Received: 15 February 2022 | Revised: 17 May 2022 | Accepted: 18 May 2022

DOI: 10.1002/nme.7049

RESEARCH ARTICLE

WILEY

An object-oriented geometric engine design for discontinuities in unfitted/immersed/enriched finite element methods

Jian Zhang[1] | **Elena Zhebel**[1,2] | **Sanne J. van den Boom**[1,3] | **Dongyu Liu**[1] | **Alejandro M. Aragón**[1]

[1]Faculty of Mechanical, Maritime and Materials Engineering, Delft University of Technology, Delft, The Netherlands

[2]EZNumeric, 2517 JR, The Hague, The Netherlands

[3]Department of Structural Dynamics, Netherlands Institute for Applied Scientific Research (TNO), Delft, The Netherlands

Correspondence
Alejandro M. Aragón, Department of Precision and Microsystems Engineering, Faculty of 3mE, Delft University of Technology.
Email: a.m.aragon@tudelft.nl

Funding information
China Scholarship Council, Grant/Award Number: 201606060130

Abstract

In this work, an object-oriented geometric engine is proposed to solve problems with discontinuities, for instance, material interfaces and cracks, by means of unfitted, immersed, or enriched finite element methods (FEMs). Both explicit and implicit representations, such as geometric entities and level sets, are introduced to describe configurations of discontinuities. The geometric engine is designed in an object-oriented way and consists of several modules. For efficiency, a *k*-d tree data structure that partitions the background mesh is constructed for detecting cut elements whose neighbors are found by means of a dual graph structure. Moreover, the implementation for creating enriched nodes, integration elements, and physical groups is described in detail, and the corresponding pseudo-code is also provided. The complexity and efficiency of the geometric engine are investigated by solving 2-D and 3-D discontinuous models. The capability of the geometric engine is demonstrated on several numerical examples. Topology optimization and problems with intersecting discontinuities are handled with enriched FEMs, where enriched discretizations obtained from the geometric engine are used for the analysis. Furthermore, polycrystalline structures that overlap with an unfitted mesh are considered, where integration elements are created so they align with grain boundaries. Another example shows that the Stanford bunny, which is discretized by a surface mesh with triangular elements, can be fully immersed into a 3-D background mesh. Finally, we share a list of main findings and conclude that the proposed geometric engine is general, robust, and efficient.

KEYWORDS

discontinuities, enriched finite element methods, geometric engine, level set, mesh generator

1 | INTRODUCTION

Commercial softwares such as Abaqus, Ansys, and COMSOL, have become the industry standard for finite element (FE) numerical analysis. They require, however, FE meshes that fit the problems' geometries, that is, where the edges of finite

elements align with the boundary and other discontinuities such as material interfaces and/or cracks. This is a computationally expensive requirement: the creation of *fitted* or *geometry-conforming* meshes for complex geometries has been estimated to take about 80% of the total analysis time.[1] Furthermore, creating a good-quality mesh is a challenging endeavour that is prone to failure, especially for 3-D models.[2] Creating *fitted* meshes is troublesome and/or time-consuming, for instance, for fractures,[3-5] structural optimization,[6-8] fluid-structure interaction,[9,10] fiber/particle-reinforced composites,[11] and structures with irregular boundaries.[12,13] For these problems, transferring the complexity of mesh generation to the finite element formulation has been shown advantageous. Unfitted/immersed/enriched finite element methods (FEMs) allow for a complete decoupling between the problem geometry and discontinuities. Methods in this category include immersed FEM,[14,15] unfitted FEM,[16] the conformal decomposition finite element method (CDFEM),[17-19] and CutFEM.[20] This list is by no means exhaustive. A shared aspect among these methods that use (usually structured) unfitted discretizations is that interactions between a background mesh and the problem's geometry have to be performed. This work presents a general object-oriented *geometric engine* that can be used for mesh interactions with both implicit and explicit representations of discontinuities in all dimensions.

