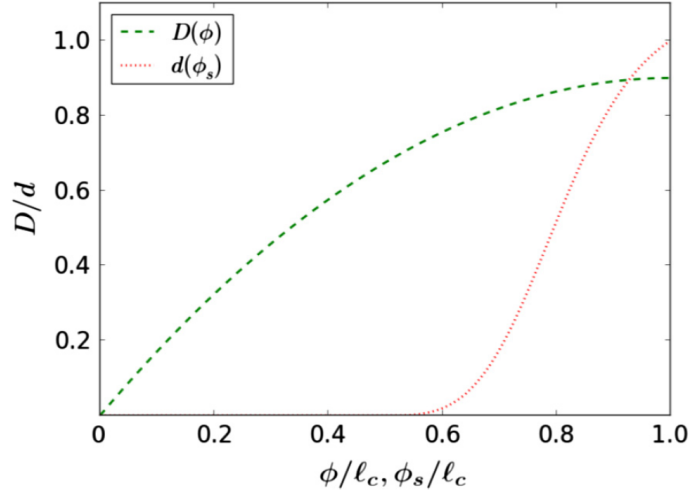


Figure 2.3: An example of a material damage function  $D(\phi)$  and interfacial damage  $d(\phi_s)$  function that fulfill all requirements given by Lé et al. [2]

In order to account for the additional dissipated energy, equation 2.17, which describes the configurational force  $g(s)$ , will also need to be modified. Lé et al. [2] give the following expression for  $g(s)$ :

$$g(s) = \int_0^{\phi_s} Y(\phi, s) D'(\phi) d\phi + \frac{1}{2} y d'(\phi_s) \Big|_{\phi=\phi_s} \quad (2.28)$$

This expression only applies to straight segments, however. It is not possible to simply apply the same modifications as those by Bernard et al. [6], since there is no direct way to map the skeleton boundary  $\Gamma_s$  onto the iso-0 curve  $\Gamma_0$ . To overcome these issues, Lé et al. [2] suggest to use the modal approach by Moreau et al. [22]. To use this approach, the following variational form needs to be solved:

$$\int_{\Gamma_0} g Y^* d\Gamma = \int_{\Omega} Y D'(\phi) Y^* d\Omega + \int_{\Gamma_s} \frac{1}{2} y d'(\phi_s) d\Gamma \quad | \quad \forall Y^* \in \mathcal{Y} \quad (2.29)$$

where  $\mathcal{Y}$  is the set of all fields that are constant along  $\nabla\phi$ . The set of admissible solutions  $\mathcal{Y}$  is then discretized into a number of nodes  $Y_i^*$  [2, 22]. The discretized configurational

force can then be written as a sum of these modes according to

$$g = \sum g_i Y_i^* \quad (2.30)$$

where each coefficient  $g_i$  is given by the following expression:

$$g_i = \frac{\int_{\Omega} Y D'(\phi) Y_i^* d\Omega + \int_{\Gamma_s} \frac{1}{2} y d'(\phi_s) Y_i^* d\Gamma}{\int_{\Gamma_0} Y_i^* d\Omega} \quad (2.31)$$

In order to give a basic impression of the difference in the results of the CZM, TLS\_V1 and TLS\_V2, Lé et al. [2] provide an example that considers a simple 1D bar. The resulting displacements fields for the three models are given in figure 2.4. As one might expect, the TLS\_V2 model yields a displacement field that lies somewhere between the CZM and TLS\_V1 model, since it incorporates material damage from TLS\_V1 as well as interfacial damage from the CZM.

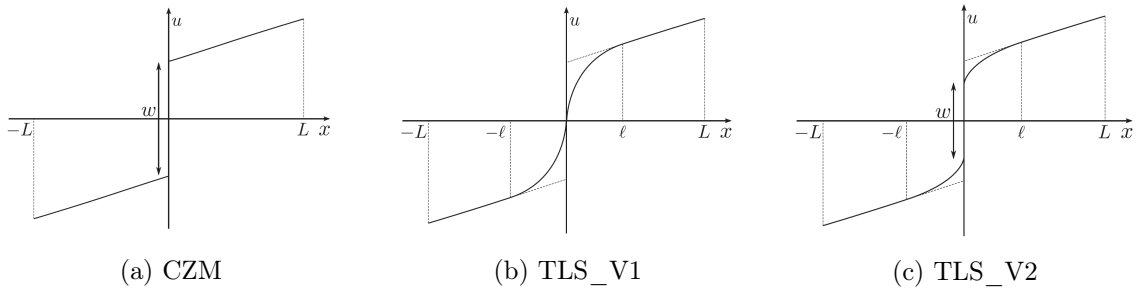


Figure 2.4: The displacement field of a 1D bar where the three different models have been applied. [2]

## 2.2 The shrinking ball method

Since the topological skeleton of the iso-0 curve plays an integral part in the TLS\_V2 method, the determination of the location of the skeleton curve is a critical part of thesis. Although Lé et al. [2] define the skeleton curve as the set of points where the gradient of the level set field is discontinuous, this definition is not very useful when determining the location of the skeleton curve. This is because, since the level set field is only defined in the nodes of the mesh, the determination of the level set field as a continuous function requires extrapolation and interpolation from these nodal values.

Instead, the fact that the level set field  $\phi$  is equivalent to the signed distance function of the iso-0 curve can be used to more easily determine the location of the skeleton curve. Many approaches to finding the medial axis of a shape have been proposed, most of which give similar results for continuous sets, though discrete sets and digital grids may give different results, depending on the method that is applied [23].

The initial method for finding the skeleton curve, as proposed by Blum [24], considers a fire with a constant propagation speed. The boundary of the original shape is set on fire, and wherever two fronts meet, the fire is quenched. The collection of these so-called ‘quench points’ describes the topological skeleton of the shape. For closed shapes that are not strictly convex, this method would generate quench points both inside and outside the object, which might appear to contradict the fourth property of the medial axis mentioned earlier. However, using only the quench points lying within the original shape, the original shape can still be retrieved from the skeleton curve and radius function, and thus no information is lost by discarding the quench points lying outside the original shape [24].

Although this definition is quite intuitive, it does not have the most straightforward implementation, since there is not a way to directly tell beforehand at which distance from the boundary a given fire front will be quenched. Rather, the skeleton curve will be defined as the set of centers of all bitangential inscribed circles (or ‘disks’) of the original shape<sup>3</sup>. These centers can be found by picking a point of the edge of the original shape, drawing a tangent circle from that point, and shrinking it until it touches the shape in at least one other point, and has no intersections with the original shape [25].

As a final remark, the terms ‘medial axis’ and ‘skeleton curve’ can or cannot be used interchangeably. In their original paper [24], Blum coins the term ‘medial axis’, but notes that it can also be referred to as the ‘skeleton’ [24] of a shape. According to Saha et al. [23], a common distinction between the two terms is that the skeleton curve has the additional property that it allows for the original shape to be restored from the skeleton curve, whereas this is not possible for the medial axis. On the other hand, the conversion from the original shape to its skeleton and vice versa is commonly called the ‘Medial Axis Transformation’ (MAT), and it is common for authors to distinguish between the ‘skeleton function’ and the ‘radius function’, and refer to the set of these two functions as the medial axis [26, 27, 28]. Regardless, for thesis, only the transformation from the original shape to the skeleton curve is required, which means that only the skeleton function needs to be found, and the radius function can be discarded. For that reason, the shape will be referred to as ‘skeleton curve’ rather than ‘medial axis’ in this thesis.

---

<sup>3</sup>Note that this definition only applies to a 2D shape. For 3D shapes, the centers of inscribed spheres would need to be used, and so forth for higher-dimensional shapes

### 2.2.1 The shrinking ball algorithm

An algorithm, called the ‘shrinking ball algorithm’, has been proposed by Ma et al. [3] as an efficient method of finding the set of bitangential disks of a surface. While this method has been designed to find the skeleton curve of point clouds rather than continuous shapes, it can also be used to find the skeleton curve for any discretized 2D shape, since generating a point cloud from a given polygon is relatively straightforward. In figure 2.5, the iterative process is demonstrated. Initially, a point  $p$  on the surface  $\mathbb{S}$  is chosen at random, and a circle with radius  $r_{init}$  is generated. For the center of this circle,  $c$ , a ‘nearest neighbor’ algorithm is used to find its closest neighbor in the point cloud,  $p_i$  (see figure 2.5a). Then, a new circle is generated, which touches the surface in  $p$ , and runs through  $p_i$  (see figure 2.5b). Both the radius  $r$  and the center  $c_p$  of this circle can be found using some basic trigonometry. This process repeats until the radius no longer changes, at which point the maximum touching circle originating from  $p$  has been found. Now, the maximum disk will be searched for the next point,  $q$  (see figure 2.5c). Rather than using an initial radius  $r_{init}$ , the point  $p_i$  will be reused as a starting point. This process will repeat until the maximal inscribed circle has been found for all points on the surface  $\mathbb{S}$ .

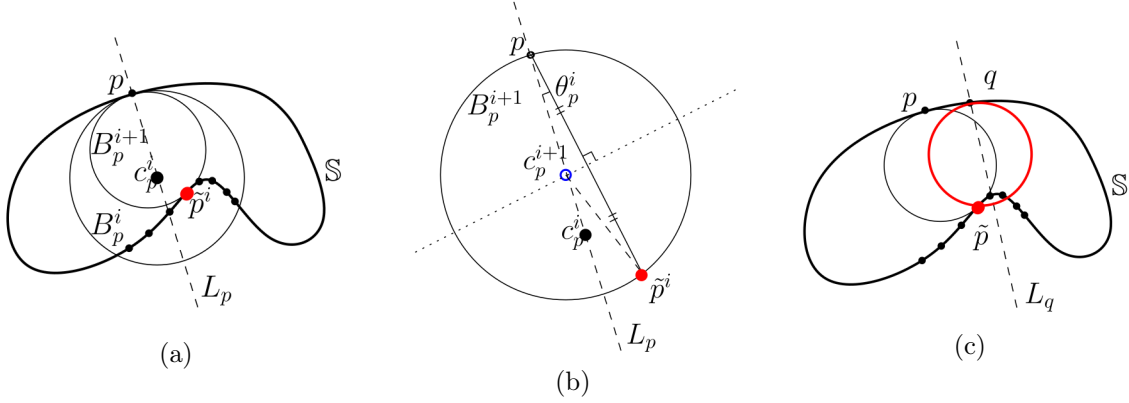


Figure 2.5: An overview of the shrinking ball algorithm by Ma et al. [3]

In listings 2.1, 2.2 and 2.3, pseudocode is provided of a possible implementation of the shrinking ball algorithm. This pseudocode combines the implementation of Ma et al. [3] with the one provided by Peters [29]. The main difference between the pseudocode of Ma et al. and Peters lies in the fact that Ma et al. provide code for the main loop, which generates the set of maximal inscribed circles, where Peters provides code which generates the maximal inscribed circle for a given point. For this reason, the shrinking ball algorithm in this paper has been split up in a main function called ‘ShrinkingBallAlgorithm’ (listing 2.1), which is close to the code provided by Ma et al. [3], and a subfunction ‘ShrinkingBall’ (listing 2.2), which is closer to algorithm by Peters<sup>4</sup>.

<sup>4</sup>It should be noted that there are some details that are glossed over in the pseudocode, for instance what happens if at the start of an iteration,  $\tilde{p}$  is the same point as  $p$ , or the code behind the distance-function, or the NearestNeighbour-algorithm that is used. Nevertheless, it gives a good impression for a basic implementation of the shrinking ball algorithm.

Listing 2.1: Pseudocode for the shrinking ball algorithm by Ma et al. [3]. This function finds the maximum inscribed circle for each point within  $S$ . This algorithm contains the main loop over the point in  $S$ , whereas the actual computation of the maximum inscribed circles occurs in the ShrinkingBall-function, which is expanded on in listing 2.2.

```

/*
 * ShrinkingBallAlgorithm
 *
 * Description:
 * This algorithm generates a set of max circles for each point in S.
 *
 * Input:
 * set <point_t> S = A given set of surface points
 *
 * Output:
 * set <circle_t> circles = A set with the max circle for each point in S
 */

// define a circle based on its center and its radius
typedef pair <point_t, double> circle_t ;

set <circle_t> ShrinkingBallAlgorithm
( set <point_t> S )
{
    // Initialize empty set to store the maximal disks
    set <circle_t> circles ;

    // Initialize p_bar
    point_t p_bar = S[random, but not 0] ;

    for ( int i = 0 ; i < S.size() ; i++ ){
        // Get the current point and its normal vector
        point_t p = S[i];
        vector_t n = p.normal_vector ();

        // Find the point p_bar for the current point
        p_bar = shrinkingBall (p, n, p_bar, S);

        // Get the maximal disk corresponding to p and p_bar
        circle_t max_circle = computeCircle (p, n, p_bar) ;

        // Add the maximal disk to the OutputSet of maximal disks
        circles.pushBack (max_circle) ;
    }

    return circles ;
}

```

Listing 2.2: Pseudocode for the ShrinkingBall-function by Ma et al. [3]. This function return the point  $\tilde{p}$ , which is the point within  $\mathbb{S}$  that intersects the maximum inscribed circle that touches the point  $p$  with normal vector  $\mathbf{n}$ .

```

/*
 * ShrinkingBall
 *
 * Description:
 * This procedure finds the point p_bar, which is the point that is
 * intersected by the maximal inscribed circle touching the point p
 *
 * Input:
 * point_t      p      = The point p, which lies within S
 * vector_t     n      = The normal vector of p
 * point_t      p_bar_0 = The initial p_bar
 * set <point_t> S      = The given set of surface points
 *
 * Output:
 * point_t      p_bar   = The point in S that is intersected by the
 *                        maximal inscribed circle touching p
 */

point_t ShrinkingBall
( point_t      p,
  vector_t     n,
  point_t      p_bar_0,
  set <point_t> S )
{
    // Initialize
    point_t p_bar = p_bar_0 ;
    circle_t circle = ComputeCircle(p, n, p_bar) ;

    point_t c = circle.first ;
    double r = circle.second ;

    // Iterate until the smallest circle is found
    while ( true ){
        point_t p_bar_i = NearestNeighbor (S[except p], c) ;
        circle_t circle_i = ComputeCircle (p, n, p_bar_i) ;

        // Get the center and radius of the circle
        double c_i = circle_i.first () ;
        double r_i = circle_i.second () ;

        if ( abs(r_i - r) < err_conv ){
            break ;
        }

        // update center, radius and p_bar for next iteration
        c = c_i ;
        r = r_i ;
        p_bar = p_bar_i ;
    }

    // Return the final p_bar
    return p_bar ;
}

```

Listing 2.3: Pseudocode for the ComputeCircle-function by Ma et al. [3]. This function finds the circle that touches the point  $p$  with normal vector  $\mathbf{n}$ , and intersects the point  $\tilde{p}$

```

/*
 * ComputeCircle
 *
 * Description:
 * This procedure finds the radius and center of a circle touching the
 * point p with normal vector n, which intersects p_bar
 *
 * Input:
 * point_t      p      = The point that is touched by the circle
 * vector_t     n      = The normal vector of p
 * point_t      p_bar   = The point that is intersected by the circle
 *
 * Output:
 * circle_t     circle  = A pair of the center and radius of the circle
 */

double ComputeCircle
( point_t      p,
  vector_t     n,
  point_t      p_bar )
{
    // Compute the angle from p to p_bar to c
    double theta = arccos((n * (p - p_bar)) / (distance (p, p_bar))) ;

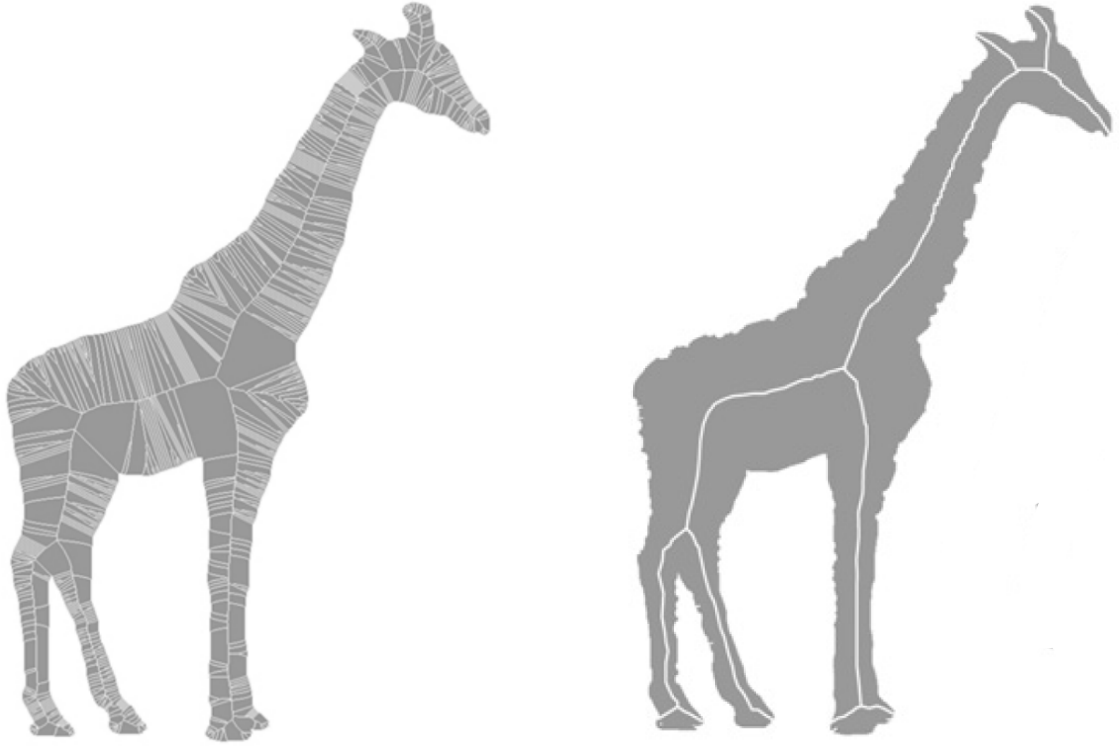
    // Compute the radius and center of the circle
    double r      = (distance (p, p_bar)) / (2 * cos (theta)) ;
    double c      = p + n * r ;

    return make_pair (c, r) ;
}

```

### 2.2.2 Noise reduction

The algorithm demonstrated in listings 2.1-2.3, as proposed by Ma et al. has been designed for point sets that contain little to no noise [3, 29]. It has been well-documented that the skeleton curve is unstable, in the sense that a small amount of noise will lead to large changes in the skeleton curve [27, 25, 30, 29]. These changes will mainly consist of large perturbations, which run between the main axis, and the locations of the boundary perturbations.



(a) The unprocessed skeleton curve of an arbitrary object with a noisy boundary.

(b) An example of a pruned skeleton curve of the same noisy object.

Figure 2.6: A demonstration of the effects of a noisy boundary on the skeleton curve [31].

Many so-called ‘pruning’ methods have been suggested, which intend to prevent scattering of the skeleton curve, either by smoothening the boundary of the original object [27, 31], or by modifying the algorithm by which the skeleton curve is found [32, 30, 29]. For the shrinking ball algorithm proposed by Ma et al., specifically, a noise reduction method has been proposed by Peters [29], which is based on a threshold that is placed on the angle  $\theta_p^i$  as shown in figure 2.5b.

This pruning method does not discard balls that have been affected by noise. Rather, it relies on the observation that for a given noisy ball, the shrinking ball algorithm will have yielded a decent medial ball in a previous iteration. A demonstration of this principle has been offered by Peters [29], which is shown in figure 2.7. Here, the shrinking ball algorithm is applied to a noise-free boundary, and a noisy boundary. For many purposes, including the purpose of the shrinking ball algorithm in this thesis, one might want to find the medial ball corresponding to the noise-free boundary, while using the data provided by the noisy boundary. As figure 2.7 shows, a medial ball similar to the minimum medial ball in figure 2.7a is generated when the algorithm is applied to the noisy case in figure 2.7b. This means that, if the algorithm can be interrupted at the right moment, a decent medial ball can be generated based on the noisy boundary [29].



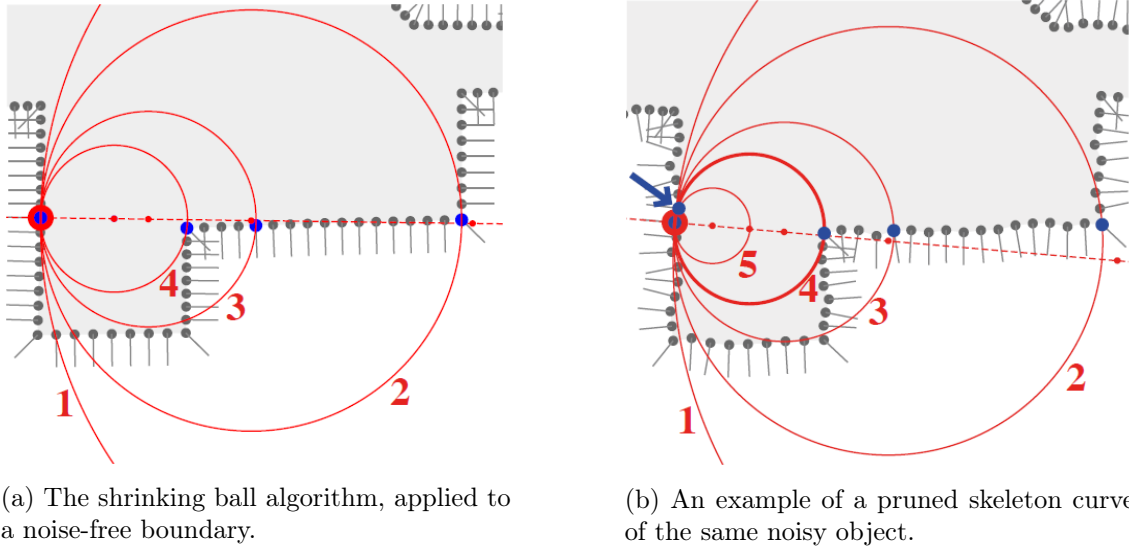


Figure 2.7: A comparison of the results of the shrinking ball algorithm for a noise-free and noisy boundary. Note that, when applied to the noisy boundary, the shrinking ball algorithm has yielded a ball in iteration 4, which is comparable to the noise-free case [29].

