UNIVERSITY OF COLORADO BOULDER

A complete methodology for the implementation of XFEM inclusion models

by

Hernan Villanueva

College of Engineering and Applied Science
Department of Mechanical Engineering

June 2013

UNIVERSITY OF COLORADO BOULDER

Abstract

College of Engineering and Applied Science
Department of Mechanical Engineering

by Hernan Villanueva

This report offers a background into the eXtended Finite Element Method to solve the shortcomings of the classical Finite Element Method, such as the numerical solution to problems with different material topologies by using level-set functions to track the location of these discontinuities. The report also provides algorithms for locating these discontinuities and subsequently dividing the domain into sub-domains capable of integration. This report ultimately expounds upon how to effectively apply the local enrichment functions that the XFEM standard approximation requires at the nodes where the discontinuities occur.

Contents

Abstract **i**

List of Figures **iv**

List of Tables **v**

Abbreviations **vi**

1 XFEM: Background **1**
- 1.1 FEM . . . 1
- 1.2 Discontinuity . . . 1
- 1.3 eXtended Finite Element Method . . . 2
  - 1.3.1 Level-set method . . . 2
  - 1.3.2 Delaunay triangulation . . . 3

2 XFEM implementation - Serial **6**
- 2.1 Glossary . . . 6
- 2.2 Introduction . . . 7
- 2.3 Summary . . . 7
- 2.4 Overall Procedure . . . 7
- 2.5 XFEM Implementation Algorithms . . . 8
  - 2.5.1 Point-to-Cells connectivity table . . . 11
  - 2.5.2 Edge table . . . 11
  - 2.5.3 Element clusters . . . 13
  - 2.5.4 Nodal clusters . . . 14
  - 2.5.5 Intersection points . . . 14
  - 2.5.6 Delaunay triangulation and assignment of main and sub-phases to pseudo-elements . . . 15
  - 2.5.7 Main phase and sub-phase . . . 16
  - 2.5.8 Nodal enrichments for pseudo-elements . . . 17
  - 2.5.9 Enrichment of higher order element nodes . . . 20
  - 2.5.10 Determine degrees of freedom used . . . 21
  - 2.5.11 Implementation Example 2 . . . 23
  - 2.5.12 Implementation Example 3 . . . 24
- 2.6 Solving the problem . . . 26
- 2.7 Preconditioner . . . 27

3 XFEM implementation - Parallel **29**
3.1 Glossary . . . 29
3.2 Introduction . . . 29
3.2.1 Parallel Communicate Sub-Phase Information . . . 30
3.2.2 Parallel Communicate Enriched Global DOFs . . . 31

4 XFEM corroboration and results **33**
4.1 Methodology . . . 33
4.2 Tests . . . 34
4.2.1 Mesh refinement sweep . . . 34
4.2.2 Conductivity ratio sweep . . . 35
4.2.3 Condition number comparison . . . 36

5 Conclusions **39**
5.1 Conclusions . . . 39

A Delaunay Triangulation code **40**
A.1 main.m . . . 40
A.2 xfem8isct.m . . . 42
A.3 xfem8tet.m . . . 43
A.4 number_configurations.m . . . 50

Bibliography **53**

List of Figures

1.1 Level-set method . . . 3
1.2 Delaunay triangulation . . . 4
1.3 Delaunay circumcircles . . . 4
1.4 2D triangulation configurations . . . 5
1.5 3D triangulation example . . . 5

2.1 XFEM Overall Procedure . . . 9
2.2 XFEM implementation example physical model . . . 10
2.3 XFEM implementation example discrete model . . . 10
2.4 Edge discretization of XFEM model example . . . 13
2.5 Intersection point computation . . . 15
2.6 Sub-phase computation algorithm . . . 16
2.7 Enrichment of middle nodes . . . 21
2.8 XFEM implementation example 2 . . . 23
2.9 XFEM implementation example 3 . . . 24

3.1 XFEM Overall Procedure in Parallel . . . 30

4.1 Thermal problem setup . . . 34
4.2 Mesh refinement sweep interface error . . . 35
4.3 Mesh refinement sweep L2 error . . . 36
4.4 Conductivity refinement sweep interface error . . . 37
4.5 Conductivity refinement sweep L2 error . . . 37
4.6 Condition number comparison - no pre-conditioner . . . 38
4.7 Condition number comparison - with pre-conditioner . . . 38

List of Tables

2.1 Point ID to cell IDs connectivity table 11
2.2 Edge to point and cell IDs connectivity table 12
2.3 Main-phase and sub-phase table for pseudo-elements 17
2.4 Initial node-element table 18
2.5 Flipping the enrichment levels 19
2.6 Final node-element table 20
2.7 Element to enrichment table 20
2.8 Initial node-element table configuration 2 23
2.9 Final node-element table configuration 2 24
2.10 Element to enrichment table configuration 2 24
2.11 Initial node-element table configuration 3 25
2.12 Final node-element table for configuration 3 25
2.13 Element to enrichment table configuration 3 25

Abbreviations

FEM	**F**inite **E**lement **M**ethod
XFEM	e**X**tended **F**inite **E**lement **M**ethod
LSM	**L**evel **S**et **M**ethod

Chapter 1

XFEM: Background

1.1 FEM

The Finite Element Method (FEM) is a numerical technique used for finding solutions to partial differential equations as well as to integral equations. Traditional Finite Element Methods (FEM) requires meshing techniques that generate discrete representation of potentially complex geometry. Difficulties arise when using the traditional FEM for analyzing fracture mechanics: under traditional FEM, introducing a discontinuity in the mesh, changing the material topology or drastically changing the shape of the material requires a new mesh to ensure that the element edges align with the discontinuity [1]. The process is laborious and difficult [2]. The eXtended Finite Element Method (XFEM) arose as a solution to this impediment by applying enrichment functions at the position of the material interface or topology discontinuity instead of re-meshing the entire structure.

1.2 Discontinuity

A discontinuity can be defined as a rapid change in a field variable over a length negligible in size compared to the entire domain analyzed [1]. For example, in solid mechanics, strain and stress fields are discontinuous across material interfaces and displacements are discontinuous at cracks; in fluid dynamics, velocity and pressure fields are discontinuous at the boundary layer between two fluids. A discontinuity is classified as "weak" or "strong". Weak discontinuities happen when a field variable (stress field, strain rate, etc.) has a kink, meaning the derivative has a jump. This can happen at boundary layers, or at a material or fluid interface. Strong discontinuities happen when the field quantity has a jump; this could include, for example, a crack in the structure [3].

1.3 eXtended Finite Element Method

The XFEM arose as a numerical technique capable of providing local enrichment functions at the position of the material interface, avoiding the need to re-mesh the entire structure while finding solutions for the discontinuous functions [4]. By doing this, XFEM appears to solve the shortcomings of the FEM by providing accurate solutions for complex problems in engineering that would be impossible to solve otherwise [1].

1.3.1 Level-set method

Level set functions are used to model the motion of these discontinuities in the elements. A level set function is a numerical scheme where the discontinuity of interest is represented as the zero level set function. Basically, a level set function has a value of zero at the boundary of its closed curve and opposite signs on the interior and the exterior (figure 1.1). The XFEM uses this method by placing the discontinuity at the boundary layer and giving the interface caused by the division positive and negative values. Since the zero level is interpreted as the discontinuity, new nodes (called pseudo-nodes) are created at this position and the enrichment of the FEM is produced at this location. This creates an advantage because instead of re-meshing the entire structure, a fixed Cartesian grid is used to divide the structure and the discontinuity into domains capable of integration. Once this mesh is settled, the XFEM will subsequently use either branched or Heaviside functions to enrich the nodes and solve for the problem.

The Level-Set Method (LSM) and the XFEM have a sort of natural coupling to solve problems with discontinuities. While the LSM is used to model the discontinuity and update its motion at each calculation, the XFEM is used to solve the problem and determine the direction of the discontinuity [5].

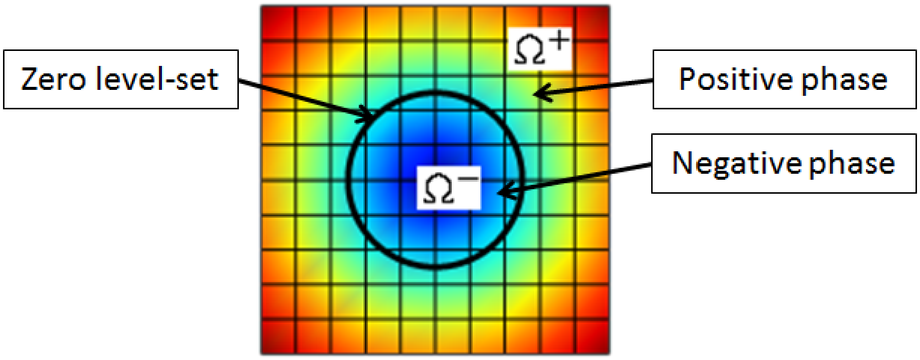

FIGURE 1.1: Level-Set Method (The discontinuity is defined as the zero level-set function).

1.3.2 Delaunay triangulation

In order to apply the LSM and the XFEM to solve a problem, a framework for dividing arbitrarily complex geometries into integrable domains must be developed. The Delaunay triangulation is a critical step in this process. The Delaunay triangulation is the subdivision of a geometric object into triangles for 2D geometry and tetrahedra for 3D (figure 1.2). This particular triangulation has the property that the circumcircle of any triangle in the triangulation does not contain the vertices of other triangles or its own in its interior (figure 1.3) [6]. Because triangle and tetrahedra are integrable elements, the XFEM method can then be applied.

For an inclusion-based XFEM model (inclusion meaning the zero level function is always a closed curve), there are only 6 triangulation configurations in 2D, while 127 different ones in 3D. Due to the low number of cases in 2D, a tabulation of the triangulation is performed instead of using the Delaunay triangulation. Figure 1.4 shows the different cases for 2D, while figure 1.5 shows the triangulation of a 3D element with four different discontinuities.

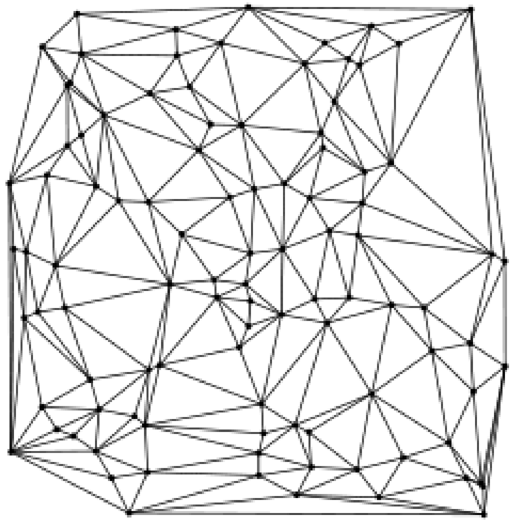

FIGURE 1.2: Delaunay Triangulation - A mesh formed of QUAD4 elements can be triangulated into TRI3 elements.

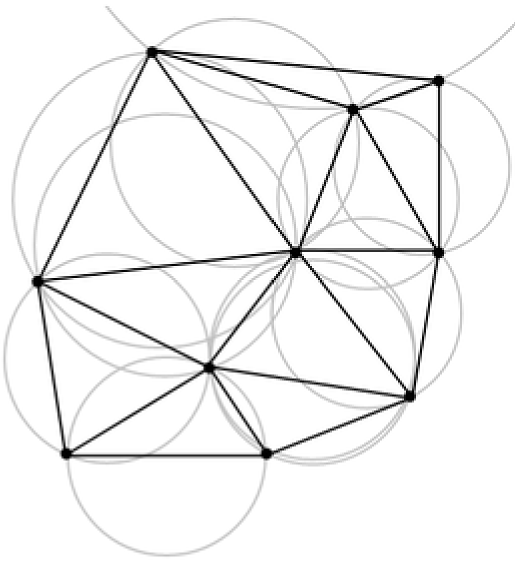

FIGURE 1.3: Delaunay Circumcircles - A set of points can be uniquely triangulated in a way that the points form circumcircles.