Even though the proposed work is applicable for unfitted/immersed/enriched FEMs, we place emphasis on the latter family of methods, for which the geometric engine was originally designed. Enriched FEMs have been vastly explored to treat discontinuities such as cracks and material interfaces. In the eXtended/Generalized Finite Element Method (X/GFEM),[21,22] which originates from the partition of unity method,[23,24] the standard finite element approximation is augmented by enrichment functions that reproduce a discontinuity in the primal field (e.g., for cracks) or its gradient (e.g., for material interfaces). However, although X/GFEM provides great flexibility in solving discontinuous problems with a *fixed* background mesh, the method is not without issues, including loss of accuracy in blending elements for certain choice of enrichment function,[25,26] the need for special formulations for prescribing non-homogeneous essential (Dirichlet) boundary conditions (BCs),[27,28] and lack of stability—that is, the condition number of the stiffness matrix may grow much faster than that of standard FEM with fitted discretizations.[29-31] There are, however, less known enriched finite element formulations that, while retaining the mesh-geometry decoupling feature of X/GFEM, are devoid of the aforementioned issues. The Interface-enriched Generalized Finite Element Method (IGFEM),[32] for instance, is a simpler enriched formulation that can be derived from X/GFEM.[33] IGFEM adds enriched degrees of freedom (DOFs) to nodes created along discontinuities—instead of associating them to nodes of the original FE mesh as done in X/GFEM. IGFEM recovers the standard finite element space and thus it can be seen as a method laying in between X/GFEM and FEM. IGFEM is intrinsically stable by a proper scaling of enrichment functions or the use of a simple diagonal preconditioner,[34,35] and nonzero essential BCs can be prescribed strongly.[36] Furthermore, multiple discontinuities within a single finite element can be handled via a hierarchical implementation.[35] The method can seamlessly resolve boundaries so it can be used to solve immersed boundary (fictitious domain) problems—and with smooth reactive tractions in Dirichlet boundaries.[36,37] In addition, IGFEM has been used to prescribe Bloch-Floquet periodic boundary conditions in the analysis of phononic crystals,[38] and for coupling non-conforming meshes and highly nonlinear contact problems.[39] In the context of topology optimization, IGFEM has been used for compliance minimization,[33] and for tailoring the fracture resistance in brittle structures.[40] Finally, IGFEM has also been generalized for the unified treatment of both weak and strong discontinuities in the Discontinuity-Enriched Finite Element Method (DE-FEM).[41] Since DE-FEM was first demonstrated for 2D fracture mechanics problems,[41] the method has been extended to 3D,[42] and has been developed to handle multiple intersecting discontinuities,[43] dynamic crack propagation,[44] and curved cracks and interfaces described by Non-Uniform Rational B-Splines (NURBS).[45] These enriched FEMs have been shown to outperform standard FEM for solving weak and/or strong discontinuous problems, including:

- **Multi-phase materials** Inclusions embedded in a matrix material with distinct properties (see Figure 1A) are prevalent in natural materials such as bone[46] and in artificial materials such as fiber-reinforced composites.[47] Multi-phase materials can also exhibit a microstructure where there is no apparent matrix phase, as is the case with polycrystalline microstructures (see Figure 1B). These materials, which are assemblies of small grains (crystals) with different material properties, include polycrystalline ceramics, metals, and alloys. Because of their common use in industrial applications, multi-phase materials require accurate numerical characterization. Material interfaces, which may have a critical effect on the macroscopic material behavior, have been modeled by assuming perfect bonding between interfaces (C^0–continuity)[48-50] and progressive degradation of the interface by means of cohesive models (C^{-1}–continuity).[43,51]
- **Solidification** Boundaries between phases can evolve as a consequence of temperature changes, for example, liquid to solid (the molten metal back into the solid state during metal casting). This solidification phenomenon plays an important role in material processing in many industries because it could lead to important variations in material properties

such as strength, thermal stability, and ductility.[52,53] It is therefore important to understand this phase transformation process so it can be precisely controlled.[54] Due to the complex nature of the problem, where the location of phase interfaces varies with time, only a limited number of analytical solutions are available.[55] Numerical techniques are therefore widely used to model the discontinuous temperature gradient field at phase interfaces.[56-60]