To determine at which point the shrinking ball algorithm should be interrupted, Peters [29] suggests to use the separation angle  $\theta_i$ , noting that a large jump in this angle can be observed when the medial ball ‘jumps’ from one side of the point cloud to the other side. When comparing  $\theta_i$  and  $\theta_{i+1}$  in figure 2.8, it becomes clear that the last ball for which  $\theta_i$  is sufficiently large should be considered an end solution to the shrinking ball algorithm. In listing 2.4 an updated version of the shrinking ball algorithm is presented, which includes the noise reduction features proposed by Peters [29].

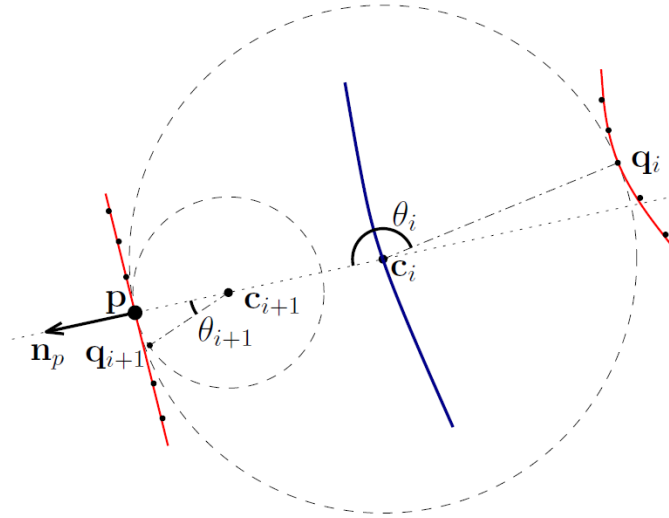


Figure 2.8: A demonstration of the moment at which the medial ball ‘jumps’ from one side of the point cloud to the other. [29]

Listing 2.4: Pseudocode for the enhanced ShrinkingBall-function by Peters [29]. This function has the same function as the ShrinkingBall-function from listing 2.2, but it includes the noise reduction features proposed by Peters.

```

/*
 * ShrinkingBall
 *
 * Description:
 * This procedure finds the point p_bar, which is the point that is
 * intersected by the maximal inscribed circle touching the point p
 *
 * Input:
 * point_t      p      = The point p, which lies within S
 * vector_t     n      = The normal vector of p
 * point_t      p_bar_0 = The initial p_bar
 * set <point_t> S      = The given set of surface points
 *
 * Output:
 * point_t      p_bar   = The point in S that is intersected by the
 *                        maximal inscribed circle touching p
 */
{
    // Initialize
    point_t    p_bar = p_bar_0 ;
    circle_t    circle = ComputeCircle(p, n, p_bar) ;

    point_t    c = circle.first ;
    double     r = circle.second ;

    // Iterate until the smallest circle is found
    while ( true ){
        point_t    p_bar_i = NearestNeighbor (S[except p], c) ;
        circle_t    circle_i = ComputeCircle (p, n, p_bar_i) ;

        // Get the center and radius of the circle
        double     c_i = circle_i.first () ;
        double     r_i = circle_i.second () ;

        if ( abs(r_i - r) < err_conv ){
            break ;
        } else if ( angle(p, c_i, p_bar_i) < angle_crit ){
            break ;
        }

        // update center, radius and p_bar for next iteration
        c = c_i ;
        r = r_i ;
        p_bar = p_bar_i ;
    }

    // Return the final p_bar
    return p_bar ;
}

```

## 2.3 The phantom node method

When modeling the failure of FRPs due to crack development, it is not a trivial task to choose a suitable damage model. Continuous models fail to properly account for the orientation of the fibers in the composite material when modeling crack formation, and thus a discontinuous model is preferred [15]. Discontinuous methods that explicitly model the crack formation, however, often require predefined crack locations. For the complex failure mechanisms of FRPs, the location and orientation of these interface elements might be very difficult to predict [16].

In order to overcome both these issues, the phantom node method, as proposed by Hansbo and Hansbo [33, 5], is used in this thesis. This method allows for crack growth in arbitrary locations that do not need to be specified beforehand, while still creating a discontinuous model of crack formation, thereby avoiding both issues mentioned before [16, 34].

### 2.3.1 Element and node definitions

The fundamental idea behind the phantom node method is that a crack in the material can be modeled by adding so-called ‘phantom nodes’ and ‘phantom elements’ to the mesh. To model a discontinuity in an element  $K$ , it is split into two elements  $K_1$  and  $K_2$ , such that  $K_1 \cup K_2 = K$  and  $K_1 \cap K_2 = \emptyset$ . The displacement fields of both  $K_1$  and  $K_2$  can be expressed using duplicated nodes on the standard element locations [5], with the standard degrees of freedom. This means that if an element is intersected by only one discontinuity, it can be replaced by two copies of that element [5, 35, 36]. For the each of nodes of the element, one copy needs to be made, which supplements the original nodes. Each phantom element is connected to the original nodes on their side of the discontinuity, and to the phantom nodes on the other side of the discontinuity, as shown in figure 2.9.

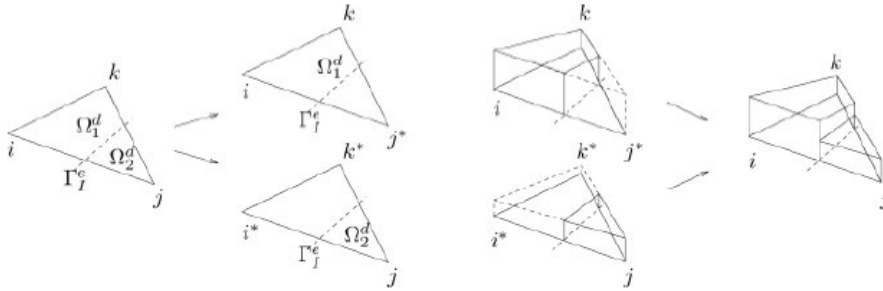


Figure 2.9: A visualization of the connectivity between two phantom elements and the original nodes  $(i, j, k)$  and the phantom nodes  $(i^*, j^*, k^*)$  [35]

An additional point that should be noted is the fact that it is possible in case of branching that multiple discontinuities appear in a single element. Often, this is dealt with by simply removing the element at hand [36, 37], but in this thesis, a part of the element will still be kept active, as shown in figures 2.10c and 2.11c. Rather than completely discarding the element containing the junction, a part of the element is kept active, by taking a shortcut straight through the element. Although this approach will still produce a lower stiffness than if the full element is accounted for, it does produce a better approximation than removing the element, without the need to define new shape functions.

Furthermore, attention needs to be paid at the element at the crack tip. Although approaches that would allow for a partial discontinuity in a crack tip element, do exist [37,

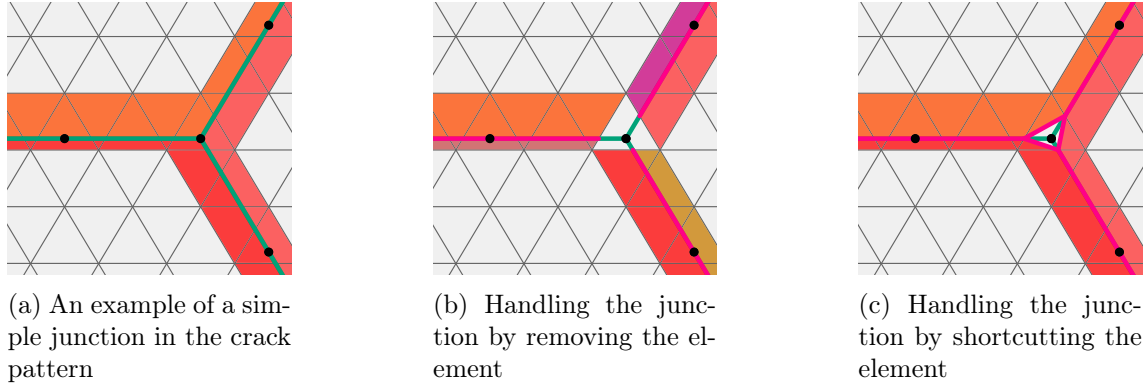


Figure 2.10: A comparison of the handling of a simple junction in the crack pattern by removing the original element, and by taking a shortcut through the element.

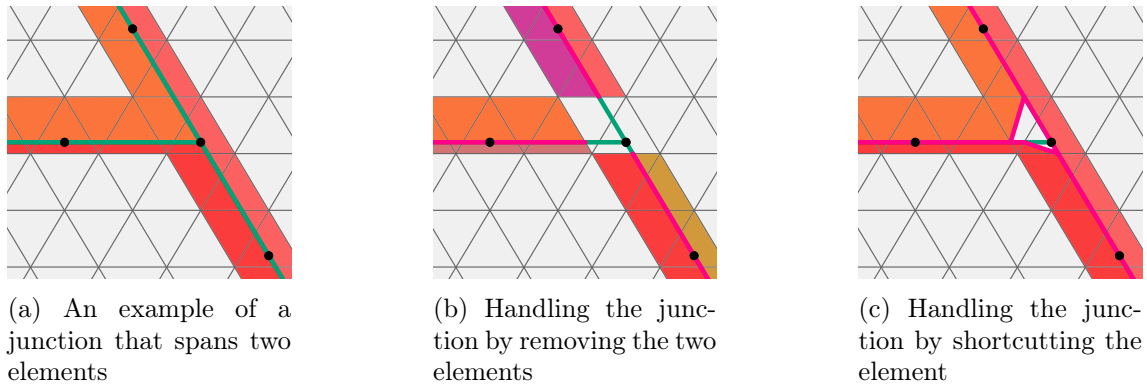


Figure 2.11: A comparison of the handling of a junction that spans two elements by removing the original element, and by taking a shortcut through the element.

38], the most common and simple approach is to force the crack tip to be located on an element edge [35, 36, 16, 39]. This implies that the crack tip element<sup>5</sup> will not be replaced by two phantom elements, and none of its nodes will be replaced by phantom nodes, as shown in figure 2.12.

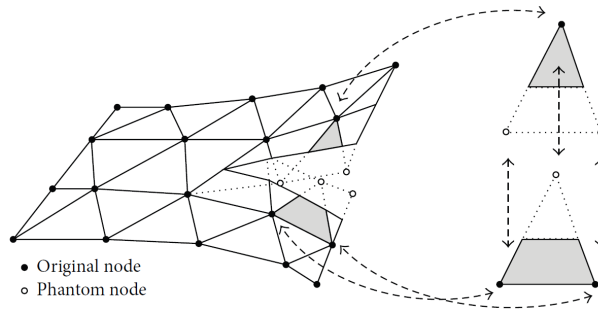


Figure 2.12: A visualization of the phantom elements at the crack tip. [40]

<sup>5</sup>Note that the term ‘crack tip element’ still refers to the same element, even though it is no longer intersected by the discontinuity

### 2.3.2 Integration scheme

To account for the difference in geometry of the phantom elements, it is not needed to define new shape functions [35, 36, 16], although it is necessary to modify the integration scheme of the intersected elements. This is due to the fact that the displacement jump splits a triangular element into a triangle and a quadrilateral element. This quadrilateral can be handled by triangulating it into two [35, 36, 37] or three [16] subdomains, each of which have their own integration points. To these triangular subdomains, the original integration scheme can be applied. In order to apply traction forces, it is also necessary to add two integration points to the location of the discontinuity [35, 16]. An visualization of this integration scheme is given in figure 2.13.

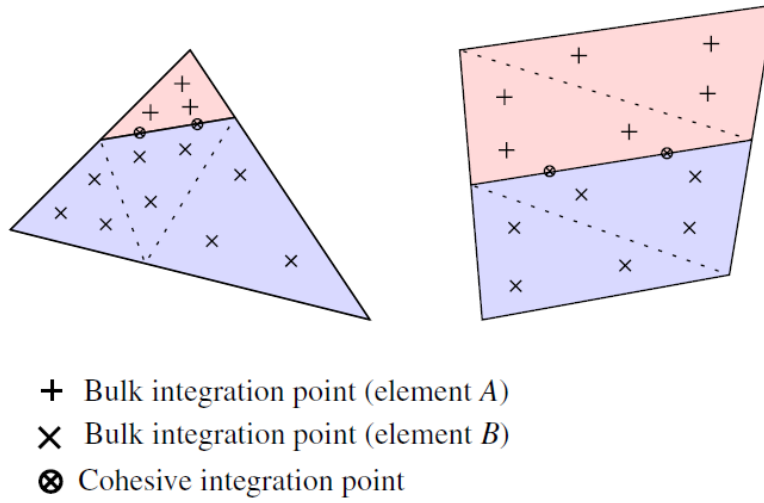


Figure 2.13: The subdomain triangulation of intersected elements and their integration points. [16]

## Implementation

The implementation of the TLS\_V2 heavily relies on the implementation of TLS\_V1 designed by Van der Meer [7] and Mororó [17], which was used to study and model the damage development of FRP materials. It was developed using Jem and Jive, two C++ libraries developed by Dynaflow Research Group, which provide a framework for Finite Element Analysis (FEA). Jem provides the most fundamental building blocks of the Jem/Jive-framework, such as Vectors, Arrays, Hashmaps, et cetera. Jive, on the other hand, handles the FEA itself. It allows for the definition of elements, nodes and degrees of freedom, as well as the creation of stiffness matrices and force vectors, and finding the corresponding solution.

Their routine consists of three so-called ‘chains’: the ‘level set update’ chain, the ‘equilibrium solution’ chain and the ‘front evolution’ chain. Each chain calls a series of modules, which in turn perform certain actions on the models that are defined in the properties file. This setup allows for a lot of flexibility when defining a FEA. An overview of the program can be found in figure 3.1. These three chains reflect the setup of the TLS method explained in section 2.1, where the details of the steps taken in each of these chains can be found.

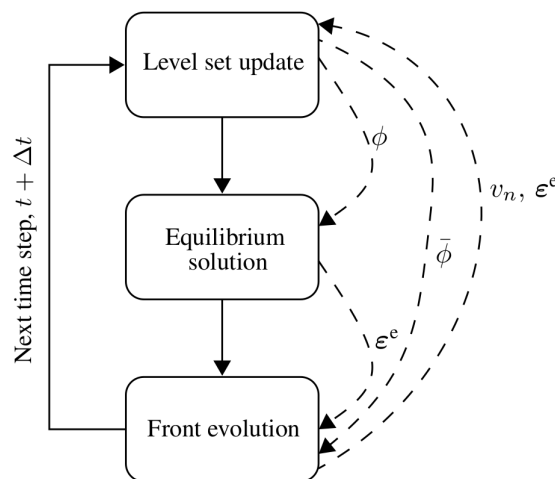


Figure 3.1: A general overview of the Thick Level Set model designed by Van der Meer and Mororó

As explained in chapter 1, this thesis mainly focuses on the determination of a skeleton curve, and how the displacement jump at this skeleton curve can be modeled. Since the skeleton curve can only be determined after the full iso-0 curve is known, it

can only be determined after the ‘level set update’ chain. On the other hand, the mesh does need to be modified before the mechanical problem is analyzed in the ‘equilibrium solution’ chain. This means that the Skeletonizer function, which determines the location of the skeleton curve based on the level set field, should be executed at the end of the ‘level set update’ chain. The PhantomNodeModel, which updates the mesh using the results from the Skeletonizer function will be executed right at the start of the ‘equilibrium solution’ chain. These two functions will be discussed in sections 3.1 and 3.2, respectively.

### 3.1 Skeletonizer

In order to find the skeleton curve based on the level set field, it is convenient to first find the iso-0 curve, and store it in a convenient format. To do this, the functions `makeLineStrings`, `makeClosedLoops` and `makeGroupedLoops` are created, which are described in sections 3.1.1, 3.1.2 and 3.1.3, respectively. The `makeLineStrings` function determines the location of the iso-0 curve, and stores the result in a `set` of `lstring_ts`. The `makeClosedLoops` and `makeGroupedLoops` functions then group the items in this `set` of `lstring_ts` in such a way that each group corresponds to a closed off damage zone.

After the iso-0 curve is known for each damage zone, the skeleton curve can be determined using the shrinking ball algorithm as described in section 2.2.1. Additionally, the atoms returned by the shrinking ball algorithm will have to be connected in a manner that corresponds to the connectivity of the skeleton curve. This will be done in the `makeAtomGraph` function, which will be explained in section 3.1.4. The resulting `atomGraph` contains the skeleton curve, stored in a `graph_t` format. It stores the atoms as vertices, and connecting segments between the atoms as edges, which captures the full connectivity of the skeleton curve. The `atomGraph` will then be organized using the `makeCrackPattern` function as described in section 3.1.5, in order to more conveniently handle the `atomGraph` in later procedures.

Once the skeleton curve is fully known, it can be mapped onto the mesh, which is done by the `makeElemGraph` function. This function takes the `atomGraph` as input, and returns an `elemGraph`, which stores elements as vertices and the intersections of the skeleton curve with the element edges as graph edges. In order to handle certain edge cases, the `makeShortCuts` function is used, which slightly modifies the `elemGraph` at corners and junctions. These two functions will be explained in sections 3.1.6 and 3.1.7.

Lastly, the `makePhantomNodes` function is called, which determines based on the `elemGraph` which elements and nodes should be replaced by phantom elements and nodes. This is the last step of the Skeletonizer, after which the `PhantomNodeModel` will take over to actually perform the modifications to the mesh that are prescribed by the Skeletonizer.

### 3.1.1 makeLinestrings

In order to find the linestring describing the iso-0 curve, it is necessary to first consider what possible shapes this linestring could have. In figure 3.2, the three possible kinds of iso-0 curves are possible. The first, and most straightforward kind is the closed linestring, which forms a closed, non-intersecting loop distinguishing the positive and negative parts of the level set field, which are inside and outside the loop, respectively.

There are, however, three non-trivial cases that require special consideration, which are shown in figure 3.2:

1. Open linestrings – damage fronts that hit the boundary of the material in a single location will form an open linestring. Although the damaged zone is still enclosed by a single linestring, they still need special attention, since they cannot be directly distinguished from disconnected linestrings.
2. Disconnected linestrings – if a damage fronts hits the boundary of the material in multiple distinct locations, it creates multiple disconnected linestring. These linestrings will have to be combined into a multilinestring in order to store the full boundary of the damaged zone in a single object.
3. Enclosed linestrings – it is also possible for a damaged zone to contain an undamaged zone within it. This means that it might be necessary to combine multiple closed linestrings into a multilinestring to fully describe the boundary of the damaged zone.

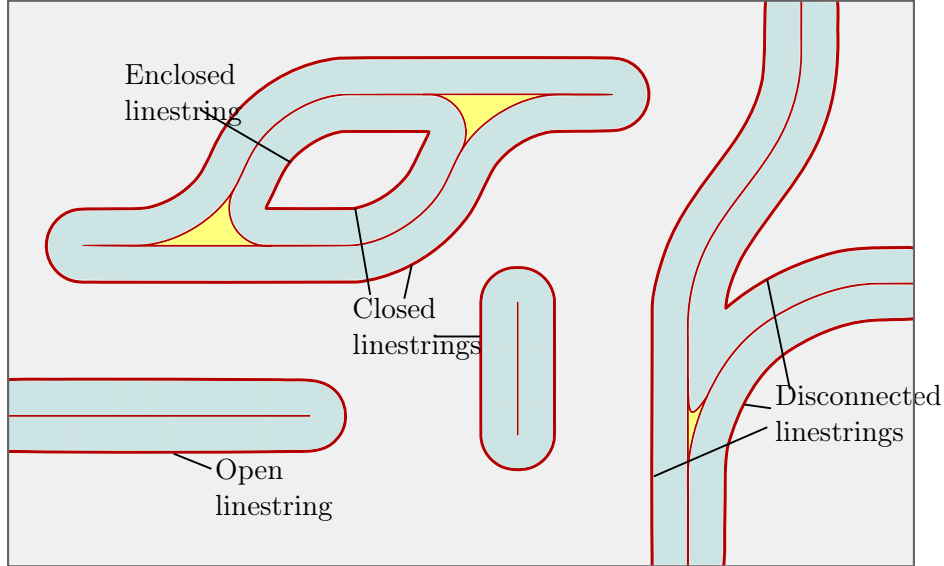


Figure 3.2: A demonstration of the differences between closed linestrings, open linestrings, disconnected linestrings and enclosed linestrings

To tackle these issues, three separate procedures will be used. First, the `makeLinestrings` function generates a set of linestrings describing the iso-0 curve. Each entry in this set will consist of three items: `iso0` is a linestring object which describes a single connected part of the iso-0 curve, which can be either a closed linestring, an open linestring, or a disconnected linestring. The elements `elemFirst` and `elemLast` correspond to the elements at the beginning and end of each linestring, respectively. Closed linestrings can easily be distinguished from open and disconnected linestrings, by checking whether `elemFirst` and `elemLast` are the same element.