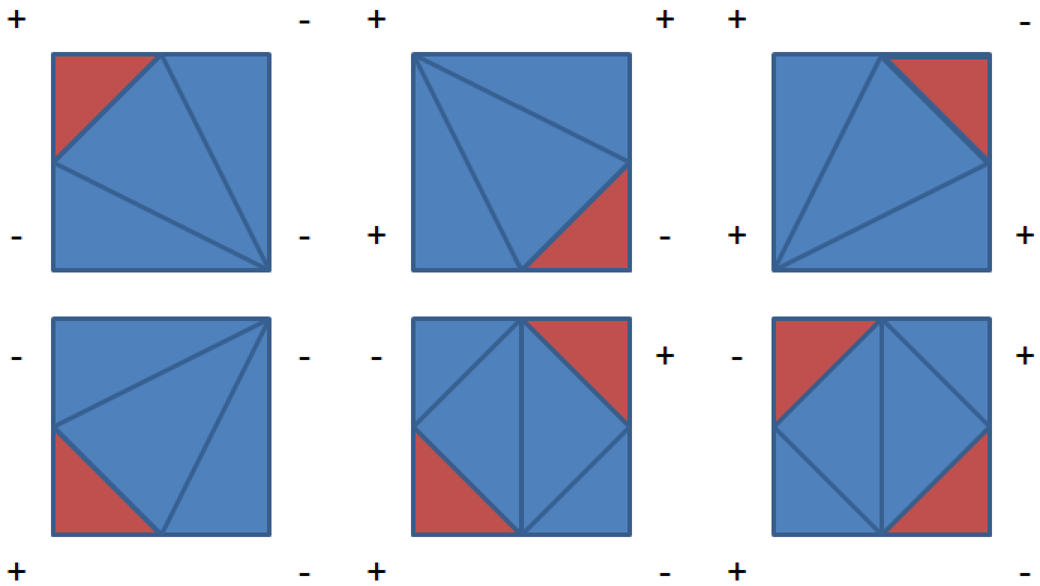

FIGURE 1.4: There are only 6 different triangulation configurations for an inclusion-based XFEM model.

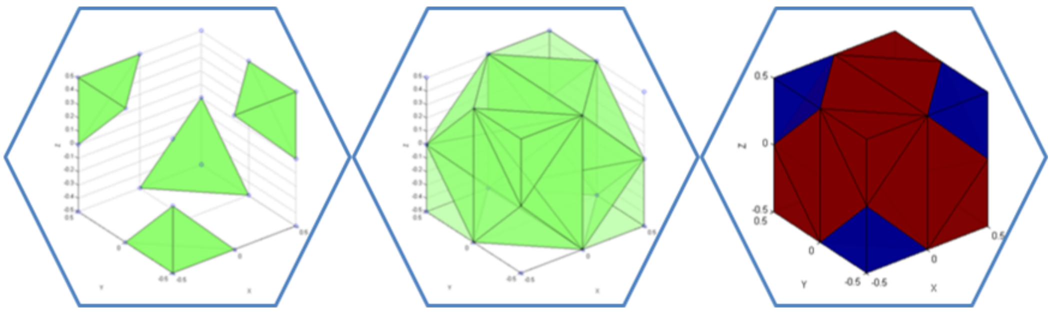

FIGURE 1.5: Triangulation in 3D is more complex and we use the Delaunay triangulation algorithm to perform the computation. This element with 4 discontinuities has 4 pseudo-elements from material phase 1 and 20 from material phase 2.

Chapter 2

XFEM implementation - Serial

2.1 Glossary

computational mesh – standard FE mesh that defines the geometry of our model.

model – physical entity, contains information about the XFEM elements and the level-set functions.

main phase (phase) – phase indicating a particular material phase (or void phase).

sub-phase – the domain of a main phase can be decomposed into multiple sub-phases.

intersection point – intersection created by the zero level-set curve cutting through an edge.

point – geometrical entity with information about coordinates, connected cells and edges.

cell – geometrical entity, a collection of points, owns a list of edges too.

edge – geometrical entity with information about the two points on its ends and its connected cells.

Delaunay triangulation – triangulation of our elements using their corner nodes and intersection points.

corner node – nodes that are at the corners of the elements.

middle node – nodes that are not at the corners of the elements, only used for higher order elements. **pseudo-element** – cells created by the triangulation of the regular element.

nodal cluster – set of elements (and their nodes) connected to a node **consistency nodes** – nodes shared by multiple elements within nodal cluster.

XFEM model/element update – an update to our lists that occurs every time the level-set distribution changes. These instances can be a new time-step, a new optimization-step, a new Newton iteration during a Hamilton-Jacobi problem or during the sensitivity analysis.

2.2 Introduction

This chapter outlines the procedure for building an XFEM model for a given distribution of the level-set function. The focus will be the sequential execution of the algorithm. This will be useful to explan the basic concepts of how the XFEM implementation works. The parallel implementation will be discussed in the following chapter and build upon the algorithms discussed here, so we recommend you read this chapter first.

2.3 Summary

The XFEM model consists of:

- Intersection points along elemental edges.
- XFEM elements sub-divided into cells for integrating the weak form of the governing equations within the individual sub-domains belonging to a particular material phase.
- Enrichment tables that define the nodal enriched degrees of freedom used to interpolate the solution within a cell.
- Parallel implementation of building XFEM model (see Chapter 3).

2.4 Overall Procedure

The main steps are:

1. *Build point-to-cells connectivity list* – (list of cells connected to a point) by looping over all cells; needs to be built only once.

2. *Build edge table in mesh* – (list of the cell edges that stores connectivity to points and cells) by looping over all cells; needs to be built only once.

3. *Build table of nodes belonging to a nodal cluster* – (first-order neighbors of a node; defined as all nodes belonging to elements connected to a node) by looping over all elements for a node using point-to-cell table; needs to be built only once.

4. *Build table of elements connected to a node* – by either looping over all elements or mapping the point-to-cells list to a node-to-elements list; needs to be built only once.

5. *Build edge intersection points* – by looping over all edges. Points are stored in mesh; however, the coordinates of the intersections are copied to the XFEM element. Needs to be built at each XFEM model or element update.

6. *Delaunay triangulation* – Each element is triangulated based on edge intersection. Needs to be built at each XFEM model or element update.

7. *Build table of phases and sub-phases* – for each triangle/tetrahedron (pseudo-cells) for each triangulated element. Needs to be built at each XFEM model or element update.

8. *Build enrichment table* – defines which enrichment levels are used to interpolate a triangulated field within an element.

9. *Activate degrees of freedom* – determine which degrees of freedom are used in the model based on the enrichment table.

This overall procedure is illustrated in Figure 2.1.

2.5 XFEM Implementation Algorithms

Consider the following XFEM model which consists of a 4-element mesh in 2D; the nodes on the left are clamped and right edge is subject to a constant pressure load. The level-set distribution leads to the intersection pattern shown below (figure 2.2). The mesh below shows the numbering of the points and cells (figure 2.3).

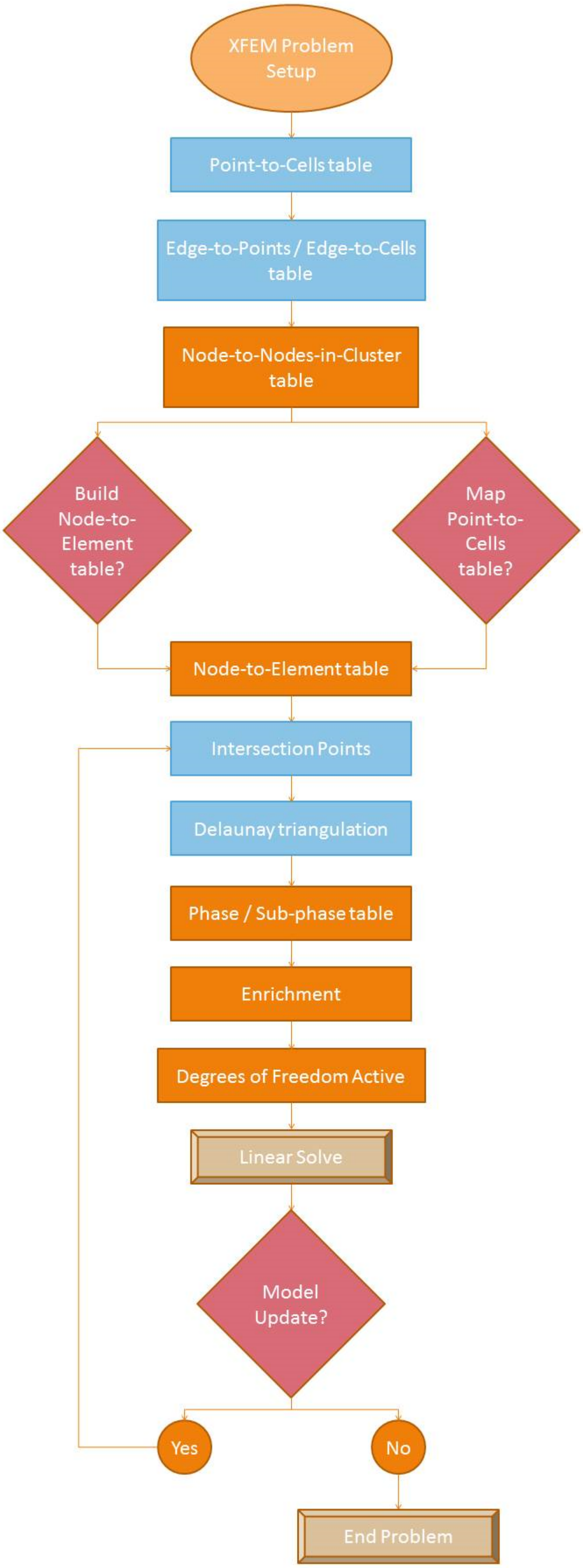

FIGURE 2.1: The diagram shows the overall procedure required to implement the XFEM model. Orange squares are steps that deal with the physics of the model, light blue indicates steps that correspond to the geometry of the mesh, diamond steps indicate decisions made by the program, and squares with double border indicate regular FEM steps

NOTE: In the computational mesh, and before the first XFEM model update, node IDs and element IDs match point IDs and cell IDs, respectively. After a model update, a new mesh is created and parsed to describe the new geometry of the problem. This resets most of the node-to-point and element-to-cell maps, and may cause this assumption to be false, especially in a parallel execution.

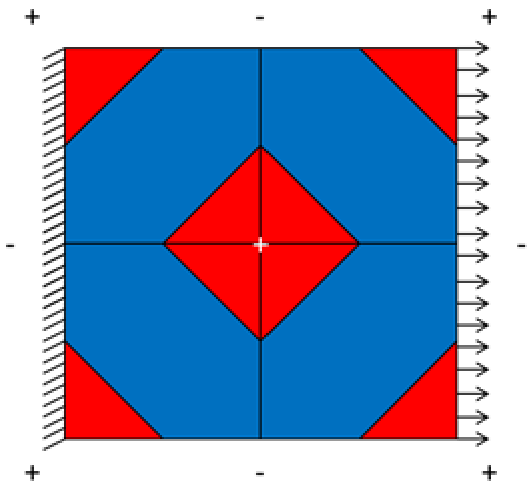

FIGURE 2.2: 4-element 2D mesh. Blue areas: material phase 1 negative level-set value at the node; red areas: material phase 2, positive level-set value.

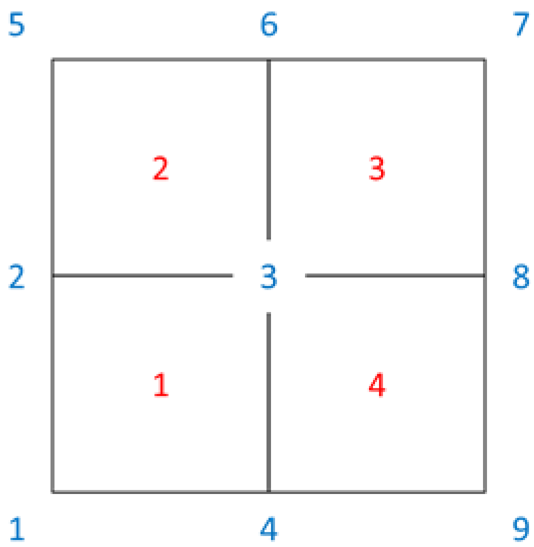

FIGURE 2.3: 4-element 2D mesh. Red numbers represent the global element ID, blue numbers represent the global node ID.