- **Two-phase flows** A flow consisting of different phases (see Figure 1C) is commonplace in industrial applications such as oil and gas in pipelines, and water and steam in nuclear reactor cooling systems.[61] Since these phases have different properties, the interface between them can result in discontinuous velocity/pressure fields (under a slip condition) and their corresponding gradient fields. This characteristic makes very challenging to solve two-phase flow problems in an analytical manner.[62,63] Although most two-phase flow formulations obtained are based on experimental data,[64] numerical simulation (based on enriched FEMs) also provides a viable alternative to describe the flow behavior.[65-68] Two important aspects should be carefully considered for obtaining an accurate approximation: A robust representation is necessary to describe the geometric configuration of interfaces, which could be altered significantly during the analysis. More importantly, appropriate enrichment functions should be constructed to account for the discontinuities in the primal/gradient fields along phase interfaces.
- **Fluid-structure interaction** Unlike two-phase flows, fluid-structure interaction (FSI) is a problem that couples fluid dynamics and solid mechanics (loosely referred henceforth as multiphysics).[69] There are multiple applications of FSI, such as dams,[70] aircrafts,[71] and even the cardiovascular system.[72] For solving the FSI problem, capturing accurately the interaction between fluid and solid, that is, both the fluid velocity and pressure at fluid-solid interfaces and the structure deformation, is a challenging endeavour for which enriched FEMs have proven to be effective.[73-75]
- **Fracture** Cracks (see Figure 1D), which could nucleate along a boundary or inside the material, may have a detrimental effect on the integrity of a structure and could even result in its failure.[76,77] Therefore, it is of utmost importance to properly characterize their effect, for which a plethora of works can be found on enriched FEMs alone. Cracks are usually modeled as strong discontinuities that introduce the kinematics of a discontinuous primal displacement field.[22,41] Enriched FEMs have been used to obtain stress intensity factors (SIFs) in stationary cracks,[42,78] and to model crack propagation by means of cohesive zone models[79,80] or by comparing the stress intensity to critical values.[21,22]
- **Dislocations** These are linear defects that generally occur in crystalline materials, for example, metals, diamonds, rocks, or ceramics.[81] Dislocations could result in phase transformations and grain growth, and even change material properties.[82] The geometrical configuration of a dislocation is represented by a glide plane with unit normal vector $\boldsymbol{n}$ (see Figure 1E), where either side of the glide surface shears with respect to another. Unlike cracks, the bare presence of dislocations can result in stresses in the material, even without applying external tractions.[83] Similarly to cracks, dislocations are treated as strong discontinuities, and the jump in the displacement field is described by the Burgers vector $\boldsymbol{b}$ that is tangent to the glide plane.[81] Therefore, similar considerations to fracture can be made when modeling dislocations.[84,85]
- **Shear bands** These are narrow zones with intense localized shearing (see Figure 1F) that are usually observed in soil,[86] rock,[87] ceramics,[88] and metal.[89] Created by strain localization, shear bands could lead to extreme deformations, instabilities, and even failure.[90] The influence of shear bands on material behavior has been widely investigated via theoretical and experimental studies.[87,91] Within the realm of enriched FEMs, a shear band can be modeled as a strong discontinuity when the width of the band relative to the medium is small,[92,93] or as two weak discontinuities otherwise.[94,95]

A common feature on the use of enriched FEMs to solve all aforementioned discontinuous problems is that a simple (usually structured) finite element mesh is used. Yet, this flexibility does not come for free: enrichment functions need to be created, which relies on the known location of discontinuities. In addition, the numerical quadrature of these discontinuous enrichment functions usually requires intersection tests between discontinuities and mesh elements, and the further subdivision of the latter into so-called *integration elements*. For some problems it is also required to keep track of the evolution of such discontinuities. In enriched FEMs all this functionality is handled by a geometric engine.

An appropriate geometric engine should be robust and work for an arbitrary number of discontinuity-mesh configurations. Still, designing a robust engine is extremely challenging, simply because it is not straightforward to write computational geometry functions that are robust when dealing with floating-point arithmetic. One may opt to use external libraries at the expense of increasing the software dependency. For instance, open-source libraries such as the Computational Geometry Algorithms Library (CGAL)[96] have been used. Ren and Younis[97] used CGAL to describe and