Possible open or disconnected linestrings are dealt with by the `makeClosedLoops` function, which links disconnected linestrings together by walking along the material boundary. Afterwards, the `makeGroupedLoops` function is called, which merges any enclosed linestring with the multilinestring that encircles it. These two functions are called as sub-routines of the `makeLinestrings` function, the pseudocode for which can be found in listing 3.1.

Listing 3.1: Pseudocode for the `makeLinestrings` function. It takes the set of elements describing the mesh as input, and returns a set of multilinestrings, each of which corresponds to a closed off damage zone.

```

set <mlstring_t> makeLinestrings
( set <element_t>      elems )
{
    // Get the set of indices of elements intersected by iso=0 curve
    set <idx_t> isoIdx = getIso0Elems (elems) ;

    // Keep track of which elements have been treated
    set <bool> elemsToDo (elems.size ()) ;
    elemsToDo = false ;
    elemsToDo[isoIdx] = true ;

    // Initialize the set of linestrings describing the iso=0 curve
    set <pair <lstring_t, pair<element_t, element_t>>> iso_0

    while ( not elemsToDo.allTrue () ){
        // Find the first element that has not been done yet
        idx_t ielem = elemsToDo.firstTrue () ;

        // Initialize the start of the linestring
        segment_t segment = elems[ielem].getIntersection () ;

        lstring_t linestring ;
        linestring.pushBack (segment) ;

        // Set the first and last elements
        element_t elemFirst = elems[ielem] ;
        element_t elemLast = elems[ielem]

        // Loop forward to add segments to the back of the linestring
        addSegmentsForward (iso_0, elemFirst, elemLast) ;

        // Check if a closed loop has been found
        if ( elemFirst != elemLast ){
            // If not, loop backwards to add segments to the linestring
            addSegmentsBackward (iso_0, elemFirst, elemLast) ;
        }

        // Add the linestring to the set of linestrings
        iso_0.pushBack (makePair(linestring, makePair(elemFirst, elemLast))) ;
    }

    // Connect the open and disconnected linestrings in iso_0
    set <mlstring_t> isoClose = makeClosedLoops (iso_0) ;

    // Group the loops to handle undamaged zones within damaged zones
    set <mlstring_t> isoGroup = makeGroupedLoops (isoClose) ;

    // return the groups of multilinestrings
    return isoGroup ;
}

```

The `makeLinestrings` function heavily relies on the `addSegmentsForward` function, which is summarized in listing 3.2. It operates by looping over the edges of the last element, and checking for each edge if the level set field value goes from positive to negative at that edge. If so, the next element can be found on the other side of that edge, and `elemLast` is updated. The algorithm terminates for closed linestrings when it arrives back at `elemFirst`, or for open linestrings if no new elements are found. It should be noted that this method only works if all element nodes are stored in a certain rotational direction<sup>1</sup>. If this is the case, however, the resulting linestring will be oriented in the same rotational direction if it encloses a damaged zone, and oriented in the opposite direction if it is contained within a damaged zone.

The `addSegmentsBackward` works in the same way, but looks at `elemFirst` instead of `elemLast`, and checks if the level set field value goes from negative to positive instead of vice versa. The pseudocode for this function can be found in appendix A in listing A.2, along with the pseudocode for the `getNextElements` function in listing A.3 and the `getNodeNegPos` and `getNodePosNeg` functions in listing A.4

Listing 3.2: Pseudocode for the `addSegmentsForward` function. It adds segments to the back of the `iso0` linestring, until either the linestring loops back onto itself, or no new segments are found. The input elements `elemFirst` and `elemLast` are used to keep track of the begin and endpoint of the linestring, and updated accordingly.

```

void addSegmentsForward
( lstring_t          iso0 ,
  element_t         elemFirst
  element_t         elemLast )
{
  // Iterate forward until no elements are found, or the loop is closed
  while ( true ){
    // Get the nodes where the level set field goes from >= 0 to < 0
    pair <node_t, node_t> nodeNegPos = getNodeNegPos (elem_last) ;
    node_t pos = nodeNegPos.first ;
    node_t neg = nodeNegPos.second ;

    // Find the neighboring element
    set <element_t> elemsNext = getNextElements (elemLast , pos , neg) ;

    // [Verify that elemsNext contains no more than 1 element]

    if ( elemsNext.size() == 0 ){
      // The end of the linestring is found, so exit the loop
      break ;
    } else {
      // Add the segment to the linestring
      segment_t segment = elemsNext[0].getIntersection () ;
      iso0.pushBack (segment) ;

      // Break if the linestring has been closed
      if ( elemsNext[0] == elemFirst ) break ;

      // Update the last element
      elemLast = elemsNext[0] ;
    }
  }
}

```

<sup>1</sup>i.e. either clockwise or counterclockwise

### 3.1.2 makeClosedLoops

Although it is possible to treat the open linestrings and disconnected linestrings separately, it is not straightforward to distinguish beforehand whether a linestring encloses a damaged zone on its own, or whether multiple linestrings are required to describe iso-0 curve of a damaged zone. In order to determine if a linestring is open or disconnected, an ad hoc solution would be required, which distinguishes these two cases based on, for instance, the distance between its first and last element. For most applications, such a solution might suffice, but more robust implementation can be created by not making this distinction at all. This approach will be described below, although it should be noted that due to time constraints, it was not possible to fully implement this procedure. The procedure that will be explained in this section is currently only applied to disconnected chains, although in principle, it should work for open chains as well.

The fundamental idea behind the `makeClosedLoops` function is that the next linestring can be found by traversing along the boundary of the material from the endpoint of the current linestring. Since the last element is located at the material boundary, and is intersected by the iso-0 curve, one of the boundary nodes must have a negative level set value, and one must have a positive value. By iterating over the boundary in the direction of the positive node, until a negative node is found, the first element belonging to the next linestring can be found. For open linestrings, this next linestring is the same linestring, and the process can be terminated directly. For disconnected linestrings, on the other hand, the process has to be repeated multiple times for disconnected linestrings, until the first element of the initial linestring has been found. The pseudocode for the `makeClosedLoops` function can be found in listing 3.3.

### 3.1.3 makeGroupedLoops

Since all multilinestrings returned by the `makeClosedLoops` function represent a closed boundary of the damaged zone, they can be treated as closed linestrings by the `makeGroupedLoops` function. Accordingly, the open multilinestrings are implicitly converted<sup>2</sup> to closed linestrings in the pseudocode of this function, shown in listing 3.4

The method by which the `makeGroupedLoops` works is relatively straightforward. It checks for each multilinestring if it is contained within another multilinestring. If this is the case, it is skipped. If not, it must be the outermost boundary of the damaged zone. Another loop over the set of linestrings is then used to gather all multilinestrings that are contained within the outermost boundary. This way, all multilinestrings belonging to a single damage zone will always be grouped together. The pseudocode for this function can be found in listing 3.4

---

<sup>2</sup>it should be noted that this implicit conversion is not actually possible. It will be treated as if this is possible, however, since it has been shown in section 3.1.2 that all multilinestrings do form a closed circuit.

Listing 3.3: Pseudocode for the makeClosedLoops function. This function takes the set of linestrings returned by the makeLinestrings function, as well as their elemFirst and elemLast elements, as input, and returns a set of multilinestring. Each multilinestring corresponds to a closed loop of the iso-0 curve.

```

set <mlstring_t> makeClosedLoops

( set <lstring_t>    iso0 ,
  set <element_t>    elemFirst ,
  set <element_t>    elemLast )

{
  // Initialize the set of multilinestrings
  set <mlstring_t> isoClosed ;

  // Keep track of which linestring have been treated
  set <bool> lineDone (iso0.size()) ;
  lineDone = false ;

  // Loop over the set of linestrings iso0
  for ( idx_t i = 0 ; i < iso0.size() ; i++ ){
    // Check if the linestring is a closed loop
    if ( elemFirst[i] == elemLast[i] ){
      // If so, add it to the iso_close straightaway
      isoClosed.pushBack (iso0[i])
      lineDone[i] = true ;
    } else {
      // Initialize a multilinestring, and get its last element
      mlstring_t multiLineString ;
      element_t currElemLast = elemLast[i] ;

      // Iterate from elemLast[i] until elemFirst[i] has been found again
      while ( true ){
        // Get the element on the other side of the damage zone
        nextElemFirst = getOppositeElement (currElemLast) ;

        // Break the loop if the loop is completed
        if ( nextElemFirst == elemFirst[i] ) break ;

        // Find this element in the elemFirst set
        for ( idx_t j = 0 ; j < elemFirst.size() ; j++ ){
          if ( elemFirst[j] == nextElemFirst ){
            currElemLast = elemLast[j] ;

            // Add the next iso_0 linestring to the multilinestring
            multiLineString.pushBack (iso0[j]) ;
            lineDone[j] = true ;
          }
        }
      }
    }
  }

  // Add the multilinestring to the isoClosed
  isoClosed.pushBack (multiLineString) ;
}

// Return the set of closed multilinestrings
return isoClosed
}

```

Listing 3.4: Pseudocode for the makeGroupedLoops function. This function takes the set of multilinestrings returned by the makeClosedLoops function as input, and groups all multilinestrings that are encircled by other multilinestrings together.

```

set <mlstring_t> makeGroupedLoops

( set <mlstring_t> isoClosed )

{
    // Initialize isoGroup
    set <mlstring_t> isoGroup

    set <bool> mlstringDone (iso_close.size ()) ;
    mlstringDone = false ;

    // Loop over the set of closed multilinestrings
    for ( idx_t i = 0 ; i < isoClosed.size () ; i++ ){
        // Check if the current mlstring lies inside another loop
        bool insideLoop = false ;
        for ( idx_t j = 0 ; j < isoClosed.size () ; j++ ){
            // Skip if it concerns itself
            if ( i == j ) continue ;

            // Check if mlstring i lies within mlstring j
            if ( isoClosed[i].within (isoClosed[j]) ){
                insideLoop = true ;
                break ;
            }
        }

        // Skip if mlstring i lies inside another mlstring
        if ( insideLoop == true ) continue ;

        // Gather all mlstrings within mlstring i
        mlstring_t mlGroup ;

        mlGroup.pushBack (isoClosed[i]) ;

        for ( idx_t j = 0 ; j < iso_close.size () ; j++ ){
            // Skip if it concerns itself
            if ( i == j ) continue ;

            // Check if mlstring j lies within mlstring i
            if ( isoClose[j].within (isoClose[i]) ){

                // If so, add mlstring j to the group
                mlGroup.pushBack (isoClose[j]) ;
            }
        }

        // Add the grouped multilinestring to isoGroup
        isoGroup.pushBack (mlGroup) ;
    }

    // Return the grouped multilinestrings
    return isoGroup ;
}

```

### 3.1.4 makeAtomGraph

In order to determine the location skeleton curve, the shrinking ball algorithm as proposed by Ma et al. [3] will be used, including the noise reduction measures proposed by Peters [29]. The procedure in the implementation is very similar to listing 2.1. The main difference lies in the way information is stored and processed. Rather than simply storing the coordinates of each of the maximal inscribed circles in a list or a vector, they are stored in a graph, making use the Boost Graph Library. This library includes minimum spanning tree algorithms, as well as iterators over a graph, both of which will be useful in later parts of the procedure.

For each atom, the index, which contains a unique `idx_t` index, and the coords, which contains a `point_t` for its coordinates, are stored in the graph. Furthermore, the Boost Graph Library requires the possibility to store the predecessor of each atom as an `idx_t`, and a `double` for the distance before it can run Prim's algorithm and create a shortest spanning tree of the atom graph. The atoms will be stored as vertices of the graph, and all aforementioned properties will be assigned to the vertices of the graph. The edges, then, will also be assigned a unique `idx_t` index, index, and contain a double length to store its length. This last property will also be used as a weightmap in Prim's algorithm.

In listing 3.5, the general structure of the `makeAtomGraph` function is presented. Some details are simplified or ignored; most significantly the conversion of the `isoGroup` variable from a `multilinestring` to a set of points with normal vectors. However, since the orientation of the linestrings has been handled carefully in the `makeLineStrings` procedure, the direction of an inward-pointing normal vector can be straightforwardly found based on the orientation of each segment of the linestring.

Since Prim's algorithm will always yield a tree<sup>3</sup>, the `makeAtomGraph` function is guaranteed to produce a set of trees that does not contain any cycles [4]. This property can be used to more easily navigate over the `atomGraph`, which will be used to loop over the skeleton curve in the `makeCrackPattern` and `makeElemGraph` functions.

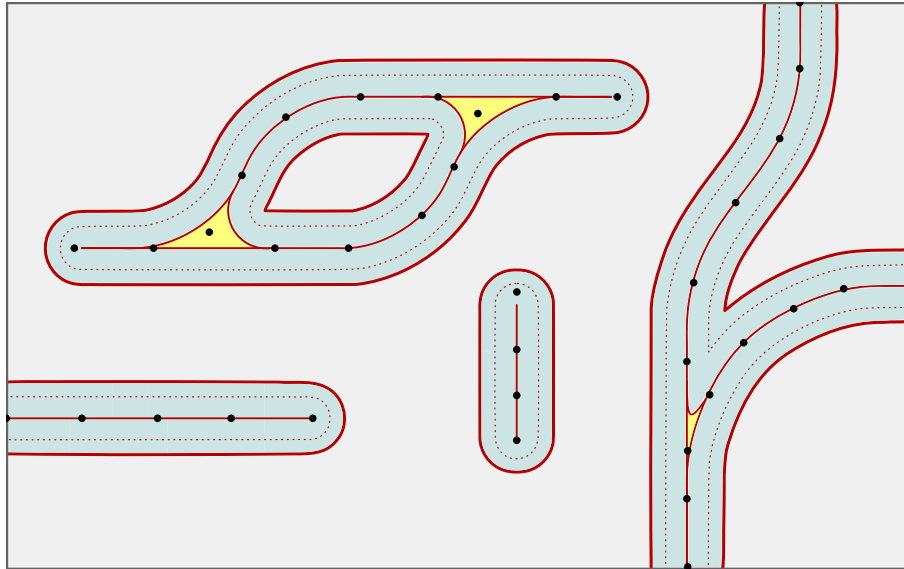


Figure 3.3: The atom locations and connections describing of the skeleton curve.

<sup>3</sup>i.e. an undirected graph which is acyclic and connected

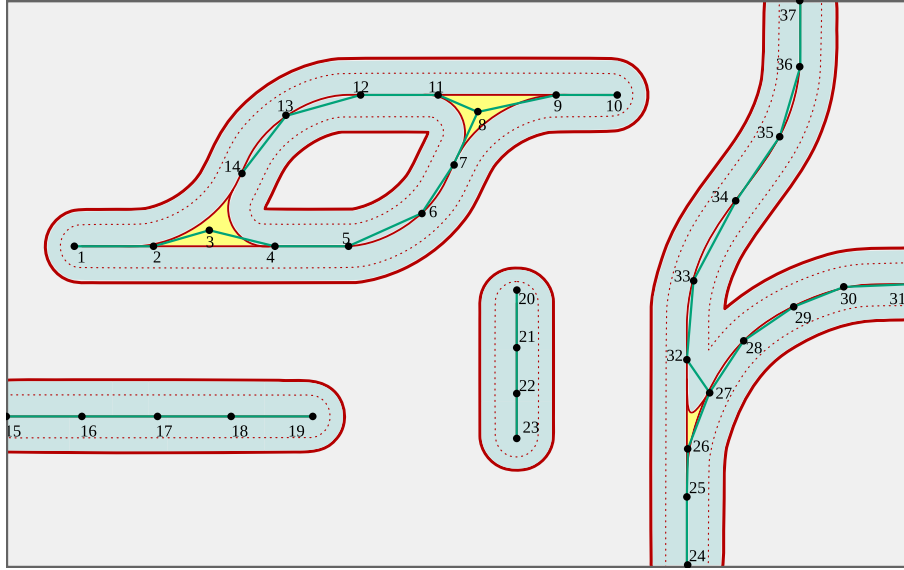


Figure 3.4: The atomGraph that would be created by the makeAtomGraph function.

Listing 3.5: Pseudocode for the makeAtomGraph function. This function transforms takes the set of multilistings returned by the makeLinestrings function, and produces a graph containing the centers of the maximal inscribed disks as vertices, and their minimum spanning graph as edges.

```

graph_t makeAtomGraph
(
    set<mlstring_t> isoGroup
)
{
    // Initialize the atom graph for the whole mesh
    graph_t atomGraph ;

    // Loop over each group of iso=0 points
    for ( idx_t i = 0 ; i < isoGroup.size () ; i++ ){

        // Apply the shrinking ball algorithm to each group of linestrings
        set<circle_t> maxDisks = ShrinkingBallAlgorithm (isoGroup[i]) ;

        // Create an individual graph for each group of linestrings
        graph_t partialAtomGraph ;

        // Add the centers of the maximal disks as vertices to the atomGraph
        partialAtomGraph.addVertices (maxDisks.first) ;

        // Apply Prim's algorithm to the atom graph of the group
        partialAtomGraph.primMinimumSpanningTree () ;

        // Add the individual graph to the graph for the whole mesh
        atomGraph.insertGraph (partialAtomGraph) ;
    }

    // Return the graph for the whole mesh
    return atomGraph ;
}

```

Crack Index	Atoms							
1	1	2	3					
2	3	4	5	6	7	8		
3	3	14	13	12	11	8		
4	8	9	10					
5	15	16	17	18	19			
6	20	21	22	23				
7	24	25	26	27				
8	27	28	29	30	31			
9	27	32	33	34	35	36	37	

Table 3.1: The crack pattern that corresponds to the atomGraph shown figure 3.4

### 3.1.5 makeCrackPattern

Once the atomGraph has been determined, it can be said that the skeleton curve is fully known, since the connectivity and coordinates of all atoms skeleton curve has been determined. However, it is useful to organize the atomGraph in a way that allows for a more convenient translation from the atomGraph to the elementGraph. To achieve this, the makeCrackPattern function has been introduced. This function takes the atomGraph as input and returns as output a set of sets of atoms, each of which corresponds to a crack. In more precise terms, each set of atoms corresponds to a series of vertices with exactly two edges, except for the first and last atom in the set.

As an example, the crackPattern that would be created for the atomGraph in figure 3.4 by the makeCrackPattern function is given in table 3.1.

The procedure accomplishes this by looping over the vertices of the atomGraph, and checking for each vertex if it has either 1 or more than 3 adjacent vertices. If this is the case, an endpoint or junction has been found, which can be seen as the starting point of the crack. For each outgoing edge of this vertex, the algorithm checks if has already been included in the crackPattern, using the edgeInPattern function, which can be found in listing A.5. If the edge has not yet been included, the procedure creates starts iterating over the atomGraph, until another endpoint or junction is found. All vertices that are encountered during this process are added to the crackPattern, until all vertices have been treated. The pseudocode for this function is given in listing 3.6.



Listing 3.6: Pseudocode for the makeCrackPattern function. This function takes the atomGraph returned by the makeAtomGraph function as input, and organizes the vertices as demonstrated in figure 3.4 and table 3.1.

```

set <set <idx_t>> makeCrackPattern

( graph_t          atomGraph )

{
  // Initialize the crackPattern
  set <set <idx_t>> crackPattern ;

  // Loop over the vertices of the atomGraph
  for ( idx_t iv = 0 ; iv < atomGraph.vertices.size () ; iv++ ){

    vertex_t atom = atomGraph.vertices[iv] ;

    // Check if the vertex is a junction or an endpoint
    if ( atom.adjacentVertices.size () != 2 ){

      // Loop over the adjacent vertices of the atom
      for ( idx_t ia = 0 ; ia < atom.adjacentVertices.size () ; ia++ ){

        edge_t segment = atomGraph.edges[atom, atom.adjacentVertices[ia]] ;

        // Check if the edge already exists in the crackPattern
        bool edge_found = edgeInPattern (segment, crackPattern) ;

        if ( not edge_found ){
          // If not, Initialize a new crack
          set <idx_t> crack ;
          vertex_t prevAtom = atom ;
          vertex_t currAtom = atom.adjacentVertices[ia] ;

          // Add the current atom to the crack
          crack.pushBack (atom.index) ;

          // Iterate until a junction or endpoint is found
          while ( true ){
            // Add the current atom to the crack
            crack.pushBack (currAtom.index) ;

            // Break the loop if the current atom is a junction or endpoint
            if ( currAtom.adjacentVertices.size () != 2 ) break ;

            // Get the atom's adjacent vertices
            set <vertex_t> adj = currAtom.adjacentVertices ;

            // Find the next atom
            vertex_t nextAtom = (adj[0] == prevAtom) ? adj[1] : adj[0] ;

            prevAtom = currAtom ;
            currAtom = nextAtom ;
          }

          // Add the crack to the crackPattern
          crackPattern.pushBack (crack) ;
        }
      }
    }
  }
  // Return the crackPattern
  return crackPattern ;
}

```

### 3.1.6 makeElemGraph

With the `atomGraph` fully known, and organized in the `crackPattern`, the skeleton curve can now be mapped onto the mesh. To accomplish this, again a graph from the Boost Graph Library will be used, which will be called `elemGraph`. In this graph, the vertices will correspond to elements, whereas the edges correspond to the intersection points of the skeleton curve with the element edges. This method of data storage makes it easier to determine for any given element which intersection points it contains, and to which elements it connects. After all, for any given vertex in a graph, its edges and adjacent vertices are known.

Similarly to the `atomGraph`, properties belonging to the elements can be stored as vertex properties, and properties belonging to the intersection points can be stored as edge properties. This will be used to store the index of each element as an `idx_t`. A `set <segment_t>` will be used to store a number of segments corresponding to the location of the skeleton curve in each element. Also, a `set <idx_t>` called `atoms` will be used to store an atom that falls within the element. This set can have a size of either zero or one, depending on if the element in question contains an atom or not. It cannot be larger than one, since an element cannot contain multiple atoms, due to the restrictions described in section 3.1.4. To store the locations of the intersection points of the skeleton curve with the element edges, a `point_t` is stored for each edge of the `elemGraph`, containing the coordinates of the intersection point.