2.5.1 Point-to-Cells connectivity table

Class: MeshCore

Function: GeneratePointCellConnectivities

We need to build a point-to-cell connectivity table for each point in our computational mesh. We loop over all base cells in the computational mesh. Base cells (*BASE_CELL* enum) are all cells that are not sideset cells (*SIDE_CELL* enum). For each base cell we loop over all of its points and store the current cell index with point index. This leads to table 2.1:

Point Id	Number of cells connected	Cell Ids
1	1	1
2	2	1,2
3	4	1,2,3,4
4	2	1,4
5	1	2
6	2	2,3
7	1	3
8	2	3,4
9	1	4

TABLE 2.1: Point ID to cell IDs connectivity table

The connectivity matches the geometry of Figure 2.3.

2.5.2 Edge table

Class: Mesh

Function: CreateEdges

To generate a global edge table in our computational mesh we initially loop over all base cells. For each cell, we determine the number of edges and loop over all edges in the cell. For each edge, we check with the current cell whether the edge has been created. In the case the edge does not exist yet, we store the following edge information:

- IDs of points at the ends of edge.
- IDs of cells to which edge is connected.

We determine the cells which are connected to an edge via the intersection of cells connected to the end points of the edge, using the Point-to-Cells table.

At this instance, each global edge knows the point IDs on its ends and the cell IDs of the cells it is connected to. We can use this information to create a map that links the global edge to the local edge index per cell. Each cell has an internal edge order list pre-built that indicates the order in which its edges are organized. For example, a QUAD4 cell will have the following internal edge order list: 0 1, 1 2, 2 3, 3 0. The cell also knows the point IDs that it owns (if not directly, point IDs can be obtained through the nodes). Using these two lists, we can compute which point IDs lay in each of the elements internal edges. By matching the point IDs of the global edge to the point IDs of the internal cell edges, we can make a map that tells us which internal edge in a cell corresponds to a global edge. This list is stored with the edge in the same manner that cell IDs are stored.

REMEMBER: Cell IDs and internal edge numbers should have a one to one correspondence, meaning that for all connected cell to the global edge, we should know which local edge index it corresponds to.

For example, for the 4-cell cluster in Figure **??** we would obtain 12 global edges (see table 2.2).

Global Edge ID	Point IDs	Cell IDs	Local Edge Num
1	1, 4	1	0
2	3, 4	1 4	1, 3
3	2, 3	1 2	2, 0
4	1, 2	1	3
5	4, 9	4	0
6	8, 9	4	1
7	3, 8	3 4	0, 2
8	2, 5	2	3
9	5, 6	2	2
10	3, 6	2 3	1, 3
11	6, 7	3	2
12	7, 8	3	1

Table 2.2: Edge to point IDs and cell IDs connectivity table

The table shows, for example, that the global edge 2 is defined by the point IDs 3 and 4 and is connected to cells with IDs 1 and 4. For cell 1, it is the local edge 1 (counting from zero CCW starting at the bottom) and for cell 4 it is local edge 3 (figure 2.4).

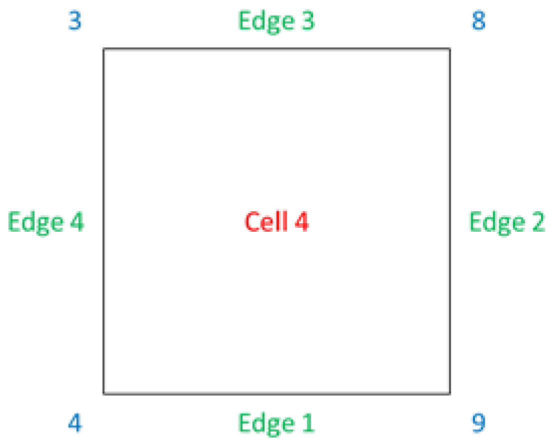

FIGURE 2.4: Edge representation in a QUAD4 element. Green edges represent the local edge index at the element.

2.5.3 Element clusters

Class: Model

Function: BuildNodeToElementTable

During the enrichment function, we will need to know which elements are connected to the node we are analyzing. A node-to-element table can be built in two ways:

- Create the table in the same manner the Point-to-Cells table was created: by looping over all elements, obtaining their list of nodes and then storing the element index (or ID) to the list of each node. A sort, unique, resize algorithm is needed at the end of the function.
- Use the already made Point-to-Cells connectivity table and use the Mesh maps to convert point IDs and cell IDs to node IDs and element IDs.

NOTE: This computes the connectivity of a node to elements in the current processor. It will not compute connectivity to off-processor elements. This issue will be later discussed in Chapter 3.

2.5.4 Nodal clusters

Class: Model

Function: BuildNodeToCornerNodesInClusterTable

To determine the enrichment level used to interpolate fields within the XFEM elements we need to determine the first-order nodal neighbors of a node. This is the set of nodes that belong to the elements connected to a node. In addition, we need to identify the nodes that belong to two or more elements; we refer to these nodes as consistency nodes. Consistency nodes are simply the nodes in a nodal cluster (nodes of the connected elements of a main node) that are shared by more than one element in the nodal cluster. This information can be obtained by looping over the connected elements of a node, and obtaining the node list for each element.

Then a sort, unique, and resize algorithm is implemented to have a unique list of nodes.

NOTE: This computes the connectivity of a node to nodes in the current processor. It will not compute connectivity to off-processor nodes. This issue will be later discussed in Chapter 3.

2.5.5 Intersection points

Class: Model

Function: UpdateElementIntersections → CreateIntersectionPoints

The intersection points are defined by the zero level-set values. We loop over all edges defined in the edge table created in Section 2.5.2 and compute the intersection points along the edge using the nodal level-set information of the edge endpoints. To do this, we map the points to nodes using the Mesh maps and then obtain the level-set nodal property from the node. Using the level-set values and the coordinates of the end points, we can do a linear interpolation to find the location of the zero level-set value. The coordinates of the intersection points are then sent to all elements connected to the edge by again mapping cells to elements using the Mesh maps. The information is stored in the corresponding XFEM element and will be used later during the Delaunay triangulation (Section 2.5.6). The process is illustrated in Figure 2.5.

NOTE: This function walks a fine line between working in the geometry of the mesh and the physics of the model. Because in the end we need to know level-set values at the nodes, the function was placed in the Model class rather than the Mesh class.

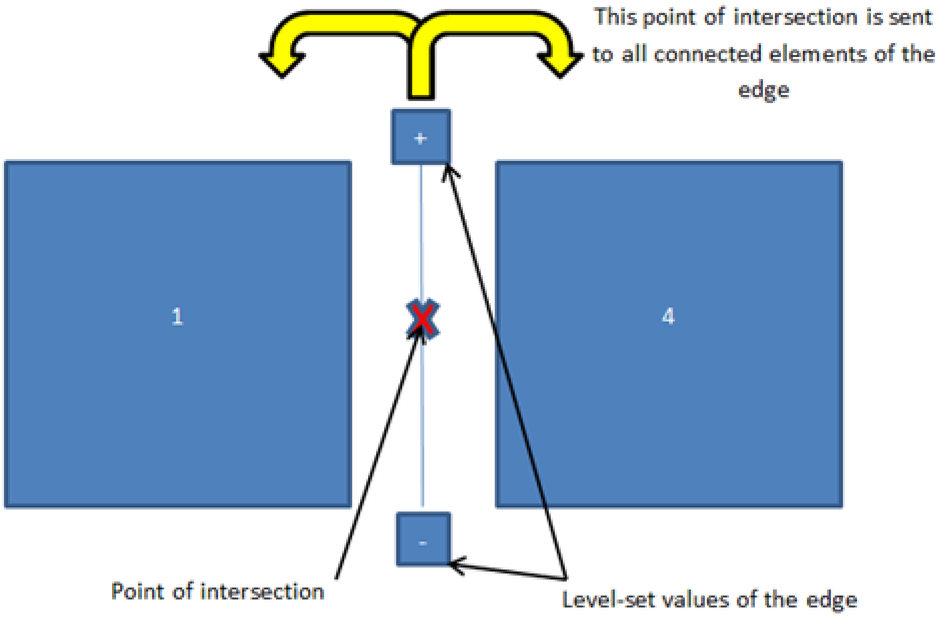

FIGURE 2.5: Mapping of intersection point to element.

2.5.6 Delaunay triangulation and assignment of main and sub-phases to pseudo-elements

Class: Model

Function: UpdateElementIntersections → UpdateXFemElement

We loop over all elements in the model and, if intersected, we perform a Delaunay triangulation, using the corner nodes and the edge intersection points stored with the elements in Section 2.5.5. The Delaunay triangulation requires only the coordinates of the element corner and edge intersection points. The Delaunay triangulation will return a list of triangles for 2D or tetrahedrons for 3D problems.

We use the Fortran library GEOMPACK3 to perform the triangulation.

NOTE: We do not take into account middle nodes for performing the triangulation. Their information is not needed for the enrichment algorithm and they will simply inherit the enrichment level from their neighbors corner nodes.

2.5.7 Main phase and sub-phase

We assign a main and sub-phase to each triangle/tetrahedron. The main phase is determined based on the average main-phase value of the pseudo-element. In case the average is zero, we apply an exception rule (TBD). The sub-phase information is based on the connectivity of pseudo-elements which belong to the same main phase and is computed via a flood-fill algorithm. To this end we collect the pseudo-elements into a pseudo-mesh; each intersected XFEM element has its own pseudo-mesh which consists of the points and the connectivity of the pseudo-elements. The main steps (figure 2.6) of the flood-fill algorithm used are:

- Build edge (or face) table for pseudo-elements of current XFEM element, analogue to Section 2.5.2.
- Loop over all elements that have not been assigned a sub-phase:
  - Find unprocessed element; recursively find neighbors with same main-phase; assign lowest unassigned sub-phase to elements found in search process.

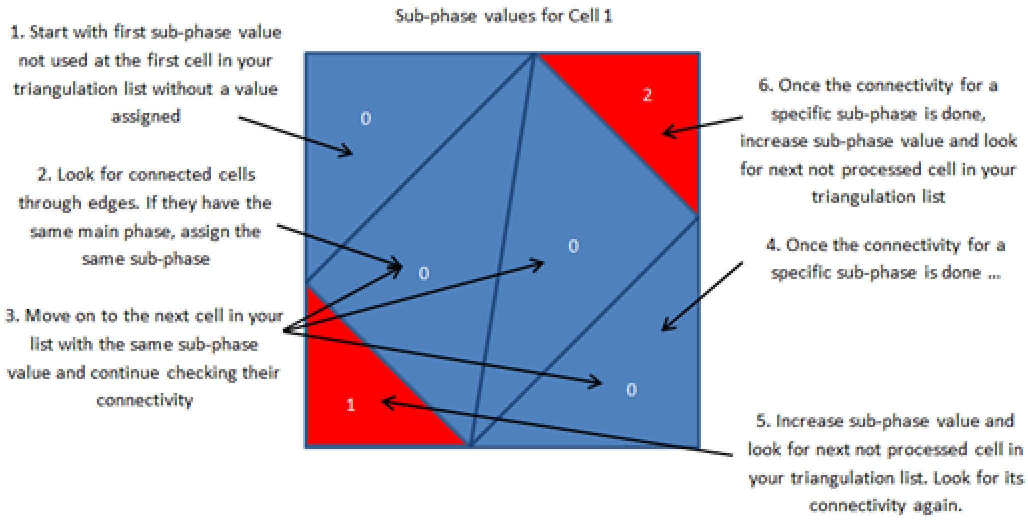

FIGURE 2.6: Sub-phase computation algorithm. Refer to Section 2.5.7 for a more detailed description.

For figure 2.6, we would get the list in table 2.3:

We have 6 different triangles generated by Delaunay triangulation. By convention, the order of the triangles is determined after the triangulation so that all phase 1 ones are

Triangle number	Main Phase	Sub-Phase
1	1	0
2	1	0
3	1	0
4	1	0
5	2	1
6	2	2

Table 2.3: Main-phase and sub-phase table for pseudo-elements.

located at the top, followed by the phase 2 triangles. Each triangle is assigned a main phase and sub-phase (see table 2.3); the assignment is stored, using a one-to-one map by the XFEM element.

2.5.8 Nodal enrichments for pseudo-elements

Class: Model

Function: DetermineEnrichments

To interpolate fields in the XFEM elements we need to determine which nodal enrichments are used within each pseudo-element. The enrichments need to be chosen such that the interpolations are continuous across adjacent elements within the same main phase and unique within pseudo-elements of the main sub-phase but topologically disconnected.