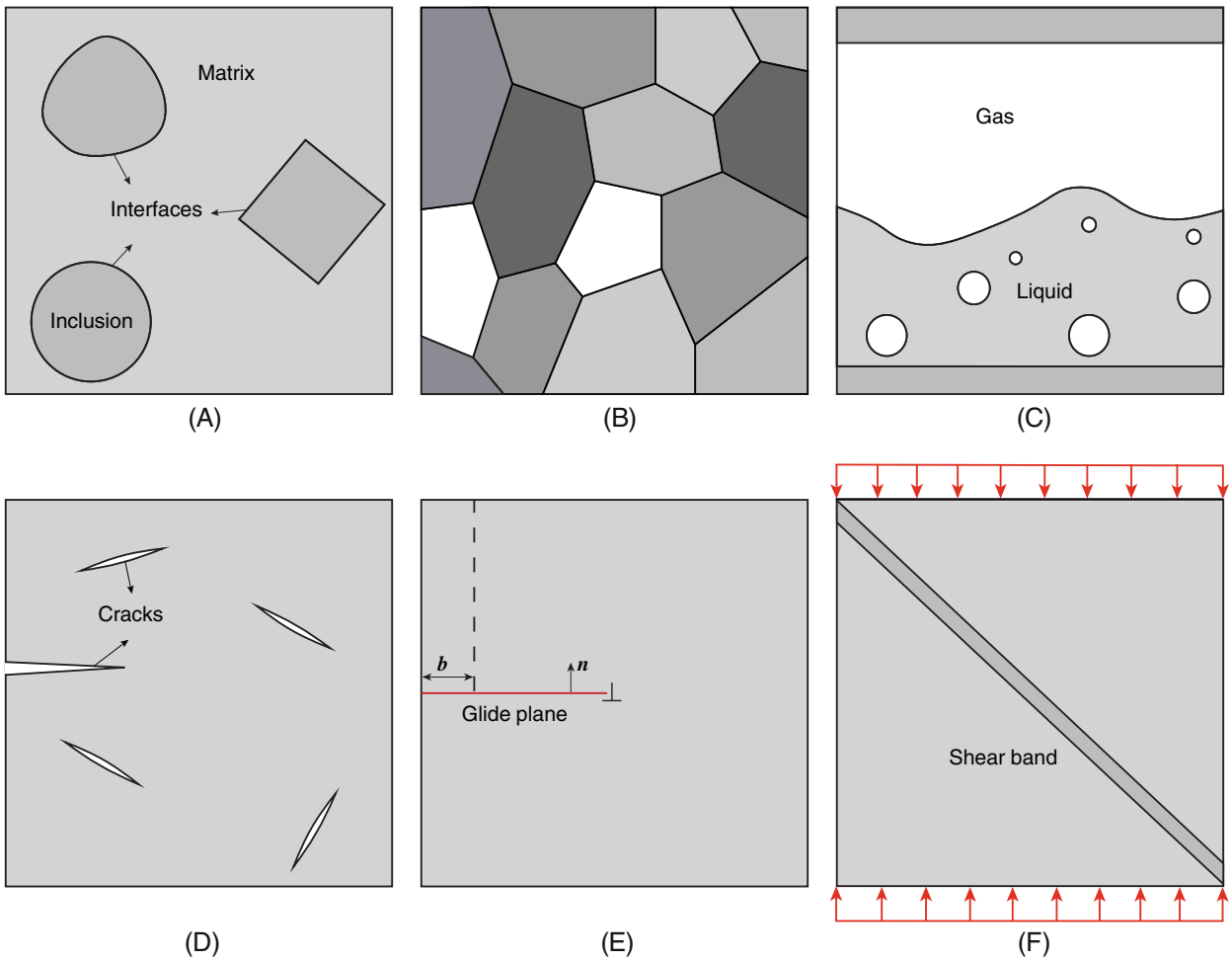

FIGURE 1 (A) Material inclusions embedded in a matrix with different material properties; (B) Polycrystalline materials, which are assemblies of different small grains (crystals); (C) Two-phase flow model where liquid and gas share an interface and interact with each other; (D) Cracks nucleate along the boundary and inside the material; (E) Dislocations lead to a discontinuous primal field where the jump is described by the Burgers vector $\boldsymbol{b}$; and (F) A shear band occurs under intense localized shearing