An additional complication for the `elemGraph` lies in the fact that, unlike the `atomGraph`, it might contain cycles at locations where the `atomGraph` contains a corner or a junction. After all, at any point where the `atomGraph` bends, it might intersect the same element edge twice. Since these intersection points are stored as edges in the `elemGraph`, they produce two edges that both connect the same pair of vertices. To handle these cycles, the `makeShortCuts` function will be introduced in section 3.1.7, which will handle all elements that contain an atom. Initially, the `elemGraph` will only trace the straight segments of the `atomGraph` that run through elements that do not contain an atom. This will be the output of the `makeElemGraph` function. Afterwards, the `elemGraph` will be modified by the `makeShortCuts` function, which produces the graph shown in figure 3.5.

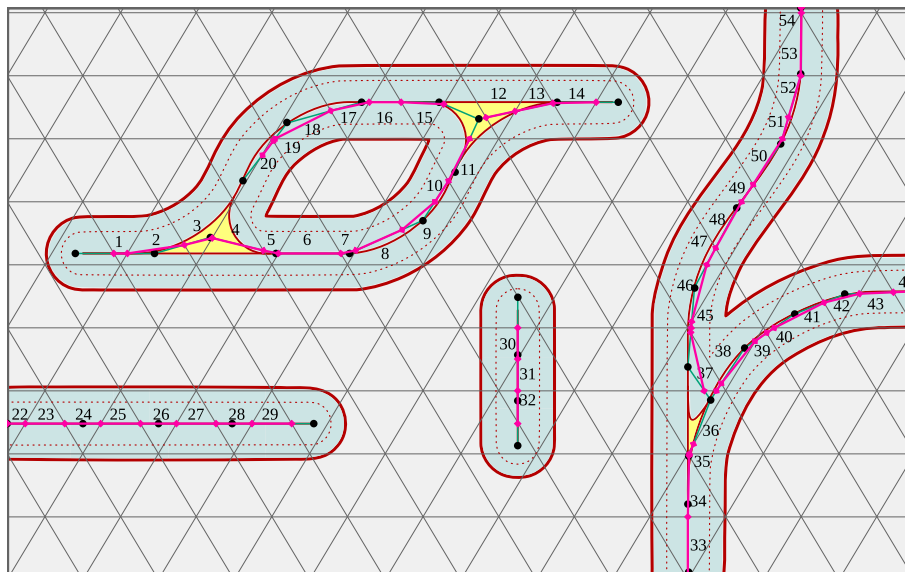


Figure 3.5: The `elemGraph` that would be created by the `makeElemGraph` function, based on the `atomGraph` shown in figure 3.3

Listing 3.7: Pseudocode for the makeElemGraph function. This function takes the atomGraph and crackPattern as input, and returns a graph describing the skeleton curve. This graph has elements as vertices and intersection points as edges.

```

graph_t makeElemGraph

( graph_t      atomGraph ,
  set <set <idx_t>> crackPattern )

{
  // Initialize the elemGraph
  graph_t elemGraph ;

  // Loop over the crackPattern
  for ( idx_t i = 0 ; i < crackPattern.size () ; i++ ){
    // Initialize crackIntxns to keep track of the crack's intersections
    set <pair <point_t, idx_t>> crackIntxns ;
    set <idx_t> crack = crackPattern[i] ;

    // Loop over the crack
    for ( idx_t j = 0 ; j < crack.size () - 1 ; j++ ){
      // Get the vertices from the atomGraph and the segment between them
      point_t sourceCoords = atomGraph.vertices[crack[j]].coords ;
      point_t targetCoords = atomGraph.vertices[crack[j+1]].coords ;
      segment_t crackEdge (sourceCoords, targetCoords) ;

      // Initialize edgeIntxns to keep track of the segment's intersections
      set <pair <point_t, element_t>> edgeIntxns ;

      for ( idx_t ie = 0 ; ie < elems.size () ; ie++ ){
        if ( crackEdge.intersects (elems[ie]) ){
          // Add the element to the elemGraph if it doesn't exist yet
          if ( not elemGraph.vertexExists (ie) ) elemGraph.addVertex (ie) ;

          // Get the intersection of the crack edge with each element
          segment_t elemIntxn = elems[ie].intersection (crackEdge) ;

          // Add the intxns to edgeIntxns and the segment to elemGraph
          edgeIntxns.pushBack (makePair (elemIntxn.first , elems[ie])) ;
          edgeIntxns.pushBack (makePair (elemIntxn.second , elems[ie])) ;
        }
      }

      // Sort the edgeIntxns
      if ( sourceCoords < targetCoords ) edgeIntxns.sortAscending () ;
      if ( sourceCoords > targetCoords ) edgeIntxns.sortDescending () ;

      // Add the edgeIntxns to the back of crackIntxns
      crackIntxns.insert (crackIntxns.end (), edgeIntxns) ;
    }

    // Loop over the crackIntxns in steps of 2
    for ( idx_t j = 0 ; j < crackIntxns.size () ; j+=2 ){
      vertex_t sourceElement = elemGraph.vertices[crackIntxns[j].first] ;
      vertex_t targetElement = elemGraph.vertices[crackIntxns[j+1].first] ;

      // Connect the vertices in the elemGraph
      elemGraph.addEdge (sourceElement, targetElement) ;
    }
  }
}

```

### 3.1.7 makeShortCuts

As mentioned in section 3.1.6, an additional difficulty for the `elemGraph`, which was not an issue for the `atomGraph`, stems from the fact that elements might be connected with the same element multiple times. This means that the acyclical property of the `atomGraph` is not preserved, and local cycles may appear. Four examples of skeleton curves that would produce such cycles are given in figure 3.6. It is noteworthy to mention that all series of cycles can be said to originate in an element containing an atom. After all, only at the location of an atom is a change in direction of the skeleton curve possible.

Because of this fact, the `makeElemGraph` itself has been set up in such a way that all elements containing an atom are not considered. This prevents the addition of unnecessary segments to the vertices of the `elemGraph`, which would later need to be removed. In the function no distinction is made between corners<sup>4</sup> and junctions<sup>5</sup>, since these two categories are not mutually exclusive. Although it appears unlikely, configurations that fall within both of these definitions can be thought of, and do occasionally occur in practice, as will be shown in figure 4.5b.

In listing 3.8, the pseudocode of the `makeShortCuts` function is given. The first part of the function adds additional segments to the `elemGraph`. Since these segments are the same for corners and junctions, as shown in figure 3.6, this part of the procedure is executed for every element that contains an atom. Afterwards, it is checked whether all intersection points lie on the same element edge. If this is the case, the intersection points are removed, along with all segments that connect to it.

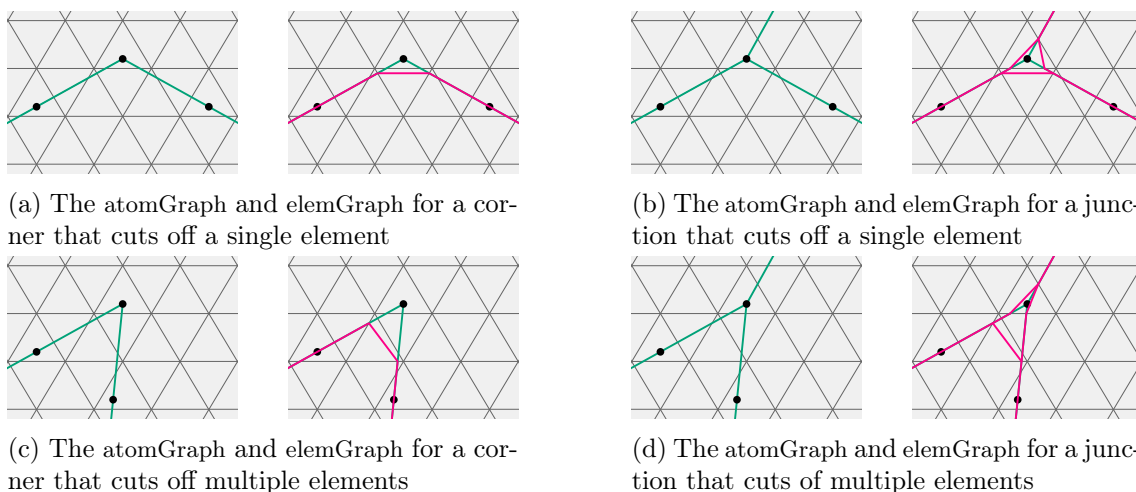


Figure 3.6: A comparison of the `atomGraph` and the `elemGraph` that is produced by the `makeElemGraph` function

<sup>4</sup>Corners are defined as elements with more than one intersection points, all of which lie on the same element edge.

<sup>5</sup>Junctions are defined as elements with more than two intersection points.

Listing 3.8: Pseudocode for the makeShortCuts function. This function makes the required modifications to the elemGraph for it to be used by the makePhantomNodes function.

```

void makeShortCuts
( graph_t          elemGraph )
{
    // Loop over the vertices
    for ( idx_t iv = 0 ; iv < elemGraph.vertices.size () ; iv++ ){
        vertex_t sourceElement = elemGraph.vertices[iv] ;
        set <edge_t> intxns = sourceElement.edges ;
        // Skip the element if it does not contain an atom
        if ( sourceElement.atoms.size () == 0 ) continue ;

        // Find the point where the outgoing edges point to different elements
        for ( idx_t io = 0 ; io < intxns.size () ; io++ ){
            for ( idx_t ip = 0 ; ip < intxns.size () ; ip++ ){
                // Make sure every combination of io and ip is treated once
                if ( io <= ip ) continue ;

                vertex_t targetElement = sourceElement ;
                point_t oPoint = intxns[io].coords ;
                point_t pPoint = intxns[ip].coords ;

                while ( true ){
                    getOppositePoints (oPoint , pPoint , targetElement.segments) ;
                    vertex_t oElem = getVertexFromPoint (oPoint , targetElement) ;
                    vertex_t pElem = getVertexFromPoint (pPoint , targetElement) ;

                    if ( oElem != pElem ) break ;
                    // Go to the next element as long as oElem and pElem are the same
                    targetElement = oElem ;
                }
                targetElement.segments.pushBack ( segment_t (oPoint , pPoint) ) ;
            }
        }

        if ( sourceElement.edges.size () < 2 ) continue ;

        while ( true ){
            // Break the loop if sourceElement has two different adjacent elements
            if ( not onlyOneAdjacent ( sourceElement ) ) break ;

            intxns = sourceElement.edges ;
            targetElement = intxns[0].target ;
            set <segment_t> segs = targetElement.segments ;

            for ( idx_t io = 0 ; io < intxns.size () ; io++ ){
                // Remove all target element segment pointing to the source element
                for ( idx_t iseg = 0 ; iseg < segs.size () ; iseg++ ){
                    if ( segs[iseg].containsPoint (intxns[io].coords) ){
                        segs.erase (iseg);
                    }
                }
                // Remove the edge itself
                elemGraph.removeEdge (intxns[io]) ;
            }
            // Progress to the next element
            sourceElement = targetElement ;
        }
    }
}

```

### 3.1.8 makePhantomNodes

Once the full `elemGraph` is known, and preparations have been taken to avoid duplicate adjacent vertices when looping over the `elemGraph`, the `makePhantomNodes` function can be called. As mentioned in section 3.1.7, this function iterates over the vertices over the `elemGraph`, turning right whenever it encounters a junction. During the procedure, the endpoints of the graph<sup>6</sup> will be used as anchors, at which a crack face begins and ends. This ensures that each side of the crack is treated in one continuous motion, which in turn makes it easier to make a distinction between the ‘left’ and ‘right’ side of the crack.

Since it was shown in section 3.1.4 that the `atomGraph` does not contain any cycles, there will be no cycles in the `elemGraph` that are larger than 2. In section 3.1.7, precautions that have been taken to avoid any hindrance in the `makePhantomNodes` procedure that may occur due to small loops in the `elemGraph`. For these reasons, the `elemGraph` will be treated as if it does not contain any loops when handling the iteration over the graph.

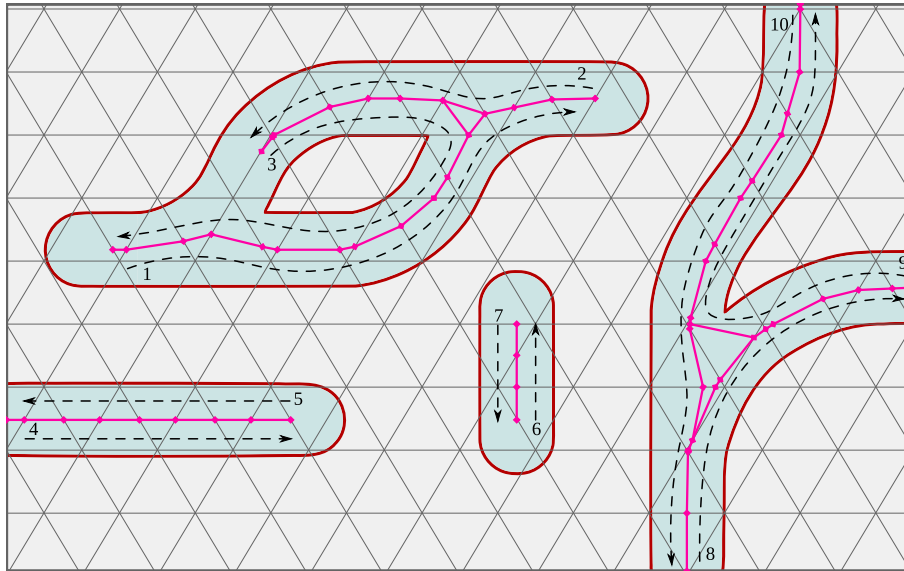


Figure 3.7: A visualization of the way the `makePhantomNodes` function iterates over a complicated crack pattern

In figure 3.7, a general overview of the manner in which the `makePhantomNodes` function should loop over the `elemGraph` is given. In order to accomplish this, special attention needs to be paid to the way that junctions are handled. After all, whenever a junction may appear, multiple internal segments can be found, out of which the correct segment needs to be chosen. To demonstrate how the procedure ensures this, an example is given in figure 3.8. Here, the current and previous element are known, as well as the current and previous intersection points, as indicated in the figure<sup>7</sup>. When the `getNextIntxn` function is called, it checks all segments in the `currElement`, and determines which segment forms the smallest angle with the internal segment in the `prevElement`. In the case of the example shown in figure 3.8, segment 2 makes the smallest angle, and so the intersection point at the bottom of the `currElement` will become the `nextIntxn`, which causes element A to become the `nextElement`.

The pseudocode for the `makePhantomNodes` procedure can be found in listing 3.9, and the `getNextIntxn` function can be found in listing 3.10.

<sup>6</sup>i.e. the element that have only one adjacent element

<sup>7</sup>It is important to emphasize again that the elements are stored as vertices and the intersection points as edges of the `elemGraph`. The segments as shown in the figure are stored as properties of the vertices.

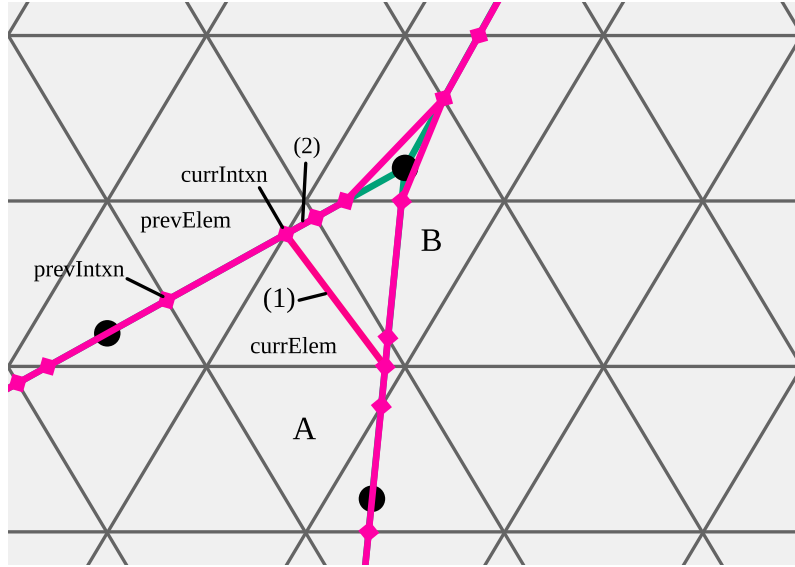


Figure 3.8: An example of the procedure by which the next intersection point is determined by the getNextIntxn function in listing 3.10

Listing 3.9: Pseudocode for the makePhantomNodes function. This function takes the elemGraph as input, and returns a set of phantom elements, a set of phantom nodes and a set of regular nodes.

```

pair <set <set <element_t>>, pair <set <set <node_t>>>> makePhantomNodes
( graph_t          elemGraph )
{
    // Initialize the phantomElems, phantomNodes and regularNodes
    set <set <element_t>> phantomElems ;
    set <set <node_t>> phantomNodes ;
    set <set <node_t>> regularNodes ;

    // Initialize the previous, current and next element
    for ( idx_t iv = 0 ; iv < elemGraph.vertices.size () ; iv++ ){
        if ( elemGraph.vertices[iv].edges.size () == 1 ){
            vertex_t prevElement = elemGraph.vertices[iv].adjacentVertices[0] ;
            vertex_t currElement = elemGraph.vertices[iv] ;
            vertex_t nextElement = prevElement ;
            vertex_t firstElement = currElement ;
            point_t prevIntxn = currElement.edges[0].coords ;
            point_t currIntxn = prevIntxn ;
            point_t nextIntxn = prevIntxn ;
            break ;
        }
    }

    while ( true ){
        // Initialize the phantom_elems, phantom_nodes and regular_nodes
        set <element_t> phantom_elems ;
        set <node_t> phantom_nodes ;
        set <node_t> regular_nodes ;

        // update the previous current and next element
        prevElement = currElement ;
        currElement = nextElement ;
        prevIntxn = currIntxn ;
    }
}

```

```

currIntxn = nextIntxn ;

// Check if currElement has 0, 1 or more segments
idx_t segmentCount = currElement.segments.size () ;

if ( segmentCount == 0 or currElement == firstElement ){
    // Endpoint is met, so the crack face is added to the pattern
    phantomElems.pushBack (phantom_elems.removeDuplicates () ) ;
    phantomNodes.pushBack (phantom_nodes.removeDuplicates () ) ;
    regularNodes.pushBack (regular_nodes.removeDuplicates () ) ;

    // Exit the loop when it is back at the first element
    if ( currElement == firstElement ) break ;

    // Clear the sets for the next loop
    phantom_elems.clear () ;
    phantom_nodes.clear () ;
    regular_nodes.clear () ;
}

// Find the nextIntxn based on currElement, prevIntxn and currIntxn
nextIntxn = getNextIntxn (currElement, prevIntxn, currIntxn) ;

// Find nextElement based on nextIntxn
for ( idx_t io = 0 ; io < currElemnt.edges.size () ; io++ ){
    if ( currElement.edges[io].coords == nextIntxn ){
        nextElement = currElement.edges[io].targetVertex ;
        break ;
    }
}

// Add the current element to phantom_elems
phantom_elems.pushBack (currElement) ;
set <node_t> nodes = currElement.getNodes () ;

// Check if each node falls left or right of the crack
for ( idx_t in = 0 ; in < nodes.size () ; in++ ){
    double angle = getAngle (currIntxn, nextIntxn, nodes[in].coords) ;
    if ( angle > 0 ) phantom_nodes.pushBack (nodes[in]) ;
    if ( angle < 0 ) regular_nodes.pushBack (nodes[in]) ;
}

// Turn all nodes of endpoint elements into regular nodes
for ( idx_t iv = 0 ; iv < elemGraph.vertices.size () ; iv++ ){
    vertex_t element = elemGraph.vertices[iv] ;
    if ( element.edges.size () == 1 ){
        set <node_t> nodes = elems[element.index].getNodes () ;

        for ( idx_t in = 0 ; nodes.size () ; in++ ){
            for ( idx_t i = 0 ; i < PhantomElems.size () ; i++ ){
                for ( idx_t j = 0 ; j < PhantomNodes[i].size () ; j++ ){
                    if ( PhantomNodes[i][j] == nodes[in] ){
                        PhantomNodes[i][j].remove () ;
                        RegularNodes[i].pushBack (nodes[in]) ;
                    }
                }
            }
        }
    }
}
}

```



Listing 3.10: Pseudocode for the getNextIntxn function. This function determines the next Intersection point based on the segments inside the current element, as well as the previous and current intersection points. It is used by the makePhantomNodes function in listing 3.9 to iterate along the crack.

```

point_t getNextIntxn

( vertex_t      currElement ,
  point_t      prevIntxn ,
  point_t      currIntxn )

{
    // Initialize the nextIntxn
    point_t nextIntxn

    // Check if currElement has 0, 1 or more segments
    idx_t segmentCount = currElement.segments.size () ;

    if ( segmentCount = 0 ){
        // The current element is an endpoint, so the loop turns around
        nextIntxn = currIntxn ;
    } else if ( segmentCount == 1 ){
        // The next intersection point is on the other side of the segment
        segment_t segment = currElement.segments[0] ;
        if ( currIntxn == segment.sourcePoint ){
            nextIntxn = segment.targetPoint ;
        } else {
            nextIntxn = segment.sourcePoint ;
        }
    } else if ( segmentCount >= 2 ){
        // Set the next intersection correctly based on the smallest angle
        double minAngle = 2 * pi () ;

        for ( idx_t is = 0 ; is < segmentCount ; is++ ){
            segment_t segment currElement.segments[is] ;

            // Check for each segment if it connects to currIntxn
            if ( segment.sourcePoint == currIntxn ){
                point_t possibleNextIntxn = segment.targetPoint
            } else if ( segment.targetPoint == currIntxn ){
                point_t possibleNextIntxn = segment.sourcePoint
            } else {
                continue ;
            }

            // Set nextIntxn to be the point that would give the smallest angle
            double angle = getAngle (possibleNextIntxn, currIntxn, prevIntxn) ;
            if ( angle < minAngle ){
                minAngle = angle ;
                nextIntxn = possibleNextIntxn ;
            }
        }
    }

    // Return the next intersection
    return nextIntxn ;
}

```

## 3.2 PhantomNodeModel

Ultimately, the Skeletonizer returns three `set <set <idx_t>>` types, containing the `phantomElements`, `phantomNodes` and `regularNodes`. It should be noted that all three of these sets refer to the original mesh, since the Skeletonizer does not make any modifications to the mesh itself. The addition of phantom nodes and elements will occur in the `PhantomNodeModel`, though the modifications that need to occur in other parts of the code to accommodate for the added elements and nodes will also be discussed in this section.