The main concept of the procedure is to clearly separate element-level and node-level operations. This separation enables the parallelization of the procedure, the process of which will be discussed in Chapter 3. The first step is to determine the enrichment levels a particular node will use to interpolate fields in the triangulated elements with a particular sub-phase. This step is done by looping over all nodes. The result of this loop is a map that links the sub-phase information of an element to an enrichment level for each node of the element. In a second step we loop over all elements to update the enrichment levels of pseudo-element based on the map built previously.

Looping over all nodal clusters, we build the following node-element table. The entries in the table are the sub-phases at the nodes within each element. The consistency node numbers are marked by a “C”. Consistency nodes can be identified by nodes which have entries in more than one column; so they can be identified easily on the fly.

Two conditions, consistency and uniqueness, must be satisfied to ensure the assigned sub-phases are consistent across the nodal cluster. The consistency condition is satisfied if all sub-phases in a row are the same. The uniqueness condition is satisfied if the sub-phase of each set of connected nodes is unique in the cluster.

Nodes/Elements	1	2	3	4
1	14			
2 "C"	0	0		
3 "C"	15	14	14	15
4 "C"	0			0
5		15		
6 "C"		0	0	
7			15	
8 "C"			0	0
9				14

TABLE 2.4: Initial node-element table.

The initial node-element table (see table 2.4) shows that the consistency condition is not satisfied since node 3 is inconsistent. Also, nodes 5, 7, and 9, for example, should be assigned a unique sub-phase. Therefore the uniqueness condition is also not satisfied. To ensure both conditions we build a second table where we iteratively correct the sub-phase until all conditions are satisfied. Note that the consistency and uniqueness checks are not needed for a 1 element cluster. The correction procedure is as follows:

1. Initialize a list of all sub-phase possible; mark them as unused; initialize a list of checked nodes. Mark all nodes as unchecked; build list of consistency nodes.

2. Ensure consistency: Repeat the following steps until all consistency nodes are checked.

   - Select node: Start with the center node, which must be a consistency node (remember that this consistency and uniqueness check will not be applied for a one-element cluster). Otherwise select the first unchecked consistency node.

   - Select sub-phase: Select the lowest unused sub-phase. Mark the selected sub-phase as used (keep consistency of sub-phase to the respective main phase).

   - Identify connected consistency nodes and connected unique nodes:

     – For each element to which the selected node belongs, connected nodes have the same sub-phase as the selected node has for this particular element. Select the connected nodes which may be either consistency or unique nodes.

  – In order to identify all connected consistency and unique nodes, search the connected elements for additional connected consistency and unique nodes.

  – The above process requires recursively (a) searching for nodes within an element with the same sub-phase and (b) identifying elements that share consistency nodes. Note: the sub-phase id might change between elements if the sub-phases are not consistent yet for a consistency node.

- Assign sub-phase: Assign the selected sub-phase to all connected nodes identified in the search process described above. For any value that needs to be changed, flip sub-phases for that element. Mark connected nodes as checked.

Nodes/Elements	1	2	3	4
1	14 → 15			
2 "C"	0	0		
3 "C"	15 → 14	14	14	15→ 14
4 "C"	0			0
5		15		
6 "C"		0	0	
7			15	
8 "C"			0	0
9				14→ 15

TABLE 2.5: Flipping the enrichment levels to keep consistency.

3. Ensure uniqueness: Loop through the remaining unchecked nodes. For each element, collect the unique nodes with an unused sub-phase, assign the next unused sub-phase to these nodes and check the sub-phase as being used. Here nodes 1, 5, 7, and 9 are assigned the sub-phase 15, 16, 17, and 18, respectively (node 1 was flipped initially, but it was not marked as checked). The outcome of the above correction procedure is table 2.6.

4. Using the original and the corrected node-element tables we can build the map for the central point (here node ID 3). See table 2.7. The rows correspond to the elements, the columns list the original sub-phases, and the entries are the enrichment levels used by the central node. When building this table we check that the map is consistent within itself (the same sub-phases within each are assigned to the same enrichments).

Nodes/Elements	1	2	3	4
1	15			
2 “C”	0	0		
3 “C”	14	14	14	14
4 “C”	0			0
5		16		
6 “C”		0	0	
7			17	
8 “C”			0	0
9				18

Table 2.6: The result of the enrichment algorithm. Nodes that are shared across elements receive the same enrichment level.

For Node 3: Element / Enrichment level	0	14	15
1	0	15	14
2	0	14	16
3	0	14	17
4	0	18	14

Table 2.7: Element to enrichment table for node ID 3. This table shows the initial enrichments node 3 received during the sub-phase algorithm and the enrichment levels after the enrichment algorithm.

5. With this map we can loop over all elements and assign enrichment levels for each node to each pseudo-element, based on their sub-phase information.

2.5.9 Enrichment of higher order element nodes

Class: Model & XFemElement

Function: DetermineEnrichments → AssignEnrichmentLevelToMiddleNodes

As mentioned in previous sections, the entire enrichment algorithm works on corner nodes and middle nodes for higher order elements simply inherit the enrichment level from their neighbors.

NOTE: We will assume for this algorithm that middle nodes are always located at the edges of the element, so higher order elements with nodes in the center or nodes in its faces will require a different implementation.

We loop over all middle nodes in an element, locate on which edge it is on, and then find the other two corner nodes in that edge. We compare the level-set value of the middle node we are analyzing to the level-set values of the two neighbor corner nodes and pick the corner node that has the same main phase. If both neighbor corner nodes have the same main phase, we pick one randomly. We copy the column from the enrichment table that corresponds to the corner node we picked and paste it in the column that corresponds to the middle node.

We repeat this procedure for all middle nodes across all elements. This procedure is illustrated in Figure 2.7.

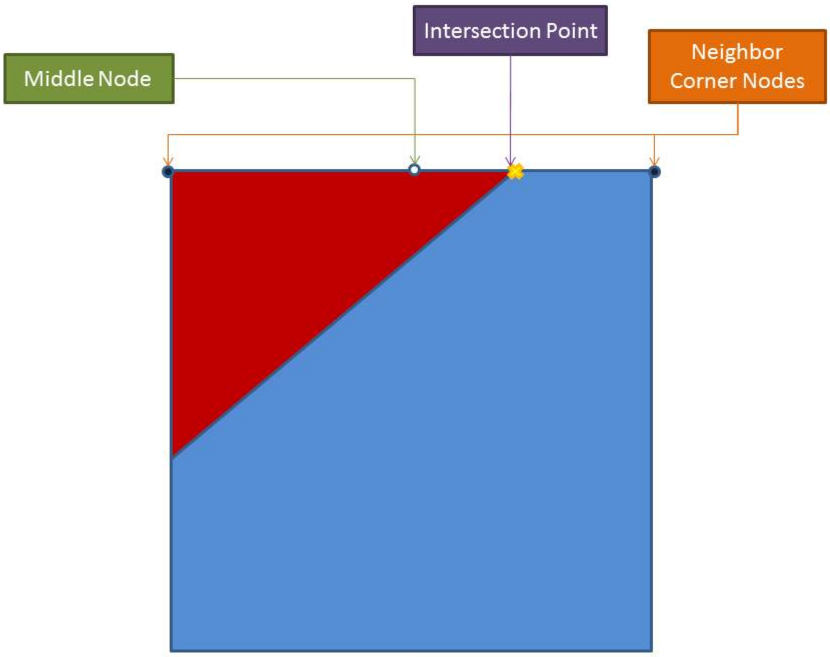

FIGURE 2.7: A middle node will have two neighbor corner nodes in an edge. The middle node copies the enrichment information from the neighbor node that shares the same main phase. In this example, the middle node copies the information from the left corner node.

2.5.10 Determine degrees of freedom used

Class: Model

Function: DetermineEnrichments → AssignEnrichmentLevelToNode → AssignEnrichmentLevelToDofs

The last step in the algorithm is to flag which enrichment levels each node (corner or middle) will use for interpolation. In a regular FEM problem, each node in the model contains a vector that flags which DOFs are active in the problem (we call this the DOF mask), and another vector that flags the boundary conditions active in the problem (we call this the DBC mask). For example, a node in a diffusion problem will have a bitset (or boolean) DOF mask vector that has the entry that corresponds to the temperature DOF (TEMP) flagged as true. In an XFEM problem, and before the enrichment function, the same node will have the TEMP DOF marked as true and also all DOFs corresponding to the enrichment of TEMP. Assuming we are using 27 enrichment levels, they will be called XFEM0000, XFEM0001, ..., XFEM0026. Of course, we will not use all of these DOFs, but we initially allocate them because we do not know the intersection distribution our level-set function will create. After the enrichment function, however, we know which DOFs we will actually use. We do not want to flag unused DOFs as false though in our regular DOF mask because the DOFs used or unused may change with every model update. Therefore, we create a third additional vector to flag the DOFs used during this model update. We will call it a temporary DOF mask vector because it will be reset and set everytime the level-set field in our model changes.

Node 3 in our test case has enrichment levels 0, 14, 15, 16, 17, 18 active. Each one of these levels can be converted into a DOF type by checking how our DOFs were arranged initially:

- 0 $\rightarrow$ TEMP
- 1 $\rightarrow$ XFEM0000
- 2 $\rightarrow$ XFEM0001

  ⋮

- 27 $\rightarrow$ XFEM00026

Then we can use these DOF types to flag in our temporary DOF mask that they are being used. Finally, when computing our list of free DOF IDs, we will intersect our regular DOF mask with our temporary DOF mask and determine which DOFs are active.

REMEMBER: the temporary DOF mask will be reset at the next XFEM model update.

2.5.11 Implementation Example 2

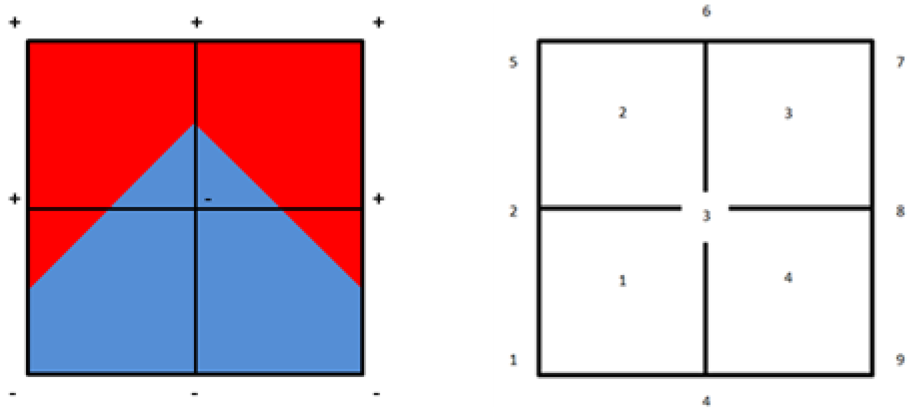

FIGURE 2.8: XFEM implementation configuration 2

Nodes/Elements	1	2	3	4
1	0			
2 "C"	14	14		
3 "C"	0	0	0	0
4 "C"	0			0
5		14		
6 "C"		14	14	
7			14	
8 "C"			14	14
9				0

TABLE 2.8: Initial node-element table for configuration 2.

Nodes/Elements	1	2	3	4
1	0			
2 "C"	14	14		
3 "C"	0	0	0	0
4 "C"	0			0
5		14		
6 "C"		14	14	
7			14	
8 "C"			14	14
9				0

Table 2.9: Final node-element table for configuration 2.

For Node 3: Element / Enrichment level	0	14
1	0	14
2	0	14
3	0	14
4	0	14

Table 2.10: Enrichment level map for configuration 2.

2.5.12 Implementation Example 3

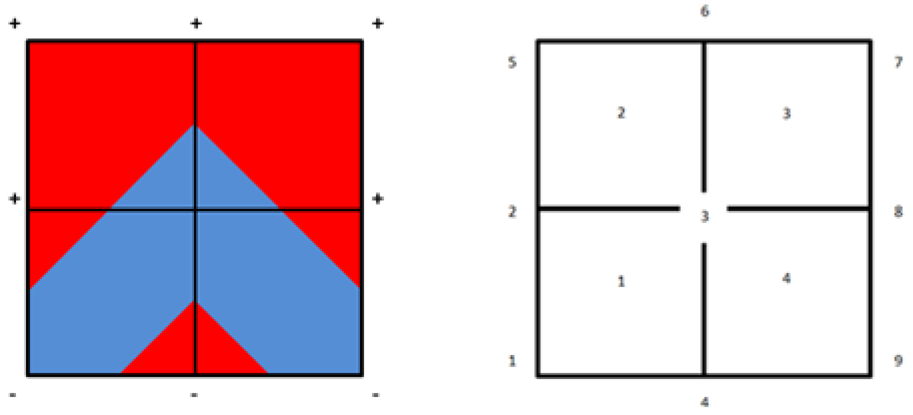

Figure 2.9: XFEM implementation configuration 3

Nodes/Elements	1	2	3	4
1	0			
2 “C”	15	14		
3 “C”	0	0	0	0
4 “C”	14			0
5		14		
6 “C”		14	14	
7			14	
8 “C”			14	15
9				0

Table 2.11: Initial node-element table for configuration 3.