update the geometric description of cracks in hydraulic fracture propagation. Rather than using a single library, Massing *et al.*[98] developed a geometric package that uses data structures and algorithms from CGAL and GTS,[99] showing good performance in detecting intersections between two unfitted and overlapping meshes and integration over cut cells. These computational geometry libraries, however, make software's learning curve steeper and may not be necessary for some implementations (overkill). Therefore, implementing an in-house geometric engine is also a viable alternative and a popular one in the enriched FEM community. In the context of X/GFEM for fracture problems, Sukumar *et al.*[78,100] described in detail the geometric operations for detecting elements that intersect with cracks and the data structures used to store relevant geometric information. Instead of using level set functions to represent cracks, Pereira *et al.*[101] adopted a mesh with triangular elements to describe a crack surface explicitly in 3-D, and briefly described the strategy used for creating integration elements. Soghrati *et al.*[102] listed all possible situations when a tetrahedral element is completely split by a material interface, and explained the scheme for performing tetrahedralization of the resulting subdomains. For speeding up geometric operations, Březina and Exner[103] proposed a set of algorithms aiming at calculating intersections between 3-D background meshes and lower-dimensional simplices, whereby the number of calculations is minimized by reusing information obtained from neighboring elements. Yang *et al.*[104] proposed an approach for reconstructing an explicit description of heterogeneous materials based on scanning electron microscopy images, where NURBS are used to represent the morphology of fibers and bounding boxes are used to check for intersections between newly added and existing inclusions. Recently, Zhang *et al.*[42] discussed a basic geometric engine that is able to deal with multiple discontinuities cutting a single element hierarchically, whereby the entire hierarchy that results from splitting completely or

partially a tetrahedron is stored in an ordered tree data structure. All these works, however, just provide a *sufficient ad hoc* solution for their particular problem. As a result, these ideas cannot be easily generalized. In addition, although effective, most works do not discuss efficiency of the computational geometry routines. Although bits and pieces on implementing a robust geometric engine to be used with unfitted/immersed/enriched FEMs can be found in the literature, a detailed discussion is still missing.

In this article we describe the objected-oriented design of a geometric engine that can be used to solve problems with discontinuities by means of enriched FEMs. Although discussed in the context of interface- and discontinuity-enriched FEMs, the engine is general and thus could also be used together with any unfitted/immersed/enriched FEM without much modification. The engine's main functions are: (i) Performing intersection tests between discontinuities and finite elements; (ii) Creating enriched nodes that are collocated at the intersection between discontinuities and sides of background elements; (iii) Creating integration subdomains considering multiple discontinuities intersecting a single element; (iv) Storing such hierarchies of integration elements in an ordered tree data structure; and (v) Creating physical groups that are then used to assign correct material properties and/or to refer to discontinuities. The design of the engine goes beyond basic requirements in many areas, including:

- **Generality** Discontinuities can be represented both implicitly (e.g., via level set functions) and/or explicitly (e.g., line segments, polygons, or even lower-dimensional meshes). Integration subdomains in all dimensions can be created by rules or by more advanced techniques such as Delaunay triangulation (tetrahedralization);[105,106]
- **Robustness** The use of tolerances is mitigated wherever possible since tests based on them are prone to fail;
- **Efficiency** Given a coordinate, a space partitioning method (concretely a k-d tree) is used to find the background mesh element that contains it. In addition, a dual graph data structure of the background mesh is used to efficiently discover neighbors of cut elements (and to avoid intersection checks). Data on subdomains of cut elements' edges (or faces) is reused for generating subcells of neighboring elements. An ordered tree data structure is used to store hierarchical element information and to efficiently iterate over quadrature elements (leaves of the tree). Finally, we exploit a flood-fill algorithm that works in conjunction with a dual graph data structure to determine which (background and integration) elements belong to a corresponding physical domain.

The design of the geometric engine is thoroughly explained with pseudo-code, and its computational complexity is investigated with several 2-D and 3-D examples of material interfaces and cracks. The capability of the proposed engine is illustrated by means of several examples. First, the topology of time-dependent discontinuities represented as a level set is optimized for minimal structural compliance in 3-D. Next, a crack junction problem, where cracks intersect at the same location, is solved via DE-FEM. Later, a discontinuous model that includes intersected cracks and material interfaces is investigated. Furthermore, we overlap structured meshes with polycrystalline structures described by polygons/polyhedra, where integration elements created align with the grain boundaries. Finally, the Stanford bunny discretized by a surface mesh with triangular elements is fully immersed into a 3-D background mesh, yielding a new volumetric discretization with an exterior boundary that contains more triangular elements.