### 3.2.1 createPhantomNodes

In order to easily switch between the updated mesh and the original mesh, two new types are defined, called `mappingOldToNew_t` and `mappingNewToOld_t`. These types are defined as `set <pair <idx_t, set <idx_t>>>` and `set <pair <idx_t, idx_t>>`, respectively. The reason for the difference between these two types lies in the fact that an old element can have multiple phantom elements in the updated mesh, whereas it is not possible for any element to have multiple original elements. Below, an example is given which demonstrates the mappings for a mesh with  $N$  original elements and  $P$  additional phantom elements. In the example, element 1 has been replaced by phantom elements  $N + 0$  and  $N + 1$ . The pseudocode for the `createPhantomNodes` function can be found in listing 3.11.

elems	mapOldToNew	mapNewToOld
<code>set &lt;element_t&gt;</code>	<code>set &lt;set &lt;idx_t&gt;&gt;</code>	<code>set &lt;idx_t&gt;</code>
$\begin{bmatrix} 0 \\ 1 \\ 2 \\ \vdots \\ N-1 \\ N+0 \\ N+1 \\ \vdots \\ N+P-1 \end{bmatrix}$	$\begin{bmatrix} 0 & [0] \\ 1 & [N+0 \quad N+1] \\ 2 & [2] \\ \vdots & \\ N-1 & [N-1] \end{bmatrix}$	$\begin{bmatrix} 0 & 0 \\ 2 & 2 \\ \vdots & \vdots \\ N-1 & N-1 \\ N-0 & 1 \\ N+1 & 1 \\ \vdots & \vdots \\ N+P-1 & 3 \end{bmatrix}$

Listing 3.11: Pseudocode for the createPhantomNodes function. This function adds the phantom elements and nodes to the mesh, based on the phantomElements and phantomNodes that are returned by the Skeletonizer.

```

void createPhantomNodes

( set <element_t>      elems ,
  set <node_t>         nodes ,
  set <dof_t>          dofs ,
  set <set <element_t>> phantomElements ,
  set <set <node_t>>   phantomNodes )

{
    // Initialize the mappings between the old and new elements
    mappingOldToNew_t mapOldToNew ;
    mappingNewToOld_t mapNewToOld ;

    // Loop over the phantomElements
    for ( idx_t i = 0 ; i < phantomElements.size () ; i++ ){
        set <pair <node_t, node_t>> nodeMapping ;

        for ( idx_t j = 0 ; j < phantomNodes[i].size () ; j++ ){
            // Create a new node on the same location as the old node
            node_t oldNode = phantomNodes[i][j] ;
            point_t oldCoords = oldNode.getCoords () ;
            node_t newNode = nodes.addNode (oldCoords) ;

            // Map the new node to the old node
            nodeMapping.pushBack (makePair (oldNode, newNode)) ;

            // Add the new degrees of freedom to the set of dofs
            dofs.addDofs (newNode) ;
        }

        for ( idx_t j = 0 ; j < phantomElems[i].size () ; j++ ){
            // Get the old element and its nodes
            element_t oldElement = phantomElements[i][j] ;
            set <node_t> nodes = oldElement.getNodes () ;

            // Convert all nodes that are found in the mapping
            for ( idx_t in = 0 ; in < nodes.size () ; in++ ){
                if ( nodeMapping.find (nodes[in]) != NULL ){
                    nodes[in] = nodeMapping.find (nodes[in]) ;
                }
            }

            // Add a new element using the converted nodes
            element_t newElement = elems.addElement (nodes) ;

            // Add the new element to the mappings
            mapOldToNew[oldElement].pushBack(newElement) ;
            mapNewToOld[newElement] = oldElement ;
        }
    }

    // Add all other elements to the mapping, mapping to themselves
    for ( idx_t i = 0 ; i < elems.size () ; i++ ){
        if ( mapOldToNew[i] != NULL and mapNewToOld[i] != NULL ){
            mapOldToNew[i].pushBack (i) ;
            mapNewToOld[i] = i ;
        }
    }
}

```

### 3.2.2 updateElements and getOriginalElements

During the procedure, only a single `set <element_t>` is used, with `set <idx_t>` types to keep track of which elements are part of the original mesh<sup>8</sup>. These `set <idx_t>` types will always be called `ielems`, possibly with some suffix to distinguish it. The three sets that will be used are `ielems_0`, `ielems_i` and `ielems_0_i`, which point to the original elements, updated elements, and original versions of updated elements, respectively.

`ielems_i` can be created from `ielems_0` using the `updateElements` function, which replaces all elements with phantom elements when applicable. This `ielems_i` can be mapped back with the `getOriginalElements` function, which yields the `ielems_0_i` set. Since `ielems_i` and `ielems_0_i` will always have the same size, the same index can be used when looping over the elements in the mesh, which allows for convenient switching between the original and updated mesh whenever necessary in other procedures. The pseudocode for these two functions can be found in appendix A.8. Using the mappings from section 3.2.1, the following three sets would be created:

$$\begin{array}{ccc}
 \text{ielems\_0} & \text{ielems\_i} & \text{ielems\_0\_i} \\
 \text{set } \langle \text{idx\_t} \rangle & \text{set } \langle \text{idx\_t} \rangle & \text{set } \langle \text{idx\_t} \rangle \\
 \begin{bmatrix} 0 \\ 1 \\ 2 \\ \vdots \\ N-1 \end{bmatrix} & \begin{bmatrix} 0 \\ N+0 \\ N+1 \\ 2 \\ \vdots \\ N-1 \end{bmatrix} & \begin{bmatrix} 0 \\ 1 \\ 1 \\ 2 \\ \vdots \\ N-1 \end{bmatrix}
 \end{array}$$

### 3.2.3 Additional modifications

When the mesh has been modified, it is necessary to modify other parts of the code, most importantly the parts where the stiffness matrix and external force vectors are built up. The main distinction that needs to be made is whether or not any given variable needs to be based on the original or updated mesh. In the implementation, all variables related to the level set field are only defined on the original mesh, whereas variables related to the mechanical problem are mostly defined on the updated mesh.

An overview of which variables are defined on the original mesh, and which are defined on the updated mesh is given in table 3.2. A point of attention is the fact that  $Y$  is based on the updated mesh, whereas  $\bar{Y}$  is only defined on the original mesh. This is due to the fact that  $Y$  needs to be computed based on the strain vector  $\boldsymbol{\varepsilon}$ , which is based on the updated mesh, but in order to compute the front velocity  $v$ , the  $\bar{Y}$  needs to be defined on the original mesh.

In equation 3.1, the parts of equation 2.6, which concerns the elemental stiffness matrix and force vector, have been colored blue or green depending on whether they should be based on the original mesh or the updated mesh, respectively. Since almost all variables except  $d$  need to be based on the updated mesh, adapting the builder of the stiffness matrix and external force vector to accommodate for the updated mesh is relatively straightforward. The integration domain  $\Omega^e$  has also been colored green to indicate that the loop

<sup>8</sup>In fact, this was also the approach that was taken in the code for section 3.1, although this was often glossed over in the pseudocode in order to improve readability, as explained in the introduction to this chapter.

Table 3.2: An overview of which variables visual overview of the global algorithm for the TLS\_V1. This same algorithm is used for the TLS\_V2 [18]

Variable	Symbol	Mesh
Level set field	$\phi$	Original
Damage	$\mathbf{d}$	Original
Displacement vector	$\mathbf{u}$	Updated
External force vector	$\mathbf{f}_{ext}$	Updated
Stiffness matrix	$\mathbf{K}$	Updated
Strain vector	$\boldsymbol{\varepsilon}$	Updated
Stress vector	$\boldsymbol{\varepsilon}$	Updated
Strain energy	$\psi$	Updated
Energy release rate	$Y$	Updated
Averaged release rate	$\bar{Y}$	Original
Level set increment	$a$	Original
Front velocity	$v$	Original

needs to be performed over the updated elements.

$$\begin{aligned}
 \mathbf{K}^e &= \int_{\Omega^e} \mathbf{B}^T \cdot \mathbf{D}(\boldsymbol{\varepsilon}, \mathbf{d}) \cdot \mathbf{B} \, d\Omega \\
 \mathbf{f}^e &= \int_{\Omega^e} \mathbf{N}^T \cdot \mathbf{t} \, d\Omega
 \end{aligned} \tag{3.1}$$

For the system of equations that is used to determine  $\bar{Y}$  in equation 2.18, it is more challenging to adapt for the updated mesh. As mentioned,  $Y$  is only defined on the updated mesh, whereas  $\bar{Y}$  is needed on the original mesh. In equation 3.2, the same highlighting has been applied, indicating which parts are based on the updated mesh, and which are based on the original mesh.

$$\begin{aligned}
 K_{ij} &= \int_{\Omega_d} d'(\phi) N_i N_j + \frac{\kappa h^2}{l_c} \frac{\partial N_i}{\partial x_k} \frac{\partial N_j}{\partial x_k} \, d\Omega \\
 L_{ij} &= \int_{\Omega_d} l_c \left( \frac{\partial N_i}{\partial x_k} \frac{\partial \phi}{\partial x_k} \right) \left( \frac{\partial N_j}{\partial x_k} \frac{\partial \phi}{\partial x_k} \right) \, d\Omega \\
 f_i^Y &= \int_{\Omega_d} N_i d'(\phi) Y \, d\Omega
 \end{aligned} \tag{3.2}$$

Even though  $K_{ij}$ ,  $L_{ij}$  and  $f_i^Y$  all refer to the original mesh, it is still necessary to loop over the updated mesh rather than the original mesh. This does mean that, for instance, the term  $d'(\phi)$  is accounted for twice for a phantom element. However, since this term is multiplied by the shape functions of these element, which consider only part of the element, the correct result will still be returned. The same applies to any other term that is based on the original mesh.

## Chapter 4

# Verification

In order to validate the implementation of both the Skeletonizer and the PhantomNodeModel as explained in sections 3.1 and 3.2, two different test will be used. The first test is a rail shear test based on Greenhalgh et al. [41, 42], which has been modeled before using TLS\_V1 by Van der Meer et al. [7]. This rail shear test is based on ASTM standard ASTM C273-20 [43] Due to the branching and merging of damage fronts in the material, as well as the complex crack patterns, it provides a good test for the Skeletonizer function.

The second test is based on the compact tension (CT) test from ISO standard ISO 7539:6 [44] and ASTM standard ASTM E647-15 [45]. Although these standards are intended for the fatigue testing of metals and alloys, they will be used in this thesis to verify the functionality of the PhantomNodeModel by loading the specimen under pure tension. It was also used by Van der Meer et al. [7] to demonstrate the influence of the initiation model on the results of a TLS\_V1 simulation.

### 4.1 Skeletonizer

For the rail shear test, the same dimensions were used as those given by Van der Meer et al. [7], as shown in figure 4.1. For the core, the Youngs modulus, Poisson's ratio, tensile strength and fracture toughness are given by  $E = 40 \text{ kN mm}^{-2}$ ,  $\nu = 0.2$ ,  $f_t = 20 \text{ N mm}^{-2}$  and  $G_c = 0.05 \text{ N mm}^{-1}$ , respectively. For the two faces, these properties are set to  $E = 200 \text{ kN mm}^{-2}$ ,  $\nu = 0.3$ ,  $f_t = 20 \text{ N mm}^{-2}$  and  $G_c = 0.05 \text{ N mm}^{-1}$ , respectively.

The critical length has been set to  $l_c = 0.60 \text{ mm}$ , and an arctangent-based damage function  $f(\phi)$  used by Bernard et al. [6] will be used in this thesis as well, which is given by

$$f(\phi) = c_2 \arctan \left( c_1 \left( \frac{\phi}{l_c} - c_3 \right) \right) + c_4 \quad (4.1)$$

where

$$\begin{aligned} c_1 &= 10 \\ c_2 &= \frac{1}{\arctan(c_1(1 - c_3)) - \arctan(-c_1 c_3)} \\ c_3 &= 0.5 \\ c_4 &= -c_2 \arctan(-c_1 c_3) \end{aligned} \quad (4.2)$$

To ensure that no infinite strains occur for elements that are completely inside the iso- $l_c$  curve, the maximum damage has been set to 0.99. This ensures that some stiffness still

remains in fully damaged elements, which prevents singularities in the nodal displacements from occurring.

In order to compute the critical energy release rate  $Y_c$ , the approach by Van der Meer et al. [7] is followed, where  $Y_c$  is given by

$$\log(Y_c) = \log(Y_c^f) + \frac{\bar{\phi}}{\bar{\phi}_{\max}} \left( \log(Y_c^G) - \log(Y_c^f) \right) \quad (4.3)$$

with

$$\begin{aligned} Y_c^G &= \frac{G_c}{2 \int_0^{l_c} f(\phi) d\phi} \\ Y_c^f &= \frac{1}{2} \frac{f_t^2}{E} \end{aligned} \quad (4.4)$$

In equation 4.3,  $\bar{\phi}$  refers to the averaged value of  $\phi$  over the domain where  $0 < \phi < l_c$ , which can vary from 0 to 0.5 [7].  $\bar{\phi}_{\max}$  is set to  $\frac{l_c}{3}$ . For the given values, the energy-based and strength-based critical release rates evaluate to  $Y_c^G = 0.05 \text{ N mm}^{-2}$  and  $Y_c^f = 0.005 \text{ N mm}^{-2}$ , respectively.

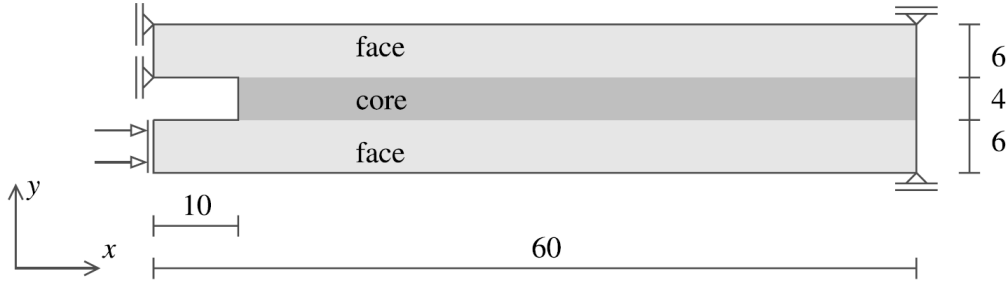


Figure 4.1: An overview of the measurements of the model for the rail shear test. [7]

The measurements of the rail shear test are given in figure 4.1. These are the same as those used in the FEA by Van der Meer et al. [7], although a courser mesh of three-noded triangular elements has been used, with a typical element size of  $h = 0.80 \text{ mm}$  for the two faces, and  $h = 0.14 \text{ mm}$  for the core. The meshing has been performed using Gmsh [46].

For the shrinking ball algorithm, the critical angle and initial radius have been set to  $t_{crit} = 90^\circ$  and  $r_{init} = 50 l_c$ , respectively. A minimum distance between the atoms of  $0.60 \text{ mm}$  is enforced, and  $\phi^* = 0.5 l_c$ . The resulting iso-0 curve, iso- $l_c$  curve, atoms and skeleton curve can be found in figure 4.2.

In figures 4.2a through 4.2e, the skeleton curve has been visualized for time steps 200, 600, 1000, 1400 and 1800, respectively. From visual inspection, it can be determined that the atoms are generated at the correct locations by the makeAtoms function from section 3.1.8, and that they are connected in the right manner for time steps 1 to 1400 by the makeAtomGraph function from section 3.1.4. However, when cycles start to develop in the iso-0 curve, issues start to appear. Since a shortest spanning tree algorithm is used, a skeleton curve that does not contain any cycles is created. An approach that can properly deal with damage zones that are topologically equivalent to an  $n$ -holed torus still needs to be found.



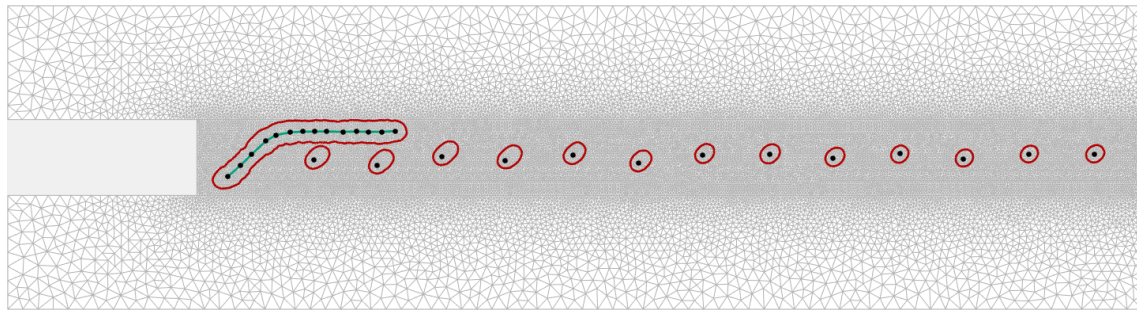
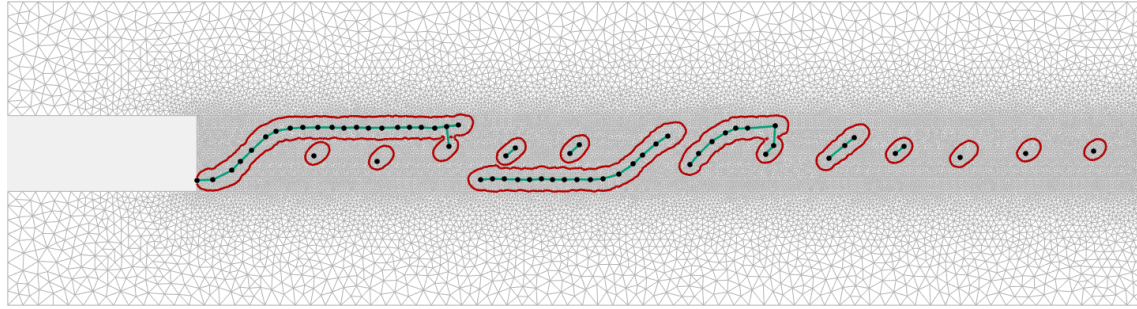
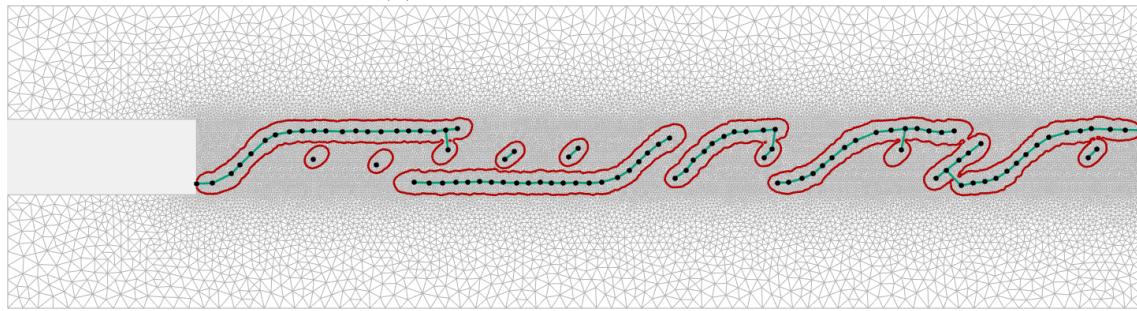
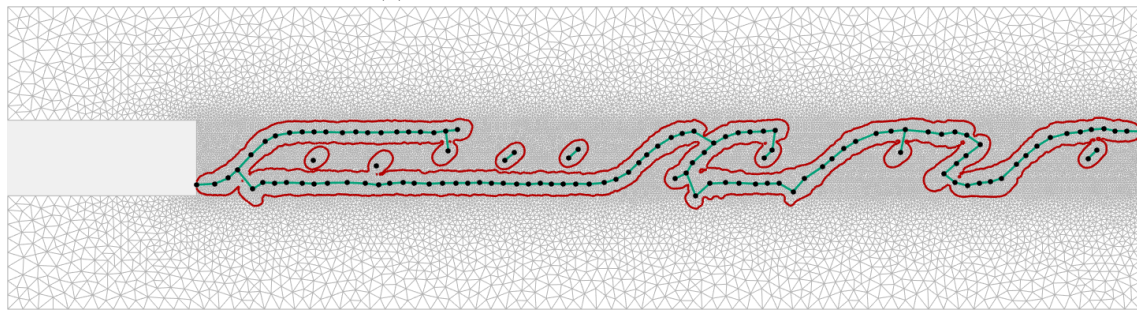
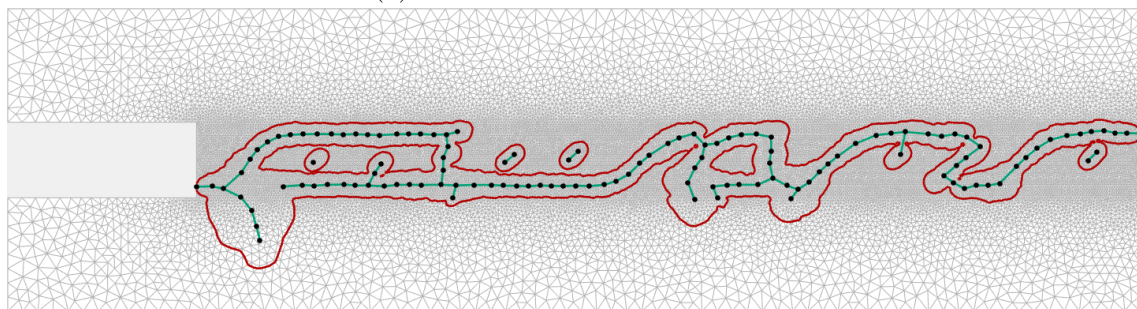
(a) The skeleton curve at  $t = 200$ (b) The skeleton curve at  $t = 600$ (c) The skeleton curve at  $t = 1000$ (d) The skeleton curve at  $t = 1400$ (e) The skeleton curve at  $t = 1800$ 

Figure 4.2: The skeleton curve at different time steps



Furthermore, once a full crack<sup>1</sup> has developed, the iso-0 curve starts widening beyond  $2l_c$ . It should be mentioned, however, that this is not a shortcoming of the model, but rather it is a result of the fact that the structure is no longer statically determinate. In figure 4.3, the load scale factor  $\gamma$  is plotted as a function of the time step  $t$ . It is obvious that at  $t = 1319$ , when the full crack has developed, the load factor increases to unrealistic values. Since both the stresses and strains are multiplied by this load factor, and the energy release rate is a function of these strains, the front velocity starts to grow rapidly in locations where usually only small growth occurs.