Nodes/Elements	1	2	3	4
1	0			
2 “C”	14	14		
3 “C”	0	0	0	0
4 “C”	15			15
5		14		
6 “C”		14	14	
7			14	
8 “C”			14	14
9				0

Table 2.12: Final node-element table for configuration 3.

For Node 3: Element / Enrichment level	0	14	15
1	0	15	14
2	0	14	
3	0	14	
4	0	15	14

Table 2.13: Enrichment level map for configuration 3.

2.6 Solving the problem

In the previous section, our algorithms determined which additional enriched degrees of freedom each node requires to account for the discontinuities in the elements. Once this is achieved, we can proceed to solve our problem like a regular FEM problem because in the end, the enriched degrees of freedom are just that, degrees of freedom like displacement or stress.

The only difference comes in which functions are used to interpolate. The original degrees of freedom are interpolated using the regular shape functions for the problem. The enriched degrees of freedom will interpolate using shape functions based on the partition of unity and the Heaviside function.

A standard *extended* finite element approximation of a function $u(x)$ is:

$$u^h(x) = \sum_{i \in I} N_i(x) u_i + \sum_{i \in I^*} M_i(x) a_i \tag{2.1}$$

The first sum is the standard FEM approximation while the second sum is the enrichment applied to the interpolation.

In this function, the domain $\Omega \in R^n$ is $n - \textit{dimensional}$ and it is meshed into n^{el} elements. The other terms mean:

$u^h(x)$ – approximated function.

$N_i(x)$ – standard FE shape function at node i.

u_i – solution for the displacement at node i.

I – set of all nodes in the domain.

$M_i(x)$ – local enrichment function at node i.

a_i – displacement of the enriched displacement at node i.

I^* – set of all enriched nodes $\in I$.

The enrichment functions are built by means of the partition of unity principle.

$$M_i(x) = N_i^*(x) * \Psi(x) \tag{2.2}$$

where $N_i^*(x)$ are the partition of unity functions such that

$$\sum_{i \in I^*} N_i^*(x) = 1 \tag{2.3}$$

The implementation in this document models weak discontinuities and therefore, the global enrichment function is chosen based on the Heaviside function as follows:

$$\psi(x) = sign(\phi(x)) = \begin{cases} -1 & \text{if } \phi(x) < 0 \\ 0 & \text{if } \phi(x) = 0 \\ 1 & \text{if } \phi(x) > 0 \end{cases} \tag{2.4}$$

2.7 Preconditioner

When a sub-domain of a material phase is too small (around $O(\epsilon^{1/2})$), the Jacobian matrix will be ill-conditioned. To solve this shortcoming, it is necessary to scale the matrix with another pre-conditioning matrix. This scaling matrix will be a function of the level-set field.

The pre-conditioner will have a scaling value for each degree of freedom in the problem. If the degree of freedom is constrained, the scaling value will be 1. To obtain these values, we check which enriched degrees of freedom each node uses. Then, proceed to compute the integral of the shape function for the node with respect to the material sub-domain that requires said enriched degree of freedom. Because a node will have different scaling values across multiple elements, there are four pre-conditioner implementations available to compute the scaling value in a nodal cluster:

- Maximum value of integrals of shape functions
- Sum of values of integrals of shape functions
- Maximum value of integrals of the derivatives of the shape functions
- Sum of values of integrals of the derivatives of the shape functions

After this computation, each node will have a scaling value for each enriched degree of freedom it is using. This scaled value is computed as the inverse of the square root of the integral calculated.

At this point, there is one scaling value for each degree of freedom in the system. These scaling values are applied to the solution vector before the computation of the residual and Jacobian. After the residual and Jacobian are computed, they are both unscaled and then the new solution vector is computed.

Chapter 3

XFEM implementation - Parallel

3.1 Glossary

communication node – a node that is shared by more than one processor.

3.2 Introduction

The algorithm developed for assigning enrichment levels to the inclusions in our model can also be extended to work in parallel. So far, for serial, the algorithm was divided into 3 main sections: computing sub-phase information at the element level, assigning enrichment-levels at the nodal level, and finally flagging the used degrees of freedom (regular and enriched) at the element level. The bottom part of the diagram in figure 2.1 illustrates this procedure.

The enrichment function is a nodal procedure and therefore, the node needs to know sub-phase information from all connected elements across all processors in order to build the initial consistency matrix. However, as mentioned in Section 2.5.3, the node only has information from connected elements in the current processor. The basis of the algorithm in parallel is that communication nodes will receive sub-phase information from off-processor elements and then each solve for consistency and uniqueness of enrichment levels. Basically, each communication node will solve our consistency matrix individually. This means the same table will be solved as many times as processors are being used, but in exchange it will avoid performing an extra MPI communication. This also ensures that there are no parallel conditional statements in the enrichment function and that only one communication between processors is performed at the beginning of the enrichment function.

After the enrichment function, each node will flag what DOFs it is using in its temporary DOF mask. As in serial, this operation is elemental: each element tells its nodes to flag their vectors with the DOFs it is using based on the activation table. This was explained in subsection 2.5.10. The change in the parallel implementation is that once nodes have flagged their used DOFs, lower-ranked processors must send this information to higher ranked processors.

Overall, with these changes, the parallel implementation will look like the diagram in Figure 3.1. The rest of the process is the same as the diagram in Figure 2.1.

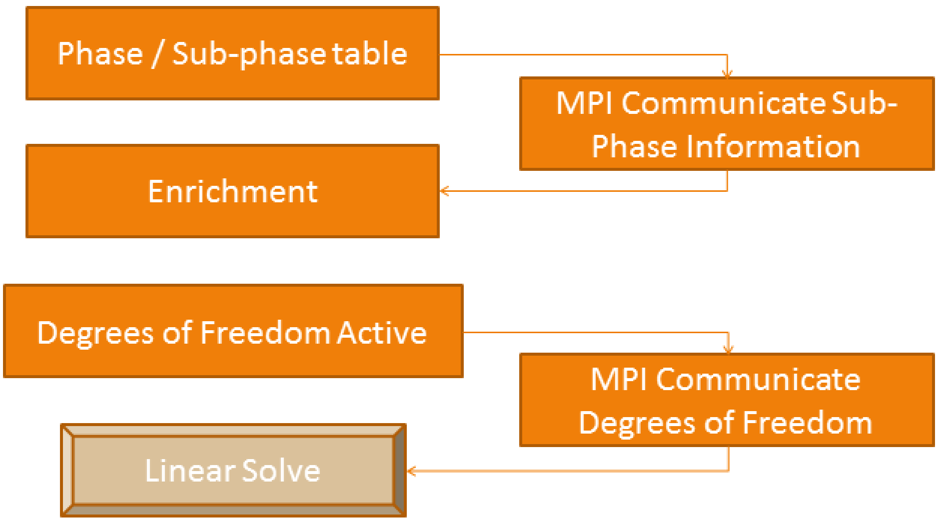

FIGURE 3.1: The diagram shows the overall procedure required to implement the XFEM model in Parallel.

3.2.1 Parallel Communicate Sub-Phase Information

Class: Model

Function: ParallelCommunicateEnrichmentInfo

The first parallel communication will be a complete exchange of information across all processors. Each communication node will send the sub-phase information to all processors that also share the node. In our implementation, we used MPI routines to send and receive information. The packaging of information goes as follows:

- Communication Node ID
- Number of on-processor connected elements
  - ID of each connected element
  - Number of nodes in each connected element
    - Node ID of each node in current element
    - Sub-phase value of each node in current element

Basically, there will be two loop levels in our package: one for the number of on-processor connected elements and one for number of nodes in each element.

Once the communication node has this information packaged, it will send it to all other processors with which it shares a communication map. At the same time, it will receive information from those other processors. MPI functions MPI_ISend and MPI_IRecv are recommended to achieve this.

Once the node receives off-processor information, it can store it and use later during the enrichment function. Using the information it received, it can compute the initial consistency matrix and perform the algorithm as if it were a serial execution.

3.2.2 Parallel Communicate Enriched Global DOFs

Class: Model
Function: ParallelCommunicateSharedGlobalDofs

The second part concerns which DOFs each communication node is using. The packaging will be slighty different:

- Communication Node ID
- Number of DOFs used
  - DOF location in temporary DOF mask

This time we will only need one loop, for the number of DOFs used by the node.

Also, we do not need to make a complete exchange. Only lower ranked processors need to send information to higher ranked processors because free DOF IDs are only counted for Master nodes. Once higher-ranked processors receive this information package, the communication nodes can flag which DOFs are being used across all processors.

This is all that is needed to implement the enrichment function in parallel, two extra communication steps. The rest of the algorithm works the same as it did in serial.

Chapter 4

XFEM corroboration and results

4.1 Methodology

Two formulations were used to corroborate the results of the XFEM implementation.

Equation 4.1 computes the difference in solutions at the discontinuity. Since the model we have implemented is based on inclusions and not crack propagation, this interface error should approach zero as the mesh gets finer.

$$\sqrt{\frac{\sum_{element}\sum_{interface}\int u^{+} - u^{-}\mathrm{d}\Gamma_i}{\sum_{element}\sum_{interface}\int \mathrm{d}\Gamma_i}} \tag{4.1}$$

This equation computes the interface "jump" across all interfaces and elements in the model, then scales it with respect to the perimeter or area of the interface, and finally takes the square root.

Equation 4.2 compares the relative difference between the XFEM solution and the FEM solution.

$$\sqrt{\frac{\int u_{XFEM} - u_{FEM}\mathrm{d}\Omega}{\int u_{FEM}\mathrm{d}\Omega}} \tag{4.2}$$

u_{XFEM} represents the XFEM solution, while u_{FEM} represents the FEM solution.

XFEM was used to solve a thermal problem with the configuration of figure 4.1. The same problem was ran using the classical FEM. The FEM problem used two different types of elements and its mesh was refined until the solution reached convergence.

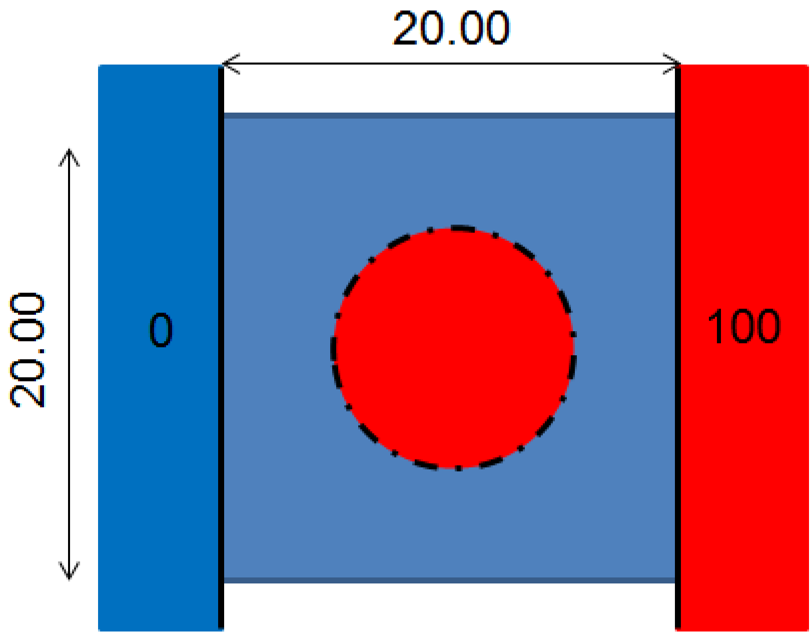

FIGURE 4.1: Thermal problem setup.

The mesh has a width of 20 units and a height of 20 units. The problem has Dirichlet boundary conditions on the sides. The temperature is prescribed to 0 on the left side and 100 on the right side. There is an inclusion at the center of the model. This inclusion is a different material with a different thermal conductivity than the material phase 1 domain.

The test consisted in modifying the diameter of the circle from 2 units to 6 units in 500 steps using different mesh refinements, different conductivity ratios and different pre-conditioners formulations.

4.2 Tests

4.2.1 Mesh refinement sweep

The mesh size was the variable in this test, while the conductivity ratio between both materials remained fixed at 10. No pre-conditioner scaling was applied. The different mesh sizes used were:

- 20x20
- 30x30
- 40x40
- 50x50

Figure 4.2 shows that as the mesh is refined, the interface error converges to zero.

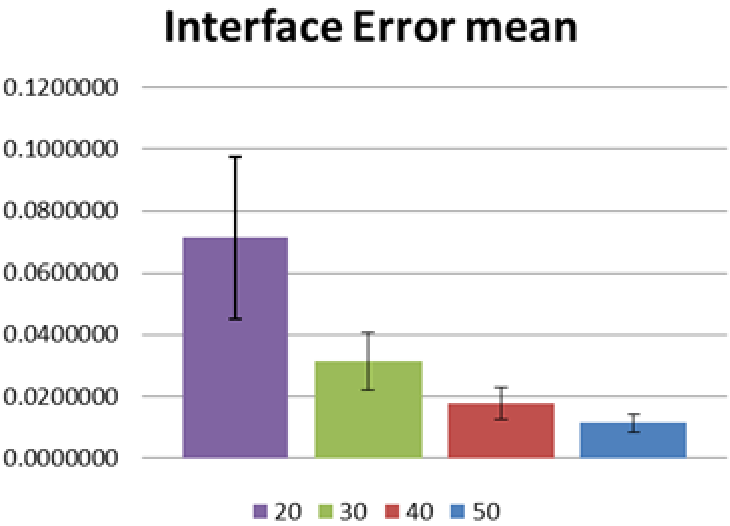

FIGURE 4.2: Mesh refinement sweep interface error.

Figure 4.3 shows that as the mesh is refined, the difference of the XFEM solution with respect to the FEM solution decreases. The larger difference for the 50x50 mesh is due to the sampling and different mesh sizes used for the XFEM and FEM problems. A different mesh resampling size fixed the issue in other tests.

4.2.2 Conductivity ratio sweep

The conductivity ratio between the different materials was the variable in this test. The mesh size was 30x30 and the pre-conditioner formulation used the maximum spatial derivative of the shape functions. The different conductivity ratios used were:

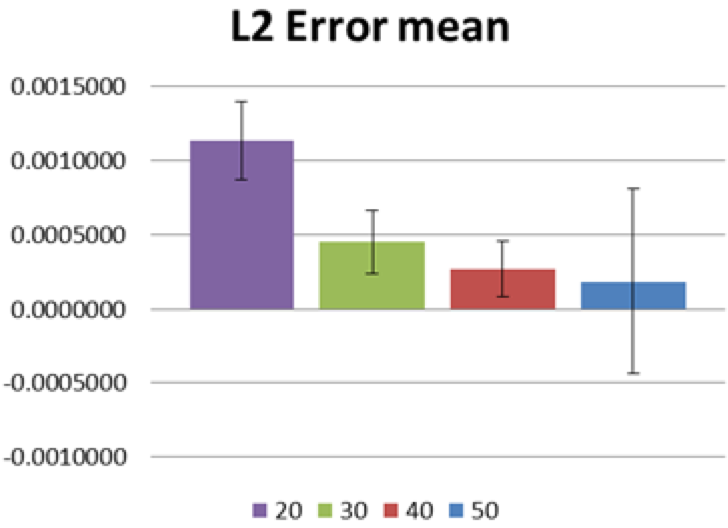

FIGURE 4.3: Mesh refinement sweep L2 error.

- 0.1
- 1
- 10
- 100
- 1000

Figure 4.4 shows that when the material conductivity is the same for both materials (a "quasi-FEM" problem), the interface error is in the order of $O(\epsilon)$. However, the greater the difference in material properties at an interface, the larger the interface jump is.

For the L2 computation, only FEM solutions with conductivity ratios of 10, 100, and 1000 were computed. Figure 4.5 shows that the difference in solutions is very small $O(10^{-4})$.

4.2.3 Condition number comparison

These tests were performed to compare the condition number of the Jacobian matrix when the pre-conditioning was applied. The mesh size was 30x30, the conductivity ratio was 10 and the pre-conditioner formulation used the maximum spatial derivative of the shape functions.

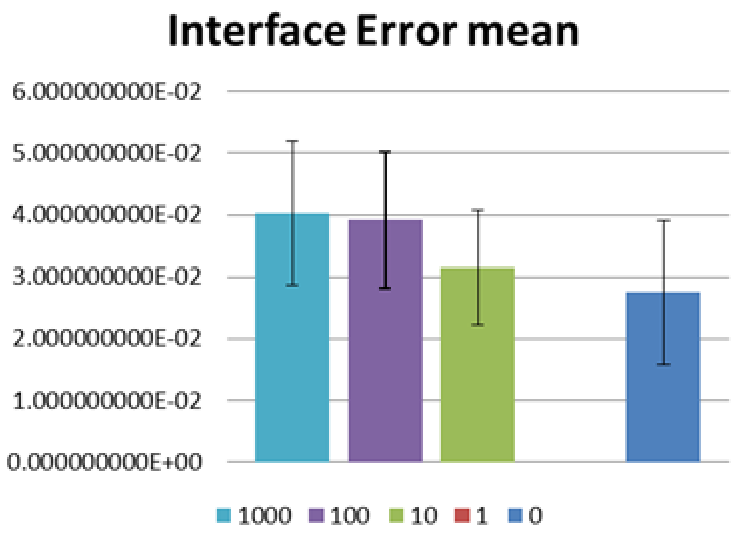

FIGURE 4.4: Conductivity refinement sweep interface error.

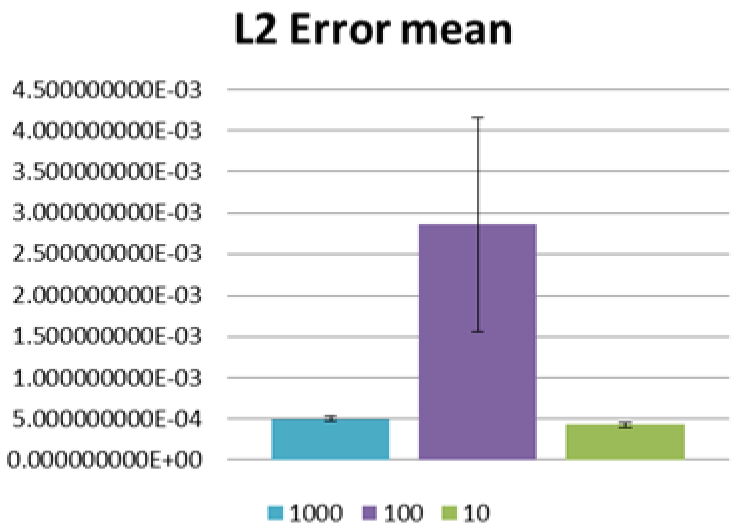

FIGURE 4.5: Conductivity refinement sweep L2 error.

Two different solution algorithms were used and compared, a direct solver and a GMRES iterative solver.

Figure 4.6 shows that the Jacobian matrix has a condition number in the order of 10^{15} when no pre-conditioning is applied, while figure 4.7 shows that condition number is decreased to the order of 10^4 when scaling is applied.

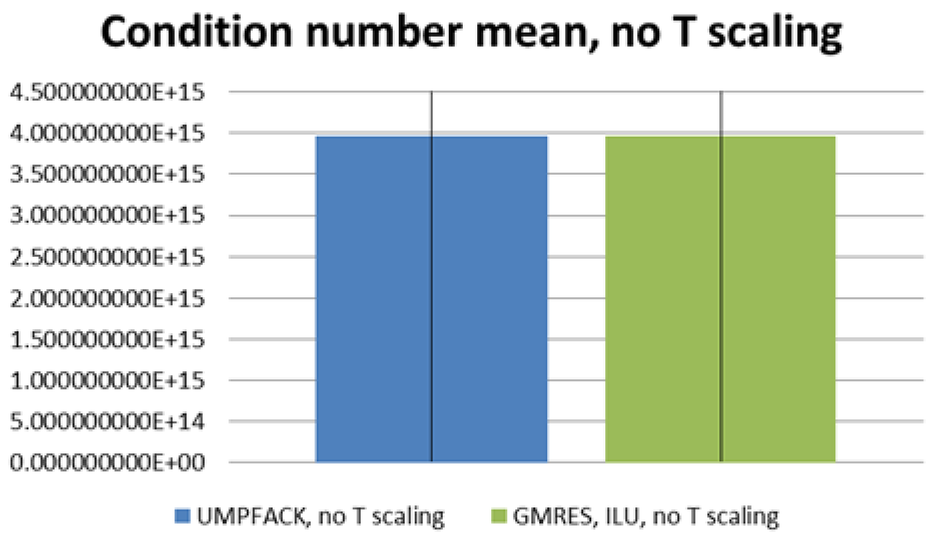

Figure 4.6: Condition number comparison - no pre-conditioner.

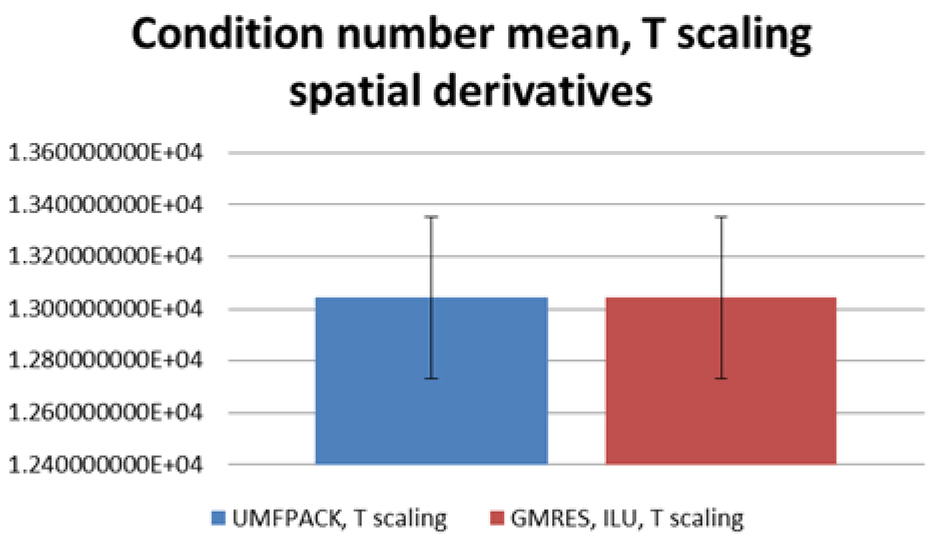

Figure 4.7: Condition number comparison - with pre-conditioner.

Chapter 5

Conclusions

5.1 Conclusions

The framework developed in this project will help eliminate the need to re-mesh a model when discontinuities present in the domain. The program is capable of dividing an element into integrable sub-domains, calculating its topology, its enrichment information and computing the normal vector and Gauss points required for integration. The program is also capable of solving XFEM problems with different topologies in 3D.

Results showed that the differences in solutions with a classical FEM problem are small.

XFEM produced Jacobian matrices with high condition numbers when intersection where close to element boundaries, but the application of a pre-conditioner as a function of the level-set field solved this shortcoming.

Appendix A

Delaunay Triangulation code

This code written in Matlab is the first attempt of the author to perform a Delaunay triangulation in an element based on the level-set function values at the corner nodes. To triangulate different discontinuity configurations change the *levs* variable: each entry corresponds to the value of the level-set function at a node. The e_x, e_y and e_z vector variables contain the coordinates of the corner nodes and can be modified to change the shape of the element.