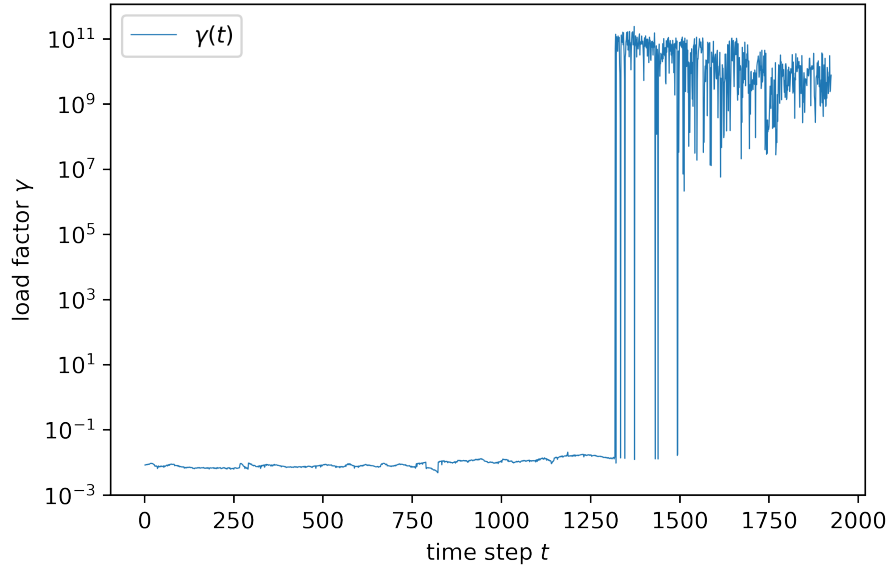


Figure 4.3: A plot of the load factor  $\gamma$  as a function of  $t$

To verify the `makeElemGraph` and `makePhantomNodes` functions, a closer inspection of the edge cases described in section 3.1.6 is required. In particular, the treatment of corners and junctions will be investigated, since these parts of the skeleton curve required special attention, as mentioned in section 3.1.7. The `elemGraphs` for a trivial corner, a trivial junction, and the four edge cases from figure 3.6 can be found in figure 4.4. Note that for figures 4.4e and 4.4f, a different time step had to be used, since these situations are less common, and were not found at  $t = 1500$ .

As demonstrated in figure 4.4, the `makeElemGraph` and `makeShortCuts` functions are able to convert the `atomGraph` to an `elemGraph` without adding superfluous segments to the vertices of the `elemGraph`. Additionally, the `makePhantomNodes` function is able to find the correct phantom nodes and regular nodes for all cases that are verified. It is clear that the presented approach works for both corners and junctions, regardless of the number of elements that are ‘cut off’ by the shortcut.

<sup>1</sup>i.e. a crack that completely intersects the material, causing a statically indeterminate structure.

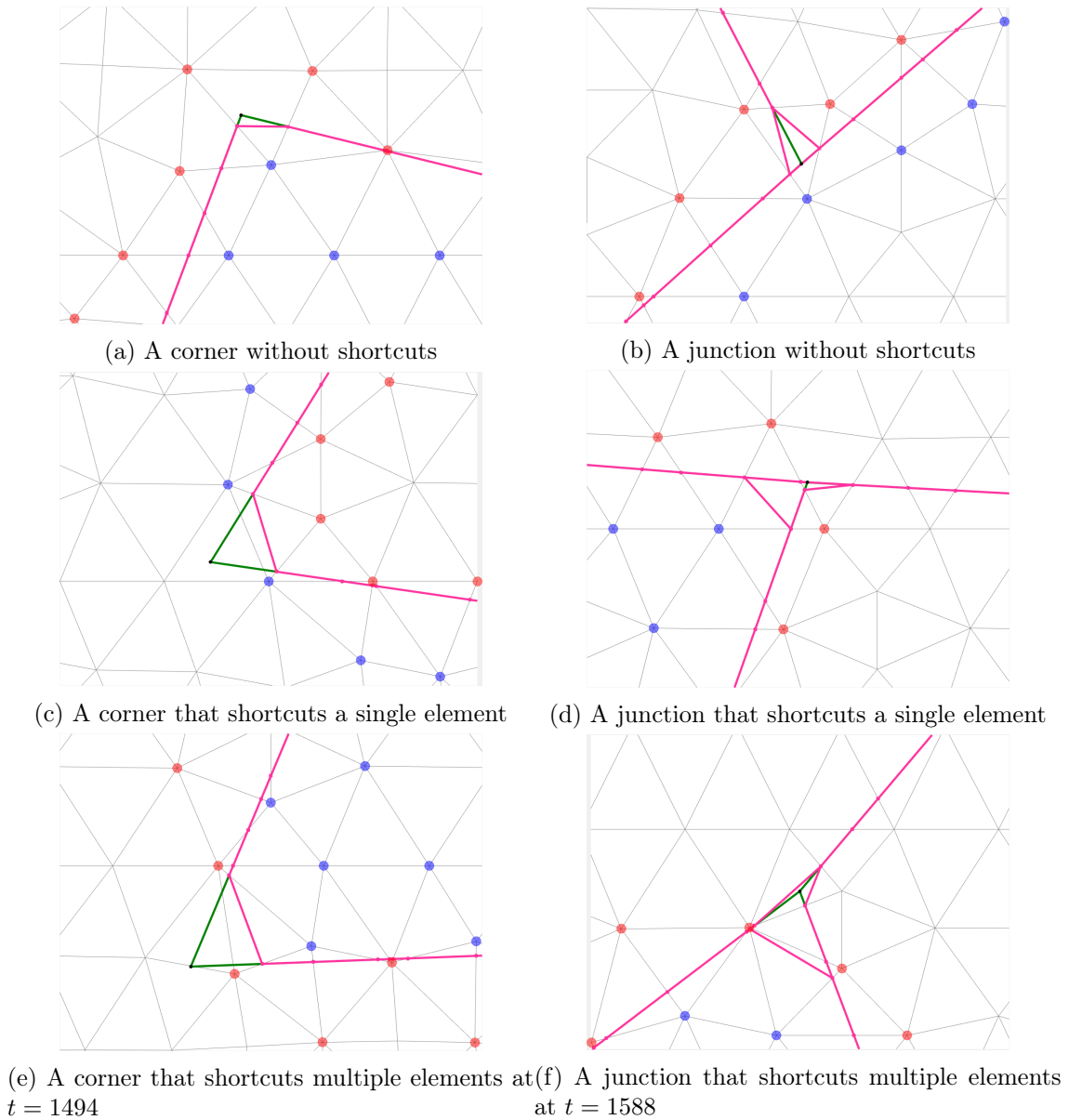
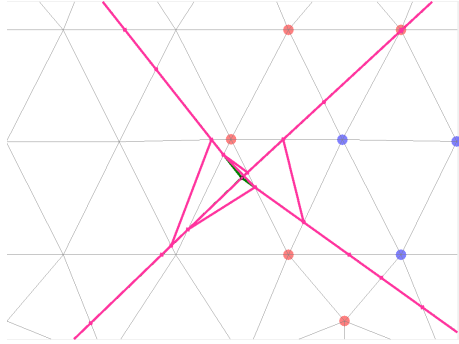


Figure 4.4: A comparison of the atomGraph and the elemGraph. The edges of the atomGraph are shown in green, whereas the segments that are stored in the vertices of the elemGraph are shown in pink. Additionally, the phantom nodes and regular nodes that belong to the phantom elements have been shaded red and blue, respectively. This has only been done for one side of the crack, to avoid duplicate coloring of the nodes.

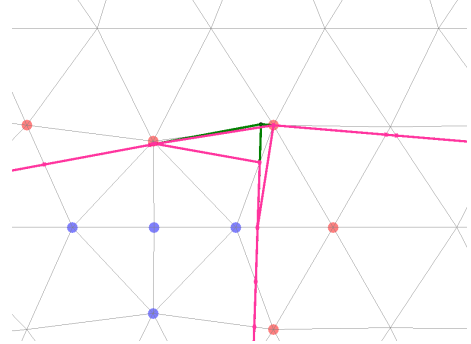
It should be noted, however, that all test cases presented so far only concern three-edged junctions, and junctions that intersect at least two different element edges. Although situations that do not adhere to these conditions are rare—in fact, for the presented example, this never occurred—these edge cases should still be handled correctly by the algorithm. To validate these situations, multiple simulations have been run with different initial settings, until the two cases presented in figure 4.5 were found.

As shown, the elemGraph that is produced by the proposed implementation still yields the correct phantom nodes and regular nodes for both cases. In figure 4.5a, it can be observed that the junction element does contain two superfluous segments, which intersect the element diagonally. These additional segments, however, do not affect the makePhantomNodes function, since this function will always turn right whenever possi-

ble. This implies that these diagonal segments will never be traversed, and will thus not affect the end result of the `makePhantomNodes` function. For the second edge case, shown in figure 4.5b, no superfluous segments were added by the `makeShortCuts` function, and the connectivity of the junction is handled as intended. Consequently, the `makePhantomNodes` function yields the correct phantom nodes and regular nodes for this edge case as well.



(a) A junction that connects to four different elements



(b) A junction that intersects only 1 element edge

Figure 4.5: A visualization of the `elemGraph` for two additional edge cases.

## 4.2 PhantomNodeModel

As stated in the introduction to this chapter, in order to validate the PhantomNodeModel, the CT-test from ISO standard ISO 7539:6 [44] and ASTM standard ASTM E647-15 [45] will be used. During the test, a vertical displacement will be imposed. Although these standards are intended for the fatigue testing of metals and alloys, they will be used in this thesis to verify the functionality of the PhantomNodeModel by loading the specimen under pure tension. It was also used by Van der Meer et al. [7] to demonstrate the influence of the initiation model on the results of a TLS\_V1 simulation.

The same material properties as those by Van der Meer et al. [7], which in turn are based on Li et al. [47], are used. This means that the Youngs modulus, Poisson's ratio, tensile strength and fracture toughness are given by  $E = 7 \text{ kN mm}^{-2}$ ,  $\nu = 0.3$ ,  $f_t = 79 \text{ N mm}^{-2}$  and  $G_c = 40 \text{ N mm}^{-1}$ , respectively.

The critical length has been set to  $l_c = 2.0 \text{ mm}$ , and the same arctangent based is applied as in the rail shear test, with the same values for  $c_1$ ,  $c_2$ ,  $c_3$  and  $c_4$ . Furthermore, the same mixed-mode energy release rate based on equation 4.3 by Van der Meer et al. [7] is applied, where  $Y_c^G$  and  $Y_c^f$  are again determined via equation 4.4, which yields  $Y_c^G = 20 \text{ N mm}^{-2}$  and  $Y_c^f = 0.4458 \text{ N mm}^{-2}$ .  $\bar{\phi}_{\max}$  and  $\phi^*$  have been set to  $\frac{\phi}{3}$  and  $\frac{\phi}{2}$ , respectively.

The meshing has again be performed with Gmsh [46], using a typical element size that ranges from  $h = 3.00 \text{ mm}$  at the outer faces to  $h = 0.20 \text{ mm}$  in the middle of the specimen at the height of the horizontal. The dimensions of the CT test are given in figure 4.6.

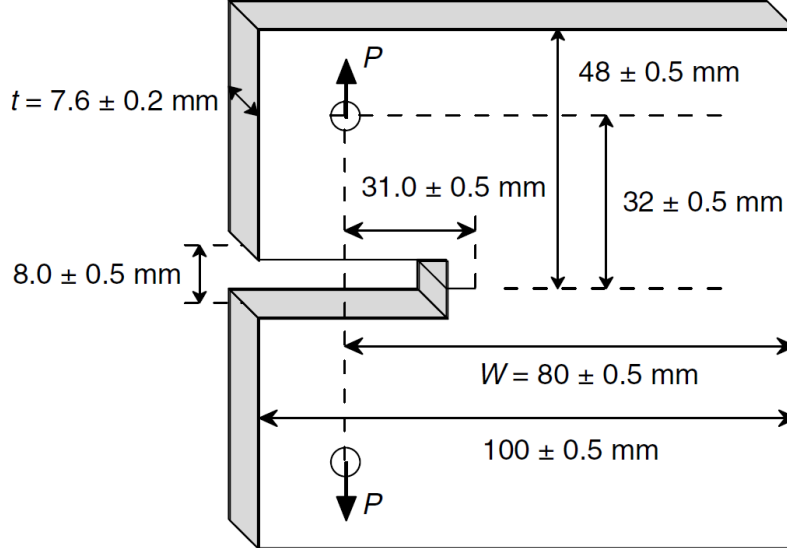
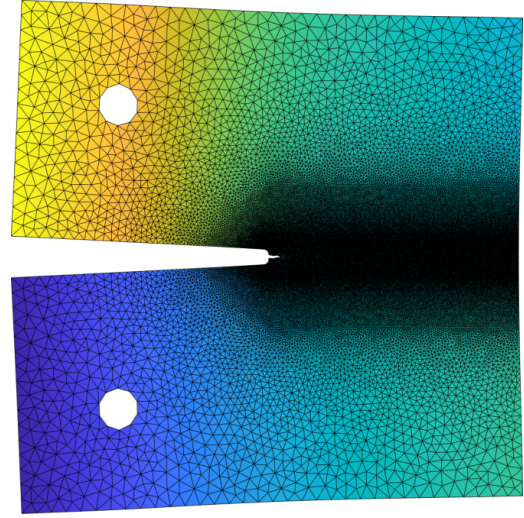
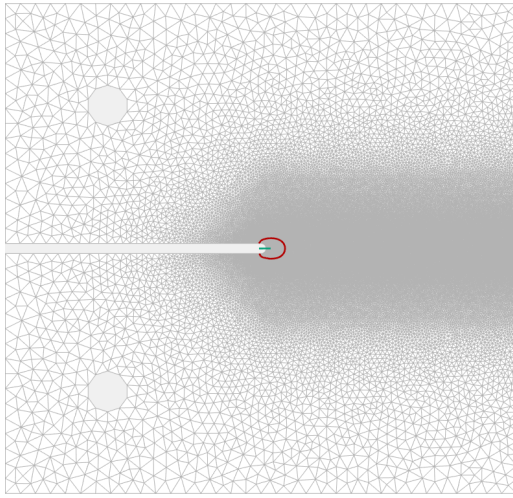


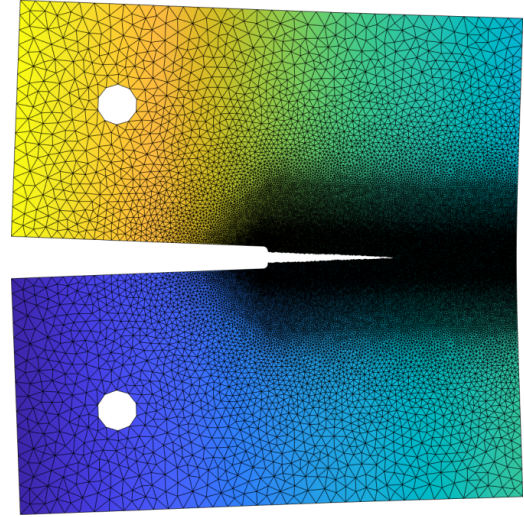
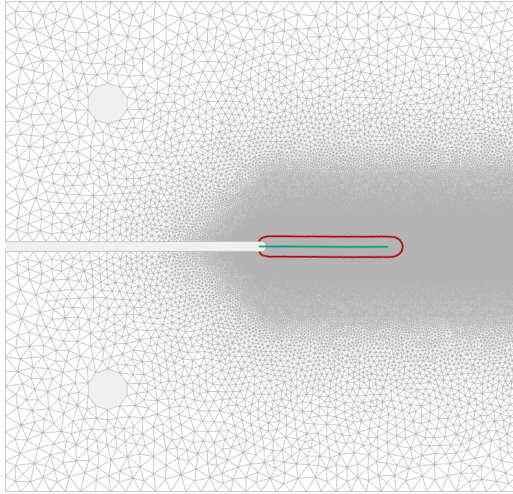
Figure 4.6: An overview of the measurements of the model for the CT test. [47]

In figure 4.7, the results from the CT test can be found, which has been performed including both the Skeletonizer and the PhantomNodeModel. The iso-0 curve and skeleton curve have been plotted along with the deformed specimen for time steps 50, 350 and 640. After time step 640, convergence was no longer reached, and the simulation breaks down. From visual inspection, it becomes apparent that the PhantomNodeModel is indeed able to update the mesh based on the skeleton curve, and explicitly model the crack formation. Additionally, issues arising due to the modification of the mesh are handled correctly when proceeding to the next time step, which allows the displacement jump to continuously develop over the course of the simulation.

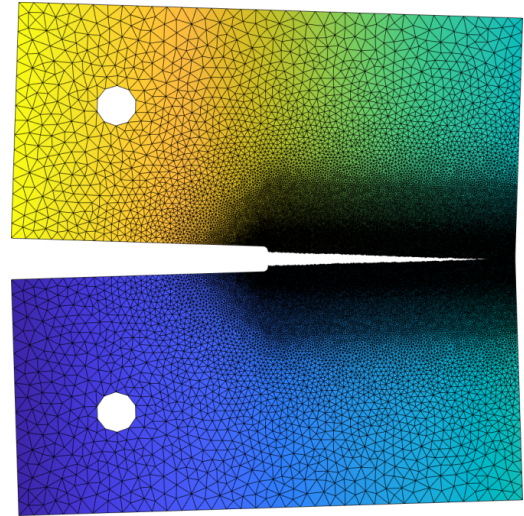
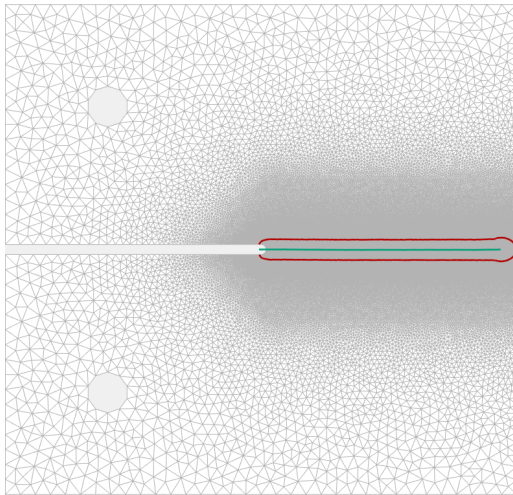




(a) The iso-0 curve and skeleton curve (left) and deformation plot (right) at  $t = 50$



(b) The iso-0 curve and skeleton curve (left) and deformation plot (right) at  $t = 350$



(c) The iso-0 curve and skeleton curve (left) and deformation plot (right) at  $t = 640$

Figure 4.7: A comparison of the skeleton curve and the resulting deformed specimen during the CT test. A scaling factor of 5 has been applied to the deformations. Due to the PhantomNodeModel, a crack formation becomes clearly visible.

In order to validate the connectivity of the phantom elements and nodes that have been generated by the PhantomNodeModel, a closer look is taken at time step  $t = 100$ . In figure 4.8, a closeup of the crack formation at this time step has been shown, with highlighted phantom elements. It can be observed that the shape of the phantom elements on either side of the crack is identical, which indicates that the elements that are intersected by the displacement jump have indeed been replaced by phantom elements, which are connected correctly to the regular elements, as well as one another.

Additionally, it can be seen in the right figure that the phantom elements on both sides of the crack end up at the crack tip element, where they share a single edge. This matches the intended behavior as described in section 2.3, and shown in figure 2.12.