A.1 main.m

```
function[] = main()
% Main program
% Modify ex, ey, and ez to change shape of element
% Change levs to change level-set configuration

% Global x,y,z coordinates of element.
% This will form a 3D cubic element cube.
ex = [-0.5 -0.5 -0.5 -0.5 0.5 0.5 0.5 0.5];
ey = [-0.5 -0.5 0.5 0.5 -0.5 -0.5 0.5 0.5];
ez = [0.5 -0.5 -0.5 0.5 0.5 -0.5 -0.5 0.5];

%% Extract a particular case (set k manually).

% Initial test for a particular levelset function.
% Randn function will return a m x n matrix of random positive and negative
% numbers.
hexsect = cell(127,8);
k = round(1 + (127-1).*rand);
```

```
% levs = randn(1,8);
levs = [-1 -1 -1 -1 -1 -1 -1 1];
% levs = [-10 1 -10 1 -10 1 -10 1];
[nsct,isct,xsct,ysct,zsct,levs] = xfem8isct(ex,ey,ez,levs);
[xtet,ytet,ztet,ctet,ptet,pnd,plist,Tetp1,Tetp2,tet1G,tet1L,tet2G,tet2L,
    size_phase1,size_phase2,volume1,volume2,total_volume1,total_volume2] =
    xfem8tet(isct,xsct,ysct,zsct,ex,ey,ez,levs);
hexsect{k,2} = xtet;
hexsect{k,3} = ytet;
hexsect{k,4} = ztet;
hexsect{k,5} = ctet;
hexsect{k,6} = ptet;
hexsect{k,7} = pnd;
hexsect{k,8} = plist;
[max(plist) min(plist)];

figure(1)
tetramesh(Tetp1,[xtet(1,:)',ytet(1,:)',ztet(1,:)'],-ones(size(Tetp1,1),1));
xlabel('X')
ylabel('Y')
zlabel('Z')
axis equal

figure(2)
tetramesh(Tetp2,[xtet(1,:)',ytet(1,:)',ztet(1,:)'],ones(size(Tetp2,1),1));
xlabel('X')
ylabel('Y')
zlabel('Z')
axis equal

figure(3)
tetramesh(ctet,[xtet(1,:)',ytet(1,:)',ztet(1,:)'],ptet);
xlabel('X')
ylabel('Y')
zlabel('Z')
axis equal

figure(4)
plot3(xtet(1,:),ytet(1,:),ztet(1,:),'X')
xlabel('X')
ylabel('Y')
zlabel('Z')
axis equal

end
```

A.2 xfem8isct.m

```
function [nsct,isct,xsct,ysct,zsct,levs] = xfem8isct(ex,ey,ez,levs)

% Intersection of hex8 based on level-set values.
%
% Input:   ex    : global x-coordinates of nodes of 3D element
%          ey    : global y-coordinates of nodes of 3D element
%          ez    : global z-coordinates of nodes of 3D element
%          levs  : level set values randomly generated
%
% Output:  nsct  : number of intersected edges
%          isect : vector flags of intersected edges
%          xsect : x-coordinates of intersections in global and local coordinates
%          ysect : y-coordinates of intersections in global and local coordinates
%          zsect : z-coordinates of intersections in global and local coordinates

% Map of node connections across edges in a 3D element.
% Edgmap is a 12 x 2 matrix.
edgmap = [ 1 2; 2 3; 3 4; 4 1; 1 5; 2 6; 3 7; 4 8; 5 6; 6 7; 7 8; 8 5];

% Values of nodes in master element (local coordinates).

xp = [-1 -1 -1 -1  1  1  1  1];
yp = [-1 -1  1  1 -1 -1  1  1];
zp = [1  -1  -1 1 1  -1  -1 1];

% Set initial value of intersected edges to zero.
% Create a zero 12 x 1 matrix to flag edges with intersected edges.
% Create a zero 12 x 2 matrix to record x, y, z coordinates in global and
% local coordinates.
nsct = 0;
isct = zeros(12,1);
xsct = zeros(12,2);
ysct = zeros(12,2);
zsct = zeros(12,2);

for i = 1:12
    % ic1 and ic2 return the values of the first and second columns of
    % edgmap, respectively, for a determined row. ic1 and ic2 represent two
    % nodes connected by a cube edge.
    ic1 = edgmap(i,1);
    ic2 = edgmap(i,2);
    % The level set values at those nodes are multiplied to determined
    % intersection.
    if levs(ic1)*levs(ic2) < 0      % If so, then there is an intersection.
        nsct = nsct+1;              % Increase number of intersected edges by one
    .
```

```
        isct(i) = 1;                % Flag edge as having an intersection in isct matrix.
        sctr = -levs(ic1)/(levs(ic2)-levs(ic1));    % Dimensionless location (ratio) of intersection.
        % Insection in glb coordinates
        xsct(i,1) = ex(ic1)+sctr*(ex(ic2)-ex(ic1));
        ysct(i,1) = ey(ic1)+sctr*(ey(ic2)-ey(ic1));
        zsct(i,1) = ez(ic1)+sctr*(ez(ic2)-ez(ic1));
        % Insection in local coordinates
        xsct(i,2) = xp(ic1)+sctr*(xp(ic2)-xp(ic1));
        ysct(i,2) = yp(ic1)+sctr*(yp(ic2)-yp(ic1));
        zsct(i,2) = zp(ic1)+sctr*(zp(ic2)-zp(ic1));
    end
end

    display('Number of intersected edges: ');
    disp(nsct);
    display('Edges with intersections: ');
    disp(isct');
    display('x-coordinates of intersections in global coordinates: ');
    disp(xsct(:,1));
    display('x-coordinates of intersections in local coordinates: ');
    disp(xsct(:,2));
    display('y-coordinates of intersections in global coordinates: ');
    disp(ysct(:,1));
    display('y-coordinates of intersections in local coordinates: ');
    disp(ysct(:,2));
    display('z-coordinates of intersections in global coordinates: ');
    disp(zsct(:,1));
    display('z-coordinates of intersections in local coordinates: ');
    disp(zsct(:,2));

end
```