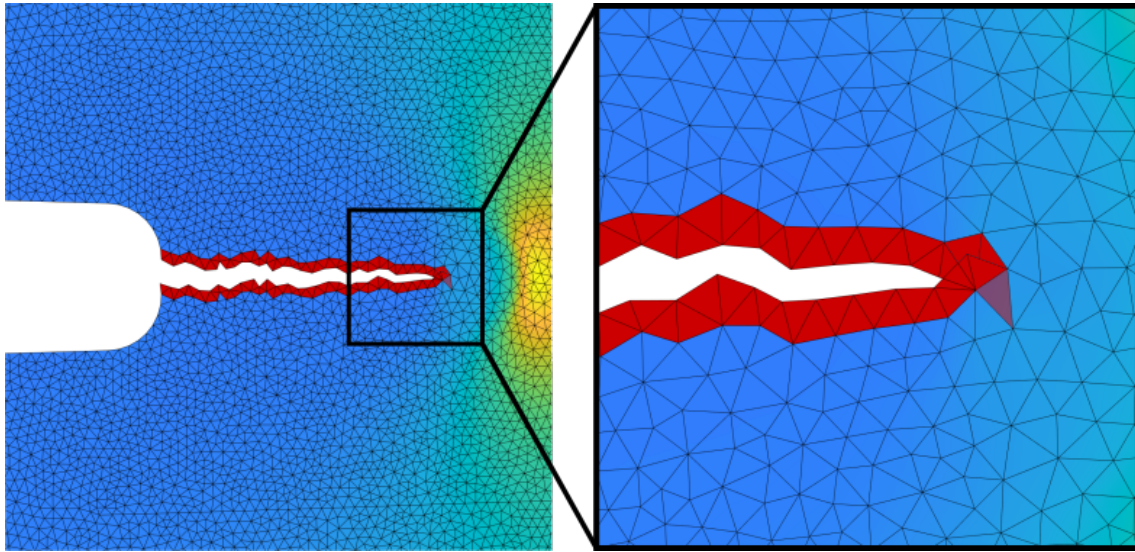


Figure 4.8: A closeup of the deformed mesh at time step  $t = 100$ . The displacements have been scaled up by a factor of 2.5, and a color gradient corresponding to the normal stress in  $y$ -direction,  $\sigma_{yy}$  has been applied. Additionally, phantom elements have been colored red, and the end point element has been shaded red.



### 4.3 Robustness

In order to validate the robustness of the algorithms that have been proposed, the influence of the mesh size on both the results and runtime of the procedure will be compared. However, the number of time steps that is required until a full crack has formed depends on the typical element size  $h$ , it is not possible to simply compare the same time step  $t$  for different mesh sizes. This issue can be solved by determining the time step  $t_{crack}$  at which a full crack has formed, and comparing the results at these time steps for the different element sizes. The huge jump in the load scale factor, shown in figure 4.3 can be used to easily pinpoint the time step at which this occurs. In table 4.1, the mesh sizes for which a model has been run are given, along with their respective values of  $t_{crack}$ . For  $h = 0.10$  mm, no value is given, because no full crack was found after the simulation had run for more than 42 hours. Due to time constraints, the run had to be broken off after  $t = 1200$ .

Mesh size mm	$t_{crack}$	Runtime		
		Skeletonizer (h)	PhantomNodeModel (s)	Total (h)
0.10	> 1200	(0.500)	(63.82)	(42.39)
0.14	1319	1.004	62.11	12.65
0.20	896	0.635	20.49	3.63
0.30	584	0.210	9.80	0.89
0.40	304	0.099	4.11	0.34

Table 4.1: The mesh sizes for which a simulation has been run, and the corresponding time steps at which a full crack is formed. For  $h = 0.10$  mm, the run was not completed, so the values at  $t = 1200$  are given, rather than  $t = t_{crack}$ . It should be emphasized that these values cannot be compared directly to those of the other mesh sizes.

In figure 4.9, the runtime of the Skeletonizer has been plotted as a function of  $t$ . In order to qualitatively compare the run-times, they are plotted as a percentage of the total runtime of each time step. Additionally, the rolling average over 20 time steps is used, rather than the raw data, to remove noise from the visualization. As mentioned previously, the time step is scaled with respect to  $t_{crack}$ . For  $h = 0.10$  mm, it has been assumed that  $t_{crack} = 2000$ .

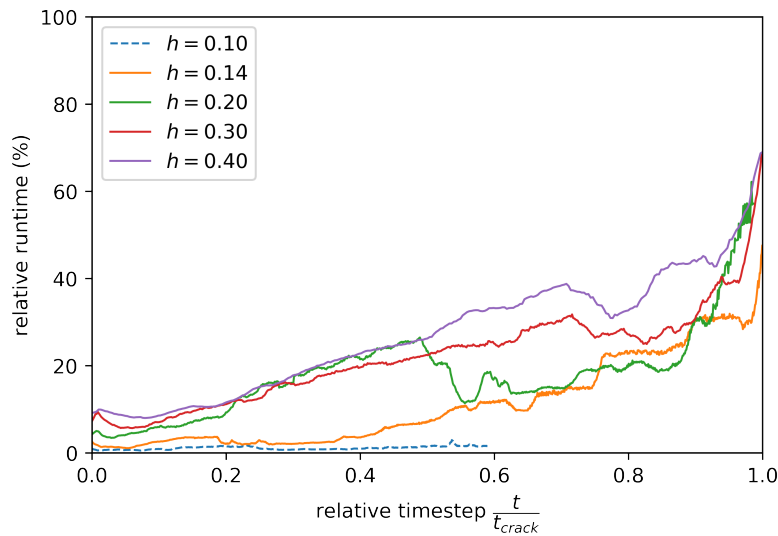
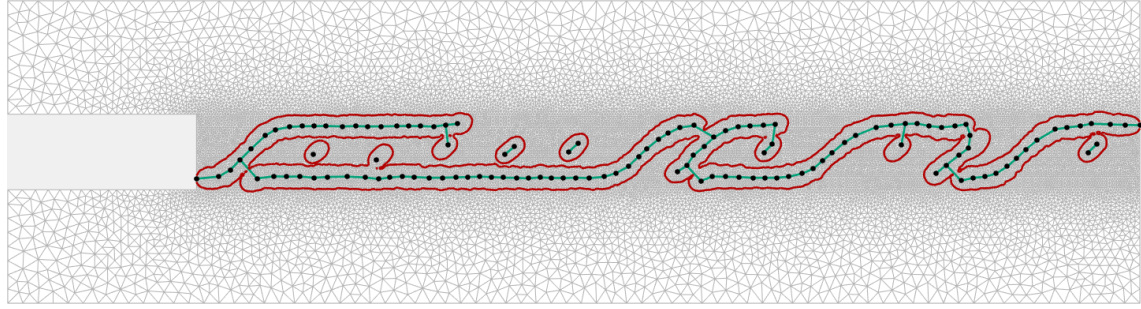
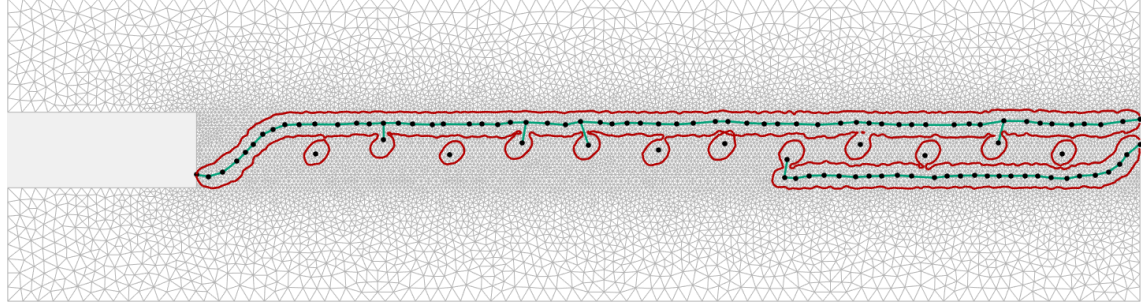


Figure 4.9: The relative runtime of the Skeletonizer procedure plotted for different mesh sizes

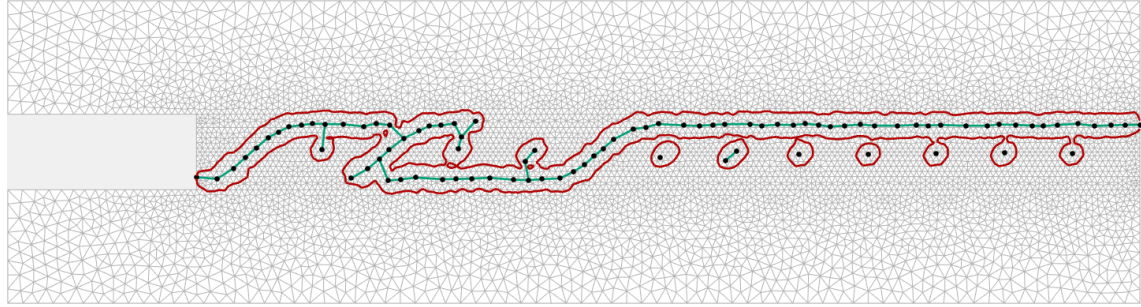
As shown in figure 4.9, the Skeletonizer procedure can take up a significant amount of the runtime. Especially when the crack pattern grows more complicated, it can account for up to 75 % of the runtime. However, as the mesh size decreases, the relative amount of the runtime that is dedicated to the Skeletonizer decreases as well. This implies that the  $\mathcal{O}$ -complexity of the Skeletonizer is less than the  $\mathcal{O}$ -complexity of the full procedure. As a result, it can be said that, despite taking up a large part of the runtime for coarse meshes, the Skeletonizer will not come to dominate the runtime of the whole procedure.



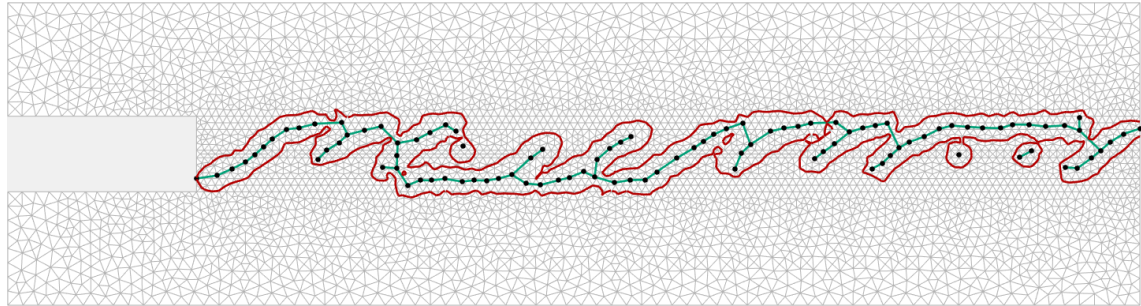
(a) The skeleton curve for  $h = 0.14$  mm at  $t = t_{crack} = 1319$



(b) The skeleton curve for  $h = 0.20$  mm at  $t = t_{crack} = 896$



(c) The skeleton curve for  $h = 0.30$  mm at  $t = t_{crack} = 584$



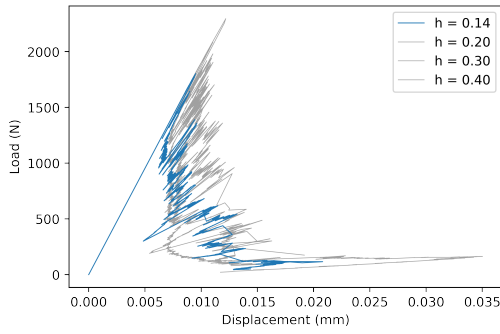
(d) The skeleton curve for  $h = 0.40$  mm at  $t = t_{crack} = 304$

Figure 4.10: The skeleton curve at  $t = t_{crack}$  for different mesh sizes

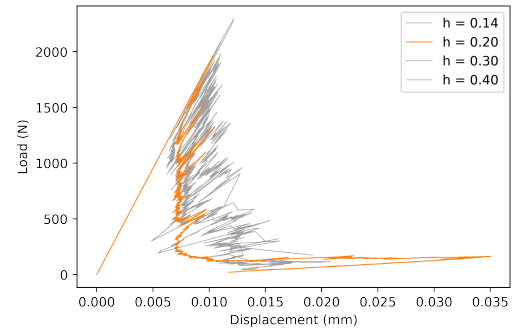


In order to compare the results for different mesh sizes as well, the skeleton curve has been plotted for different mesh sizes in figure 4.10. It is directly apparent that depending on the mesh size, different crack patterns are found. In particular, the location(s) at which the crack moves from the bottom face to the top face vary greatly. This result is not unexpected, since the location at which the initial crack start to form is ultimately up to chance. Once a crack has started to form, however, stress concentrations will start to appear at the end points of the crack, which causes it to continue to grow. The resulting energy release in turn causes relaxation throughout the rest of the material. These two factors make it more likely for a crack to continue to grow than for new cracks to form.

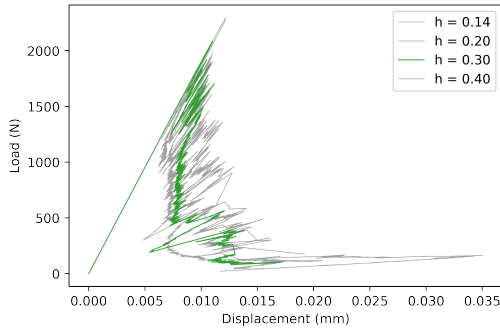
Finally, the load-displacement diagrams are compared for different mesh sizes in figure 4.11. As stated before, the results of this model are not yet physically accurate. However, it can still be useful to validate the internal consistency of the model, by checking the mesh sensitivity of the results. Based on visual inspection, it can be said that the choice of mesh size does not have a large effect on the load-displacement diagrams. Only for  $h = 0.40$  mm, somewhat larger loads and displacements are found before snapback occurs, though this could simply be due to the fact that an  $0.40$  mm mesh is relatively coarse for this problem.



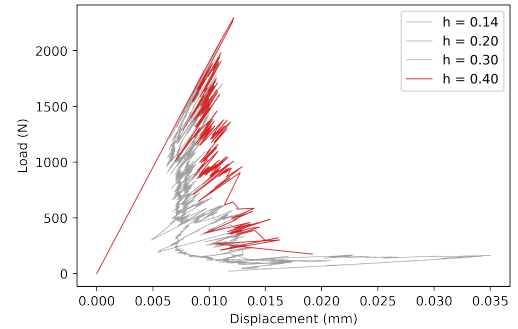
(a) The load-displacement diagram of the rail shear test with  $h = 0.14$  mm



(b) The load-displacement diagram of the rail shear test with  $h = 0.20$  mm



(c) The load-displacement diagram of the rail shear test with  $h = 0.30$  mm



(d) The load-displacement diagram of the rail shear test with  $h = 0.40$  mm

Figure 4.11: The load-displacement diagram for different mesh sizes

## Chapter 5

# Conclusion

### 5.1 Aim and research questions

As stated in the introduction, the aim of this thesis is to create a basis upon which an implementation of TLS\_V2 can be built. For any implementation of TLS\_V2, it is necessary to have an explicitly defined discontinuity in the displacement field, which is located on the skeleton of the iso-0 curve. To accomplish this, the three research questions posed in the introduction have been answered and will be summarized below.

*How can the skeleton curve be defined for a given damage front?*

In chapter 3, it has been shown that the Shrinking Ball Algorithm by Ma et al. [3] and Peters [29] can be used to find a set of atoms that lie on the skeleton curve. This set of atoms can then be connected using Prim's algorithm [4], which finds the shortest spanning tree for a given set of points. This shortest spanning tree is identical to the skeleton curve, provided that the distance between atoms is sufficiently small. The proposed procedure is able to handle branching and merging of damage fields well. However, for damage fields that are not acyclical, the connectivity of the skeleton curve will not be entirely accurate. This is due to the fact that the shortest spanning tree of a set of points will never contain any cycles, and thus the topological properties of the iso-0 curve are not preserved.

*How can the skeleton curve be discretised for a given 2D mesh consisting of triangular elements?*

In order to link the skeleton curve to the elements it intersects, it is necessary to map the skeleton curve onto the mesh. In a sense, this yields a discretized version of the skeleton curve. In chapter 3, an approach is presented that organizes the skeleton curve and maps it onto the mesh. By essentially cutting up the skeleton curve at its junctions, and handling each of its straight parts separately, the discretized skeleton curve can easily be found. Junction and sharp corners are given special attention, and modified in such a way that the discretized skeleton curve does not intersect any element edge more than once. During the validation in chapter 4, no skeleton curves were found for which the proposed implementation did not yield the intended results.

*How can the displacement jump be explicitly modeled on the skeleton curve?*

Due to its applicability to Fibre-Reinforced Composites, the Phantom Node Method by Hansbo and Hansbo [5, 33] is used to define the displacement jump on the skeleton curve. This method requires the definition of phantom elements, which replace the elements that are intersected by the skeleton curve. In chapter 3, an algorithm is proposed via which the

phantom nodes and phantom elements can be determined. This algorithm iterates over the crack pattern from end point to end point, while turning right whenever a junction is encountered. Provided that no cycles occur in the discretized skeleton curve, this approach will have iterated over each edge exactly twice when it has returned to its starting point. Furthermore, any junction where  $n$  edges meet will have been encountered  $n$  times. This means that no additional measures need to be taken to ensure that the element located at this junction will be replaced by  $n$  phantom elements. Again, no skeleton curves have been found during validation for which the implementation did not produce the correct results.

Due to the modification of the mesh, it is necessary to carefully keep track of the elements that have been added and removed. To conveniently translate from the original mesh to the updated mesh and vice versa, two mappings have been proposed. The mapping from the original to the updated mesh can be used to create a set containing all elements in the updated mesh. Then, the mapping from the updated to the original mesh can be used to create a set of the same size, but containing the corresponding original elements. These two sets can then be used to conveniently switch back and forth between the updated and original mesh, depending on which is needed.

## 5.2 Limitations and recommendations

It should be noted that the proposed algorithms have only been implemented and verified for 2D triangular meshes. In order to apply the proposed algorithm to a quadrilateral mesh, some modifications will be required, particularly in the way the iso-0 curve is determined based on the level set field. However, there are no fundamental issues that would make a general 2D implementation of the proposed algorithms impossible. Generalizing into 3D, on the other hand, poses considerable problems that do not have a straightforward solution. Since a lot of procedures rely on the fact that a 2D crack pattern can be represented as a graph, where each edge corresponds to (a part of) a crack. In 3D, this is not possible, because a crack in a three-dimensional material is represented as a surface in 3D space, rather than a line in 2D space. It is uncertain whether modifications to the proposed algorithms could be applied to make them support 3D models as well, or whether completely new solutions would need to be found.

Another shortcoming of the proposed implementation is that it does not handle cyclical damage fields well. The first issue for cyclical damage fields occurs when the atoms are connected using Prim's algorithm, which does not preserve the cycles in the damage field. In chapter 4, a solution has been proposed, which could add the missing segments to the skeleton curve. This solution would check for each end point whether another atom can be found in its vicinity that would form an angle of more than, for example,  $135^\circ$  with the end point atom. If this solution—or any other solution that would yield a cyclical skeleton curve—is implemented, algorithms that follow will have to be modified as well to cope with the cycles in the skeleton curve. Especially the iterative procedures over the crack pattern will have to be updated to ensure that the cycles are not skipped because they don't contain an end point. Regardless, these modifications will be relatively small and easy to implement. Often, this only entails a repetition of the iterative procedure for each cycle.

Lastly, it should be mentioned that the functions and procedures that have been proposed only form a basis upon which TLS\_V2 can be built. No attention has been paid to the shape functions of the phantom elements, correcting for the energy dissipation through the interfacial damage progression, and including traction forces at the location of the displacement jump. Although this thesis does demonstrate that it is possible to create

a general implementation of TLS\_V2, and offers a framework upon which this can be done, the results found so far are not physically accurate. This means that, at the moment, the work that has been presented in this thesis cannot yet be applied in practical situations. In order to do so, the main priority should be to include the interfacial damage as a function of the level set field in the model, and modify the expressions for the configurational force  $g(s)$  and averaged energy release rate  $\bar{Y}(s)$  accordingly.

### 5.3 Contribution to the field

In spite of the fact that the model proposed in this thesis does not yet produces physically accurate results, it does provide a robust basis for an implementation of TLS\_V2 in 2D. It has provided a method by which the skeleton curve can be found for arbitrary crack patterns, and it has shown how this skeleton curve can be discretized for any given mesh. Additionally, a method has been presented by which the displacement jump on the skeleton curve can be explicitly modeled, and it has been demonstrated how this method can be implemented for arbitrary crack patterns.

# Appendix A

## Program listings

Listing A.1: Pseudocode for the getIso0Elems function. This function takes the mesh as input and returns a set of elements that are intersected by the iso-0 curve.

```
set <element_t> getIso0Elems
( set <element_t> elems )
{
    // Initialize iso0Elems
    set <element_t> iso0Elems ;

    // Loop over all elements
    for ( idx_t ielem = 0 ; ielem < elems.size () ; ielem++ ){
        // Get the element nodes
        set <node_t> nodes = elems[ielem].getNodes () ;

        bool hasPos = false ;
        bool hasNeg = false ;

        // Check for each node if the level set field is positive or negative
        for ( idx_t inode = 0 ; inode < nodes.size () ; inode++ ){
            if ( nodes[inode].getLevelSet () >= 0 ){
                hasPos = true ;
            } else {
                hasNeg = true ;
            }
        }

        // Add the element to iso0Elems, if both positive and negative
        if ( hasPos == true and hasNeg == true ){
            iso0Elems.pushBack ( elems[ielem] ) ;
        }
    }

    // Return iso0Elems
    return iso0Elems
}
```

Listing A.2: Pseudocode for the `addSegmentsBackward` function. It performs the same function as the `addSegmentsForward` function in listing 3.2, but adds segments to the front of the linestring rather than to the back. Also, it uses the `getNodePosNeg` function instead of the `getNodeNegPos` function, to retrieve the next element. Note that, if the `addSegmentsForward` function has already been excuted, this function only needs to be run if the linestring is not a closed loop yet.

```

void addSegmentsBackward
( lstring_t          iso0 ,
  element_t         elemFirst
  element_t         elemLast )
{
  // Iterate backward until no elements are found, or the loop is closed
  while ( true ){
    // Get the nodes where the level set field goes from < 0 to >= 0
    pair <node_t, node_t> nodePosNeg = getNodePosNeg (elem_first) ;
    node_t neg = nodePosNeg.first ;
    node_t pos = nodePosNeg.second ;

    // Find the neighboring element
    set <element_t> elemsNext = getNextElements (elemLast, neg, pos) ;

    // [Verify that elemsNext contains no more than 1 element]

    if ( elemsNext.size() == 0 ){
      // The end of the linestring is found, so exit the loop
      break ;
    } else {
      // Add the segment to the linestring
      segment_t segment = elemsNext[0].getIntersection () ;
      iso0.insert (iso0.begin (), segment) ;

      if ( elemsNext[0] == elemLast ){
        // The linestring has been closed, so exit the loop
        break ;
      } else {
        // Update the first element
        elemFirst = elemsNext[0] ;
      }
    }
  }
}

```

Listing A.3: Pseudocode for the getNextElements function. It takes an element and two of its nodes as input, and returns a vector containing all elements in the mesh that share both nodes with the source element elem0

```
/*
 * getNextElements
 *
 * Description:
 * This function finds the neighboring element for a given element and two
 * of its nodes
 */

set <element_t> getNextElements

( element_t      elem0 ,
  node_t        node1 ,
  node_t        node2 )

{
  // Initialize the set of neighboring elements for output
  set <element_t> elemsNext ;

  // [Check if node_1 and node_2 are indeed element nodes]

  // Get all possible neighboring elements
  set <element_t> possibleNeighbors = node_1.getElements () ;

  // Check which of the possible neighbors also contain node_2
  for ( idx_t ie = 0 ; ie < possibleNeighbors.size () ; ie++ ){

    // Get the nodes of each possible neighbor
    element_t possibleElem = possibleNeighbors[ie] ;
    set <node_t> nodes = possibleElem.getNodes () ;

    // Check if one of the nodes is node_2
    for ( idx_t in = 0 ; in < nodes.size() ; in++ ){
      if ( nodes[in] == node_2 ){
        // If so, add next_elem to the set of neighboring elements
        elemsNext.pushBack (possibleElem) ;
      }
    }
  }

  // Return the set of neighbouring elements
  return elemsNext ;
}
```

Listing A.4: Pseudocode for the `getNodeNegPos` and `getNodePosNeg` functions. These functions loop around the nodes of the input element in a counterclockwise direction, and return the pair of nodes where the level set field value goes from negative to positive, or negative to positive, respectively. For a T3 mesh, there can be no more than 1 solution, although this is not the case for quadrilaterals or higher-order elements

```

pair <node_t, node_t> getNodeNegPos
( element_t      elem )
{
    set <node_t> nodes = elem.getNodes () ;

    // Loop over the nodes of the element
    for ( idx_t in = 0 ; in < nodes.size() ; in++ ){

        // Set jn to the node after in
        idx_t jn = (in + 1) % nodes.size() ;

        if ( nodes[in].getLevelSet () >= 0 ){
            if ( nodes[jn].getLevelSet () < 0 ){
                // Return nodes in and jn
                return makePair (nodes[in], nodes[jn]) ;
            }
        }
    }

    // [Verify that a possible pair has been found]
}

pair <node_t, node_t> getNodePosNeg
( element_t      elem )
{
    set <node_t> nodes = elem.getNodes () ;

    // Loop over the nodes of the element
    for ( idx_t in = 0 ; in < nodes.size() ; in++ ){

        // Set jn to the node after in
        idx_t jn = (in + 1) % nodes.size() ;

        if ( nodes[in].getLevelSet () < 0 ){
            if ( nodes[jn].getLevelSet () >= 0 ){
                // Return nodes in and jn
                return makePair (nodes[in], nodes[jn]) ;
            }
        }
    }

    // [Verify that a possible pair has been found]
}

```



Listing A.5: Pseudocode for the `edgeInPattern` function. This function checks if a given edge appears in the crack pattern. It returns `true` if that is the case, and `false` if not.

```
bool edgeInPattern
(
    edge_t      edge ,
    set<set<idx_t>> pattern )
{
    // Get the source and target vertex of the edge
    vertex_t source = edge.sourceVertex ;
    vertex_t target = edge.targetVertex ;

    // Loop over the cracks in the pattern
    for ( idx_t i = 0 ; i < pattern.size () ; i++ ){

        set<idx_t> crack = pattern[i] ;

        // Loop over the indices in each crack
        for ( idx_t j = 0 ; j < crack.size () - 1 ; j++ ){

            // Check if the source and target are found
            if ( source.index == crack[j] and target.index == crack[j+1] ){
                return true ;
            } else if ( target.index == crack[j] and source.index == crack[j+1] ){
                return true ;
            }
        }
    }

    // return false if no match is found
    return false ;
}
```

Listing A.6: Pseudocode for the `addAllSegments` and `addAllSegmentsExcept` functions. These functions are used by the `makeShortCuts` function in listing 3.8 to add all possible internal segments to an element. The `addAllSegmentsExcept` will skip any edge pointing to the `exceptElement`.

```

void addAllSegments
( vertex_t          element )
{
    // Remove all existing segments
    element.segments.clear () ;

    set <edge_t> intxns = element.edges ;

    // Loop over the edges of the element twice
    for ( idx_t io = 0 ; io < intxns.size () ; io++ ){
        for ( idx_t ip = io+1 ; ip < intxns.size () ; ip++ ){
            segment_t elemSegment (intxns[io].coords , intxns[ip].coords) ;
            element.segments.pushBack (elemSegment) ;
        }
    }
}

void addAllSegmentsExcept
( vertex_t          element ,
  vertex_t          exceptElement )
{
    // Remove all existing segments
    element.segments.clear () ;

    set <edge_t> intxns = element.edges ;

    // Loop over the edges of the element twice
    for ( idx_t io = 0 ; io < intxns.size () ; io++ ){
        // Skip if the exception element is found
        if ( io.targetElement == exceptElement ) continue ;

        for ( idx_t ip = io+1 ; ip < intxns.size () ; ip++ ){
            // Skip if the exception element is found
            if ( ip.targetElement == exceptElement ) continue ;

            segment_t elemSegment (intxns[io].coords , intxns[ip].coords) ;
            element.segments.pushBack (elemSegment) ;
        }
    }
}

```

Listing A.7: Pseudocode for the `getAngle` function. This function calculates the angle from point A to point B to point C, and returns a value between  $-\pi$  and  $\pi$ .

```
double getAngle
( point_t      A ,
  point_t      B ,
  point_t      C )
{
    // Find the angle based on teh atan2 function
    double angle = atan2 (C.y-B.y, C.x-B.x) - atan2 (B.y-A.y, B.x-A.x) ;

    // Add or subtract 2 pi if the angle falls outside of the range
    if ( angle <= -pi () ){
        angle += 2 * pi () ;
    } else if ( angle > pi () ){
        angle -= 2 * pi () ;
    }

    // Return the angle
    return angle ;
}
```

Listing A.8: Pseudocode for the `updateElements` and `getOriginalElements` functions. The `updateElements` function converts `ielems_0` to `ielems_i`, and the `getOriginalElements` converts `ielems_i` to `ielems_0_i`, as explained in the section above.

```

set <idx_t> updateElements

( set <idx_t>          ielems_0 ,
  mappingOldToNew_t    mapOldToNew )

{
  // Initialize the ielems_i
  set <idx_t> ielems_i ;

  // Add the updated elements from the mapping to the ielems_i
  for ( idx_t i = 0 ; i < ielems_0.size () ; i++ ){
    set <idx_t> newElements = mapOldToNew.find (ielems_0[i]) ;

    for ( idx_t j = 0 ; j < newElements.size () ; j++ ){
      ielems_i.pushBack (newElements[j]) ;
    }
  }

  // Return the ielems_i
  return ielems_i ;
}

set <idx_t> getOriginalElements

( set <idx_t>          ielems_i ,
  mappingNewToOld_t    mapNewToOld )

{
  // Initialize the ielems_0_i
  set <idx_t> ielems_0_i ;

  // Add the original elements from the mapping to the ielems_0_i
  for ( idx_t i = 0 ; i < ielems_i.size () ; i++ ){
    idx_t oldElement = mapNewToOld.find (ielems_i[i]) ;
    ielems_0_i.pushBack (oldElement) ;
  }

  // Return the ielems_0_i
  return ielems_0_i ;
}

```

# Bibliography

- [1] N. Moës, C. Stolz, P.-E. Bernard, and N. Chevaugeon, “A Level Set Based Model for Damage Growth: The Thick Level Set Approach,” *International Journal for Numerical Methods in Engineering*, vol. 86, pp. 358–380, Dec 2010. doi:10.1002/nme.3069
- [2] B. Lé, N. Moës, and G. Legrain, “Coupling Damage and Cohesive Zone Models with the Thick Level Set Approach to Fracture,” *Engineering Fracture Mechanics*, vol. 193, pp. 214–247, Apr 2018. doi:10.1016/j.engfracmech.2017.12.036
- [3] J. Ma, S. W. Bae, and S. W. Choi, “3D Medial Axis Point Approximation Using Nearest Neighbours and the Normal Field,” *The Visual Computer*, vol. 28, pp. 7–19, Jan 2012. doi:10.1007/s00371-011-0594-7
- [4] R. C. Prim, “Shortest Connection Networks and Some Generalizations,” *Bell System Technical Journal*, vol. 36, pp. 1389–1401, Nov 1957. doi:10.1002/j.1538-7305.1957.tb01515.x
- [5] A. Hansbo and P. Hansbo, “A Finite Element Method for the Simulation of Strong and Weak Discontinuities in Solid Mechanics,” *Computational Methods in Applied Mechanics and Engineering*, vol. 193, pp. 3523–3540, Aug 2004. doi:10.1016/j.cma.2003.12.041
- [6] P.-E. Bernard, N. Moës, and N. Chevaugeon, “Damage Growth Modeling Using the Thick Level Set (TLS) Approach: Efficient Discretization for Quasi-Static Loadings,” *Computer Methods in Applied Mechanics and Engineering*, vol. 233, pp. 11–27, Aug 2012. doi:10.1016/j.cma.2012.02.020
- [7] F. P. van der Meer and L. J. Sluys, “The Thick Level Set Method: Sliding Deformations and Damage Initiation,” *Computer Methods in Applied Mechanics and Engineering*, vol. 285, pp. 64–82, Mar 2015. doi:10.1016/j.cma.2014.10.020
- [8] J. A. Sethian, “Theory and Algorithms and Applications of Level Set Methods for Propagating Interfaces,” *Acta Numerica*, vol. 5, pp. 309–395, Jan 1996. doi:10.1017/S0962492900002671
- [9] —, “A Fast Marching Level Set Method for Monotonically Advancing Fronts,” *Proceedings of the National Academy of Sciences of the United States of America*, vol. 93, pp. 1591–1595, Feb 1996. doi:10.1073/pnas.93.4.1591
- [10] S. Osher and J. A. Sethian, “Fronts Propagating with Curvature Dependent Speed,” *Journal of Computational Physics*, vol. 79, pp. 12–49, Nov 1988. doi:10.1016/0021-9991(88)90002-2
- [11] G. Allaire, F. Jouve, and A.-M. Toader, “A Level-Set Method for Shape Optimization,” *Comptes Rendus Mathématique*, vol. 334, pp. 1125–1130, Apr 2002. doi:10.1016/S1631-073X(02)02412-3

- [12] —, “Structural Optimization Using Sensitivity Analysis and a Level-Set Method,” *Journal of Computational Physics*, vol. 194, pp. 363–393, Feb 2004. doi:10.1016/j.jcp.2003.09.032
- [13] —, “Structural Optimization Using Topological and Shape Sensitivity via a Level-Set Method,” *Comptes Rendus Mathématique*, vol. 334, pp. 1125–1130, Apr 2002. doi:10.1016/S1631-073X(02)02412-3
- [14] G. Allaire, F. Jouve, and N. van Goethem, “A Level-Set Method for the Numerical Simulation of Damage Evolution,” *International Congress on Industrial and Applied Mathematics*, vol. 1, pp. 3–22, Dec 2007. doi:10.4171/056-1/1
- [15] F. P. van der Meer and L. J. Sluys, “Continuum Models for the Analysis of Progressive Failure in Composite Laminates,” *Journal of Composite Materials*, vol. 40, pp. 2131–2156, Aug 2009. doi:10.1177/0021998309343054
- [16] —, “A Phantom Node Formulation with Mixed Mode Cohesive Law for Splitting in Laminates,” *International Journal of Fracture*, vol. 158, pp. 107–124, May 2009. doi:10.1007/s10704-009-9344-5
- [17] L. A. T. Mororó and F. P. van der Meer, “Combining the Thick Level Set Method with Plasticity,” *European Journal of Mechanics - A/Solids*, vol. 79, Jan 2020. doi:10.1016/j.euromechsol.2019.103857
- [18] L. A. T. Mororó, “Parallel Computing with the Thick Level Set (TLS) Method,” Feb 2020, used this paper while it was still being written.
- [19] F. P. van der Meer and L. J. Sluys, “A Level Set Model for Delamination – Modeling Crack Growth without Cohesive Zone or Stress Singularity,” *Engineering Fracture Mechanics*, vol. 79, pp. 191–212, Jan 2012. doi:10.1016/j.engfracmech.2011.10.013
- [20] M. Latifi and L. J. Sluys, “A Thick Level Set Interface Model for Simulating Delamination in Composites,” *International Journal for Numerical Methods in Engineering*, vol. 111, pp. 303–324, Nov 2016. doi:10.1002/nme.5463
- [21] J. A. Sethian, “Fast Marching Methods and Level Set Methods for Propagating Interfaces,” *Proceedings of the National Academy of Sciences of the United States of America*, vol. 93, pp. 1591–1595, Feb 1996. doi:10.1073/pnas.93.4.1591
- [22] K. Moreau, N. Moës, and N. Chevaugeon, “Concurrent Development of Local and Non-Local Damage with the Thick Level Set Approach: Implementation Aspects and Application to Quasi-Brittle Failure,” *Computer Methods in Applied Mechanics and Engineering*, vol. 327, pp. 306–326, Dec 2017. doi:10.1016/j.cma.2017.08.045
- [23] P. K. Saha, G. Borgefors, and G. S. di Baja, *Skeletonization: Theory and Methods and Application*, 1st ed. Academic Press, Jan 2017. ISBN 9780081012918
- [24] H. F. Blum, “A Transformation for Extracting New Descriptors of Shape,” in *Models for the Perception of Speech and Visual Form*, W. Warthen-Dunn, Ed. MIT Press, Nov 1967, pp. 362–380. ISBN 9780262230261
- [25] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 2nd ed. Prentice Hall, Jan 2002. ISBN 9780201180756
- [26] E. C. Sherbrooke, N. M. Patrikalakis, and F.-E. Wolter, “Differential and Topological Properties of Medial Axis Transforms,” *Graphic Models and Image Processing*, vol. 58, pp. 574–592, Nov 1996. doi:10.1006/gmip.1996.0047

- [27] D. Shaked and A. M. Bruckstein, “Pruning Medial Axes,” *Computer Vision and Image Understanding*, vol. 69, pp. 156–169, Feb 1998. doi:10.1006/cviu.1997.0598
- [28] J. H. Elder, T. Oleskiw, A. Yakubovich, and G. Peyré, “On Growth and Formlets: Sparse Multi-Scale Coding of Planar Shape,” *Image and Vision Computing*, vol. 1, pp. 1–13, Jan 2013. doi:10.1016/j.imavis.2012.11.002
- [29] R. Y. Peters, “Geographical Point Cloud Modelling with the 3D Medial Axis Transform,” Ph.D. dissertation, Delft University of Technology, Mar 2018.
- [30] S. W. Choi, “On the Stability of Medial Axis Transform,” *Journal of Applied Mathematics and Computing*, vol. 23, pp. 419–433, Jan 2007. doi:10.1007/BF02831988
- [31] A. S. Montero and J. Lang, “Skeleton Pruning by Contour Approximation and the Integer Medial Axis Transform,” *Computers & Graphics*, vol. 36, pp. 477–487, Aug 2012. doi:10.1016/j.cag.2012.03.029
- [32] J. August, K. Siddiqi, and S. W. Zucker, “Ligature Instabilities in the Perceptual Organization of Shape,” *Computer Vision and Image Understanding*, vol. 76, pp. 231–243, Dec 1999. doi:10.1006/cviu.1999.0802
- [33] A. Hansbo and P. Hansbo, “An Unfitted Finite Element Method and Based on Nitsche’s Method and for Elliptic Interface Problems,” *Computational Methods in Applied Mechanics and Engineering*, vol. 191, pp. 5537–5552, Nov 2002. doi:10.1016/S0045-7825(02)00524-8
- [34] B. Y. Chen, S. T. Pinho, N. V. de Carvalho, P. M. Báiz, and T.-E. Tay, “A Floating Node Method for the Modeling of Discontinuities in Composites,” *Engineering Fracture Mechanics*, vol. 127, pp. 104–134, Sep 2014. doi:10.1016/j.engfracmech.2014.05.018
- [35] J. Mergheim, E. Kuhl, and P. Steinmann, “A Finite Element Method for the Computational Modeling of Cohesive Cracks,” *International Journal for Numerical Methods in Engineering*, vol. 63, pp. 3523–3540, Feb 2005. doi:10.1016/j.cma.2003.12.041
- [36] J.-H. Song, P. M. A. Arelas, and T. Belytschko, “A Method for Dynamic Crack and Shear Band Propagation with Phantom Nodes,” *International Journal for Numerical Methods in Engineering*, vol. 67, pp. 868–893, Feb 2006. doi:10.1002/nme.1652
- [37] T. Rabczuk, G. Zi, A. Gerstenberger, and W. A. Wall, “A New Crack Tip Element for the Phantom-Node Method with Arbitrary Cohesive Cracks,” *International Journal for Numerical Methods in Engineering*, vol. 75, pp. 577–599, Jan 2008. doi:10.1002/nme.2273
- [38] T. Chau-Dinh, G. Zi, P.-S. Lee, T. Rabczuk, and J.-H. Song, “Phantom-Node Method for Shell Models with Arbitrary Cracks,” *Computers and Structures*, vol. 92–93, pp. 242–256, Feb 2012. doi:10.1016/j.compstruc.2011.10.021
- [39] N. Vu-Bac, H. Nguyen-Xuan, L. Chen, C. K. Lee, G. Zi, X. Zhuang, G. R. Liu, and T. Rabczuk, “A Phantom-Node Method with Edge-Based Strain Smoothing for Linear Elastic Fracture Mechanics,” *Journal of Applied Mathematics*, vol. 2013, pp. 1–12, Jul 2013. doi:10.1155/2013/978026
- [40] J.-H. Song, P. Lea, and J. Oswald, “Explicit Dynamic Finite Element Method for Predicting Implosion/Explosion Induced Failure of Shell Structures,” *Computational Methods for Fracture*, vol. 2013, pp. 1–11, Oct 2013. doi:10.1155/2013/957286

- [41] C. E. Rogers, E. S. Greenhalgh, and P. Robinson, “Developing a Mode II Fracture Model for Composite Laminates,” Jun 2008, from the 13th European Conference on Composite Materials. url:[https://extra.ivf.se/eccm13\\_programme/abstracts/514.pdf](https://extra.ivf.se/eccm13_programme/abstracts/514.pdf)
- [42] E. S. Greenhalgh, C. E. Rogers, and P. Robinson, “Fractographic Observations on Delamination Growth and the Subsequent Migration through the Laminate,” *Composites Science and Technology*, vol. 69, pp. 2345–2351, Nov 2009. doi:10.1016/j.compscitech.2009.01.034
- [43] ASTM C273-20, “Standard Test Method for Shear Properties of Sandwich Core Materials,” ASTM International, Tech. Rep., 2020.
- [44] ISO 7539-6:2018, “Corrosion of Metals and Alloys — Stress Corrosion Testing — Part 6: Preparation and Use of Precracked Specimens for Tests under Constant Load or Constant Displacement,” International Organisation for Standardization, Tech. Rep., 2018. url:<https://www.iso.org/obp/ui/#iso:std:iso:7539:-6:ed-4:v2:en>
- [45] ASTM E647-15, “Standard Test Method for Measurement of Fatigue Crack Growth Rates,” ASTM International, Tech. Rep., 2015.
- [46] C. Geuzalne and J.-F. Remacle, “Gmsh: A 3-D Finite Element Mesh Generator with Built-In Pre- and Post-Processing Facilities,” *International Journal for Numerical Methods in Engineering*, vol. 79, pp. 1309–1331, May 2009. doi:10.1002/nme.2579
- [47] S. Li, M. D. Thouless, A. M. Waas, J. A. Schroeder, and P. D. Zavattieri, “Use of a Cohesive-Zone Model to Analyze the Fracture of a Fibre-Reinforced Polymer-Matrix Composite,” *Composites Science and Technology*, vol. 65, pp. 537–549, Mar 2005. doi:10.1016/j.compscitech.2004.08.004