A.3 xfem8tet.m

```
function [xtet,ytet,ztet,ctet,ptet,pnd,plist,Tetp1,Tetp2,tet1G,tet1L,tet2G,tet2L,size_phase1,size_phase2,volume1,volume2,total_volume1,total_volume2] = xfem8tet(isct,xsct,ysct,zsct,ex,ey,ez,levs)

% Intersection of hex8 based on level-set values.
%
% Input:  nsct   :  number of insected edges
%         isect  :  vector flags intersected edges
%         xsect  :  x-coordinates of insections in global and local coordinates
```

```
%          ysect  :  y-coordinates of insections in global and local coordinates
%          zsect  :  z-coordinates of insections in global and local coordinates
%          ex     :  global x-coordinates of nodes
%          ey     :  global y-coordinates of nodes
%          ez     :  global z-coordinates of nodes
%          levs   :  levs
%
% Output: xtet    :  x-coordinates of nodes of triangulated element
%          ytet   :  y-coordinates of nodes of triangulated element
%          ztet   :  z-coordinates of nodes of triangulated element
%          ctet   :  connectivity of tets in triangulated element
%          ptet   :  phase of each tetrahedron
%          pnd    :  nodal levelset values (actual values), # of nodes
%          plist  :  phase of each node (in terms of -1, 1 and 0)
%          Tetp1  :  nodes of tetrahedrons for main phase 1
%          Tetp2  :  nodes of tetrahedrons for main phase 2
%          tet1G  :  global coordinates of tetrahedron for phase 1
%          tet1L  :  local coordinates of tetrahedron for phase 1
%          tet2G  :  global coordinates of tetrahedron for phase 1
%          tet2L  :  local coordinates of tetrahedron for phase 2
%          size_phase1 : number of tetrahedrons in phase 1
%          size_phase2 : number of tetrahedrons in phase 2
%          volume1      : volume of a tetrahedron in phase 1
%          volume2      : volume of a tetrahedron in phase 2
%          total_volume1 : total volume of all tetrahedrons in phase 1
%          total_volume2 : total volume of all tetrahedrons in phase 2

%% Identify coordinates and phases to triangulate

% Nodes in local coordinates
xp = [-1 -1 -1 -1  1  1  1  1];
yp = [-1 -1  1  1 -1 -1  1  1];
zp = [1  -1  -1 1 1  -1  -1 1];

% Find intersected edges. Order of edges depends on configuration of edgmap
% isct flagged intersected edges with a 1
itr = find(isct>0);
% Sort nodes by main phase. Determine which nodes have positive or negative
% level-set values.
ip1 = find(levs<0);
ip2 = find(levs>0);

% Id-numbers of node at edge intersections, i.e. create pseudonodes
ips = 9:8+length(itr);

% Create phase vector for triangulated element. Vector contains level-set
% values at the 8 original nodes plus zero values for the new pseudonodes
pnd = [levs zeros(1,length(itr))];
```

```

% Create nodes for main phase 1
% Coordinates where nodes are negative plus coordinates of intersections in
% global coordinates
exp1 = [ex(ip1) xsct(itr,1)'];
eyp1 = [ey(ip1) ysct(itr,1)'];
ezp1 = [ez(ip1) zsct(itr,1)'];
% Nodes and pseudonodes numbers of main phase 1
ipx1 = [ip1 ips];

% Create nodes for main phase 2
% Coordinates where nodes are positive plus coordinates of intersections in
% global coordinates
exp2 = [ex(ip2) xsct(itr,1)'];
eyp2 = [ey(ip2) ysct(itr,1)'];
ezp2 = [ez(ip2) zsct(itr,1)'];
% Nodes and pseudonodes numbers of main phase 2
ipx2 = [ip2 ips];

% Combine triangulation points of main phase 1 and 2 in local and global
% coordinates
xtet = [ex xsct(itr,1)';xp xsct(itr,2)'];
ytet = [ey ysct(itr,1)';yp ysct(itr,2)'];
ztet = [ez zsct(itr,1)';zp zsct(itr,2)'];

% Same as xtet, ytet, ztet, but only with global coordinates. Useful for
% triangulation below.
exp = [ex xsct(itr,1)'];
eyp = [ey ysct(itr,1)'];
ezp = [ez zsct(itr,1)'];

%% Triangulate main phase 1 and 2 together
% If we triangulate main phase 1 and 2 separately,tetrahedron will
% superpose for some level-set combinations. We need to triangulate the
% entire element as a whole.

Tp = DelaunayTri(exp',eyp',ezp');
Tetp = Tp.Triangulation(:,:);

%% Separate triangulation into main phases 1 and 2

Tetp1 = zeros(1,4);
Tetp2 = zeros(1,4);
index1 = 1;
index2 = 1;
% If tetrahedron contains a negative node, it is phase 1. Phase 2,
% otherwise.
for i = 1:size(Tetp,1)
```

```
        if pnd(Tetp(i,1)) < 0||pnd(Tetp(i,2)) < 0||pnd(Tetp(i,3)) < 0||pnd(Tetp(i,4)) < 0
            Tetp1(index1,:) = Tetp(i,:);
            index1 = index1 + 1;
        else
            Tetp2(index2,:) = Tetp(i,:);
            index2 = index2 +1;
        end
end

%% Display coordinates of phase 1 tetrahedrons
% Display coordinates of phase 1 tetrahedrons in global coordinates
% Display volume of each phase 1 tetrahedron
total_volume1 = 0;
display('The global coordinates for the phase 1 tetrahedrons are: ')
for i = 1:size(Tetp1,1)
    tet1G = [xtet(1,Tetp1(i,1)),ytet(1,Tetp1(i,1)),ztet(1,Tetp1(i,1));xtet(1,Tetp1(i,2)),ytet(1,Tetp1(i,2)),ztet(1,Tetp1(i,2));xtet(1,Tetp1(i,3)),ytet(1,Tetp1(i,3)),ztet(1,Tetp1(i,3));xtet(1,Tetp1(i,4)),ytet(1,Tetp1(i,4)),ztet(1,Tetp1(i,4))];
    disp(tet1G)
    % Calculate volume. Algorithm: For 4 points in tetrahedron P,Q,R,S,
    % volume = abs(det([Q-P;R-Q;S-R]))/6
    a = tet1G(2,:)-tet1G(1,:);
    b = tet1G(3,:)-tet1G(2,:);
    c = tet1G(4,:)-tet1G(3,:);
    volume1 = abs(det([a;b;c]))/6;
    total_volume1 = total_volume1 + volume1;
    display('The volume of phase 1 tetrahedron is: ')
    disp(volume1)
end

% Display coordinates of phase 1 tetrahedrons in local coordinates
display('The local coordinates for the phase 1 tetrahedrons are: ')
for i = 1:size(Tetp1,1)
    tet1L = [xtet(2,Tetp1(i,1)),ytet(2,Tetp1(i,1)),ztet(2,Tetp1(i,1));xtet(2,Tetp1(i,2)),ytet(2,Tetp1(i,2)),ztet(2,Tetp1(i,2));xtet(2,Tetp1(i,3)),ytet(2,Tetp1(i,3)),ztet(2,Tetp1(i,3));xtet(2,Tetp1(i,4)),ytet(2,Tetp1(i,4)),ztet(2,Tetp1(i,4))];
    disp(tet1L)
end

% Display total volume of phase 1 tetrahedrons
    display('The total volume of phase 1 tetrahedrons is: ')
    disp(total_volume1)

%% Display coordinates of phase 2 tetrahedrons
% Display coordinates of phase 2 tetrahedrons in global coordinates
```

```
% Display volume of each phase 2 tetrahedron
total_volume2 = 0;
display('The global coordinates for the phase 2 tetrahedrons are: ')
for i = 1:size(Tetp2,1)
    tet2G = [xtet(1,Tetp2(i,1)),ytet(1,Tetp2(i,1)),ztet(1,Tetp2(i,1));xtet(1,Tetp2(i,2)),ytet(1,Tetp2(i,2)),ztet(1,Tetp2(i,2));xtet(1,Tetp2(i,3)),ytet(1,Tetp2(i,3)),ztet(1,Tetp2(i,3));xtet(1,Tetp2(i,4)),ytet(1,Tetp2(i,4)),ztet(1,Tetp2(i,4))];
    disp(tet2G)
    % Calculate volume.
    a = tet2G(2,:)-tet2G(1,:);
    b = tet2G(3,:)-tet2G(2,:);
    c = tet2G(4,:)-tet2G(3,:);
    volume2 = abs(det([a;b;c]))/6;
    total_volume2 = total_volume2 + volume2;
    display('The volume of phase 2 tetrahedron is: ')
    disp(volume2)
end

% Display coordinates of phase 2 tetrahedrons in local coordinates
display('The local coordinates for the phase 2 tetrahedrons are: ')
for i = 1:size(Tetp2,1)
    tet2L = [xtet(2,Tetp2(i,1)),ytet(2,Tetp2(i,1)),ztet(2,Tetp2(i,1));xtet(2,Tetp2(i,2)),ytet(2,Tetp2(i,2)),ztet(2,Tetp2(i,2));xtet(2,Tetp2(i,3)),ytet(2,Tetp2(i,3)),ztet(2,Tetp2(i,3));xtet(2,Tetp2(i,4)),ytet(2,Tetp2(i,4)),ztet(2,Tetp2(i,4))];
    disp(tet2L)
end

% Display total volume of phase 2 tetrahedrons
    display('The total volume of phase 2 tetrahedrons is: ')
    disp(total_volume2)

%% Display number of tetrahedrons in each phase.
size_phase1 = size(Tetp1(:,1));
size_phase1 = size_phase1(1);
size_phase2 = size(Tetp2(:,1));
size_phase2 = size_phase2(1);
display('The number of tetrahedrons in phase 1 is: ')
disp(size_phase1);
display('The number of tetrahedrons in phase 2 is: ')
disp(size_phase2);

%% Locate triangle interfaces - new algorithm
% Use function ismember to compare array vectors
% Display only result if three nodes repeat

display('The interfaces can be found on the triangles with nodes: ')
```

```
for i = 1:size(Tetp1,1)
    for j = 1:size(Tetp2,1)
        r = ismember(Tetp1(i,:),Tetp2(j,:));
        Tetp1_tri = Tetp1(i,:);
        Tetp1_tri = Tetp1_tri(r);
        if size(Tetp1_tri,2) == 3
            disp(Tetp1_tri)
        end
    end
end

%% Combine triangulation of nodes in ctet
% Size of ptet depends on number of tetrahedrons
% Phase 1 tetrahedrons have value of -1, phase 2 value of 1 in ptet

% Original:
% ctet=[Tetp1;Tetp2];
% ptet=[-ones(size(Tetp1,1),1);ones(size(Tetp2,1),1)];

% New method:
% Provide two options for triangulation
% Option 1: based on volume
% If one phase is significantly larger than the other, switch tetramesh
if total_volume2 <= (0.2*(total_volume1 + total_volume2))
    ctet=[Tetp2;Tetp1];
    ptet=[-ones(size(Tetp2,1),1);ones(size(Tetp1,1),1)];
else
    ctet=[Tetp1;Tetp2];
    ptet=[-ones(size(Tetp1,1),1);ones(size(Tetp2,1),1)];
end

% Option 2: based on user's choice
% option = input('Choose triangulation option 1 or 2: ');
% if option == 1
%     ctet=[Tetp2;Tetp1];
%     ptet=[-ones(size(Tetp2,1),1);ones(size(Tetp1,1),1)];
% elseif
%     ctet=[Tetp1;Tetp2];
%     ptet=[-ones(size(Tetp1,1),1);ones(size(Tetp2,1),1)];
% end

%% Check connectivity between phases. PART of ORIGINAL CODE

% Obsolete? plist produces the same value as pnd
% Check connectivity for main phase 1 and creat sub-phase information

% Matrix plisp1 formed of (nodes + pseudonodes) x 1 elements with value of
% -2. ipx1 represents location of phase 1 nodes + pseudonodes. At these
```

```
% locations, values are replaced by -1
plisp1 = -2*ones(size(xtet,2),1);
plisp1(ipx1) = -1;

domp1 = 0;
% While there are phase 1 nodes
while ismember(-1,plisp1) > 0
    domp1 = domp1 + 1;
    % Find where the negative nodes are in the nodes + pseudonodes vector
    % Value are progressively changed by 0, one at a time
    ppp = find(plisp1 == -1);
    plisp1(ppp(1)) = 0;
    % Locate where values were changed to zero and create new variable ppp
    % pid represents the node where value has transformed into zero
    while ismember(0,plisp1) > 0
        ppp = find(plisp1 == 0);
        pid = ppp(1);
        for it = 1:size(ctet,1)
            % If the node at pid belongs to Tetp1 and is negative
            if ismember(pid,ctet(it,:)) > 0 && ptet(it) < 0
                plisp1(ctet(it,:)) = max(0,plisp1(ctet(it,:)));
            end
        end
        % At the location of phase 1 nodes + pseudonodes, the value will be
        % replaced to 1.
        plisp1(pid) = domp1;
    end
end

% Check connectivity for main phase 2 and creat sub-phase information
% Functions the same as previous routine, but for phase 2
plisp2 = -2*ones(size(xtet,2),1);
plisp2(ipx2) = -1;

domp2 = 0;
while ismember(-1,plisp2) > 0
    domp2 = domp2+1;
    ppp = find(plisp2 == -1);
    plisp2(ppp(1)) = 0;
    while ismember(0,plisp2) > 0
        ppp = find(plisp2 == 0);
        pid = ppp(1);
        for it = 1:size(ctet,1)
            if ismember(pid,ctet(it,:)) > 0 && ptet(it) > 0
                plisp2(ctet(it,:)) = max(0,plisp2(ctet(it,:)));
            end
       end
       plisp2(pid) = domp2;
```

```
    end
end

% Build phase list including subphase information
% Replace all pseudonodes by 0
plisp1(ips) = 0;
plisp2(ips) = 0;

% Find location of the pseudonodes
idp1 = find(plisp1>0);
idp2 = find(plisp2>0);

% Create a zero (nodes + pseudonodes) x 1 matrix plist
plist = zeros(size(xtet,2),1);
% Create list that shows which original nodes belong to phase I and II
plist(idp1) = -plisp1(idp1);
plist(idp2) = plisp2(idp2);

end
```

A.4 number_configurations.m

```
function[] = number_configurations()
% This function computes all possible combinations of level-set function
% values at the corner nodes to obtain the number of 3D triangulation
% combinations possible.

% x,y,z coordinates of element.
% This will form a 3D cubic element cube.
ex = [0 1 1 0 0 1 1 0];
ey = [0 0 1 1 0 0 1 1];
ez = [0 0 0 0 1 1 1 1];

%% Sweep over all possible level-set configurations.

% Twelve edges. Maximum number of configurations:
maxc = 2^12;
icase = zeros(maxc,1);
hexsect = cell(127,8);

% Set initial configuration, k
% This loop calculates the total number of possible configurations of
% edge intersections
% Routine uses simple negative/positive level set values
k = 0;
```

```
for i1 = -1:2:1
    for i2 = -1:2:1
        for i3 = -1:2:1
            for i4 = -1:2:1
                for i5 = -1:2:1
                    for i6 = -1:2:1
                        for i7 = -1:2:1
                            for i8 = -1:2:2
                                levs = [i1 i2 i3 i4 i5 i6 i7 i8];
                                [nsct,isct,xsct,ysct,zsct,levs] = xfem8isct(ex,ey,ez,levs);
                                % isct is a 12 x 1 matrix representing each edge of the 3D cube element
                                % Since each edge can have one or zero intersections, the isct vector becomes a binary display
                                cbin = sprintf('%d%d%d%d%d%d%d%d%d%d%d%d',isct(1),isct(2),isct(3),isct(4),isct(5),isct(6),isct(7),isct(8),isct(9),isct(10),isct(11),isct(12));
                                % cbin is a binary number displayed as a string, and converted into the decimal cdec.
                                cdec = bin2dec(cbin);
                                % If icase is equal to zero, it means it is a new unique configuration and therefore,
                                % number of total configurations k should
                                % be increased by one
                                if icase(cdec+1) == 0 && nsct > 0
                                    k = k+1;
                                    [xtet,ytet,ztet,ctet,ptet,pnd,plist,Tetp1,Tetp2,tet1G,tet1L,tet2G,tet2L] = xfem8tet(isct,xsct,ysct,zsct,ex,ey,ez,levs);
                                    hexsect{k,1} = cdec;
                                    hexsect{k,2} = xtet;
                                    hexsect{k,3} = ytet;
                                    hexsect{k,4} = ztet;
                                    hexsect{k,5} = ctet;
                                    hexsect{k,6} = ptet;
                                    hexsect{k,7} = pnd;
                                    hexsect{k,8} = plist;
                                    [max(plist) min(plist)];
                                    icase(cdec+1)=1;
                                end
                            end
                        end
                    end
                end
            end
        end
    end
end
```

```
fprintf('Number of configurations = %d\n',k);

end
```

Bibliography

[1] Yazid Abdelaziz and Abdelmadjid Hamouine. A survey of the extended finite element. *Computers & Structures*, 86(11):1141–1151, 06 2008. URL http://search.ebscohost.com/login.aspx?direct=true&db=aph&AN=31919820&site=ehost-live. M3: Article.

[2] O. C. Zienkiewicz, Richard Lawrence Taylor, Perumal Nithiarasu, and J. Z. Zhu. *The finite element method*. Elsevier/Butterworth-Heinemann, Oxford; New York, 6th edition, 2005. ISBN 0750664312; 0750663200; 0750663219; 0750664312. O. C. Zienkiewicz, R. L. Taylor, P. Nithiarasu; of plates :ill. (some col.) ;25 cm; Previous ed.: 2000; Includes bibliographical references and indexes; [v. 1]. The finite element method: its basis and fundamentals – [v. 2]. The finite element method: for solid and structural mechanics – [v. 3]. The finite element method for fluid dynamics.

[3] A. Hansbo and P. Hansbo. A finite element method for the simulation of strong and weak discontinuities in solid mechanics. *Computer Methods in Applied Mechanics and Engineering*, 193(33-35):3523–3540, 2004. PT: J; UT: WOS:000223073300004.

[4] Thomas-Peter Fries and Ted Belytschko. The extended/generalized finite element method: An overview of the method and its applications. *International Journal for Numerical Methods in Engineering*, 84(3):253–304, OCT 15 2010. PT: J; NR: 207; TC: 63; J9: INT J NUMER METH ENG; PG: 52; GA: 667OA; UT: WOS:000283202200001.

[5] M. Stolarska, DL Chopp, N. Moes, and T. Belytschko. Modelling crack growth by level sets in the extended finite element method. *International Journal for Numerical Methods in Engineering*, 51(8):943–960, JUL 20 2001. PT: J; UT: WOS:000169624300003.

[6] D. T. Lee and B. J. Schachter. Two algorithms for constructing a delaunay triangulation. *International Journal of Computer & Information Sciences*, 9(3):219–242, 06/01 1980. URL http://dx.doi.org/10.1007/BF00977785. J2: International Journal of Computer and Information Sciences.