2 | GEOMETRIC ENGINE

The proposed object-oriented design of the geometric engine is composed of several modules, as schematically illustrated in Figure 2:

i) `Discontinuities` $\boldsymbol{D}$ is a collection of discontinuities, which can be represented implicitly (e.g., by means of a level set function) or explicitly (e.g., by means of geometric entities (see Section 2.1);
ii) `SpacePartitioner` $\mathbf{S}$ is a k-d tree data structure for partitioning the space occupied by the background mesh, enabling fast search of background mesh element(s) containing a given point along a discontinuity (see Section 2.2);
iii) `DualGraph` $\mathbf{G}$ is a graph data structure of the background mesh for fast neighboring element search (see Section 2.3);
iv) `Interactor` is used for processing the interactions between the background mesh and discontinuities (see Section 2.4);
v) `Intersector` finds intersection points between discontinuities and the sides of background mesh elements (see Section 2.5);

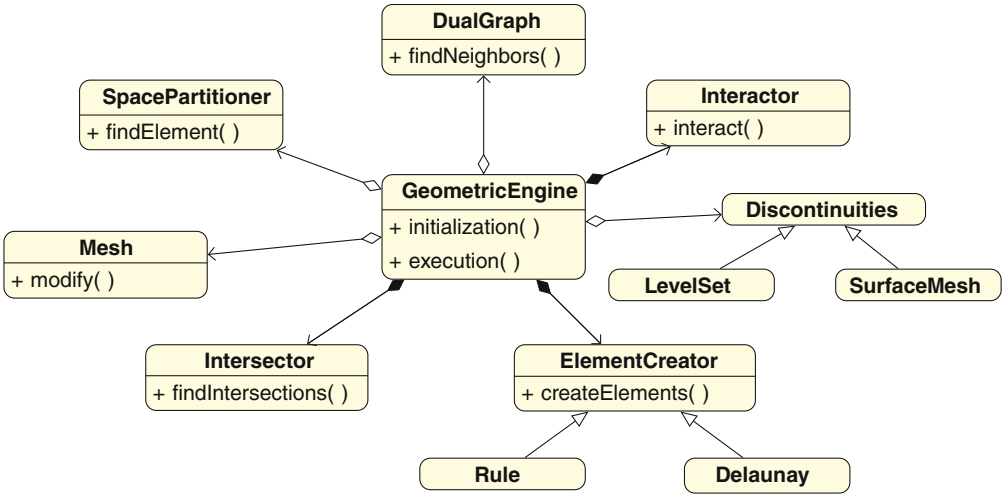

FIGURE 2 Diagram of modules composing the geometric engine represented the unified modeling language (UML) notation.

vi) `ElementCreator` creates lower-dimensional and integration elements (see Section 2.6);
vii) `Mesh` $\mathbf{M}$ contains data structures that store data regarding finite element nodes $\mathcal{N}$, elements $\mathcal{E}$, and physical groups $\mathcal{P}$ (where geometrical entities are combined into more meaningful groups, for example, mathematical, functional, or material properties[107]). $\mathbf{M}$ is updated via adding enriched nodes and new created elements and creating new and modifying original physical groups (see Section 2.7).

In addition, an associative container $\mathbf{I}$ keeps information about enriched nodes and sub-domains of cut element edges and/or faces, and aids in creating integration elements of any cut element sharing the same cut edge/face. A hierarchical ordered tree data structure $\mathbf{T}$ is used for storing cut mesh elements and their children. This tree enables fast traversal of integration elements, which is required for computing their local contributions to system matrices and vectors. Pseudo-code for the behavior of this geometric engine is given in Algorithm 1.

Algorithm 1. Geometric engine: Pre-process a background mesh for the use in the finite element analysis, such as DE-FEM

Input: Background mesh $\mathbf{M} := \{\mathcal{N}, \mathcal{E}, \mathcal{P}\}$ (node, element and physical group sets), discontinuity set $\mathcal{D}$, physical group set $\mathcal{P}_{\text{new}}$, and tolerance η

function initialization($\mathbf{M}$)
  $\mathbf{S} \leftarrow$ SpacePartitioner($\mathbf{M}$) – *Create a k-d tree partitioning of the background mesh*
  $\mathbf{G} \leftarrow$ DualGraph($\mathbf{M}$) – *Create a dual graph based on the background mesh*
  return $\mathbf{S}, \mathbf{G}$
end function

function execution($\mathbf{M}, \mathbf{S}, \mathbf{G}, \mathcal{D}, \mathcal{P}_{\text{new}}, \eta$)
  $\mathbf{I}, \mathbf{T} \leftarrow \emptyset, \emptyset$ – *Initialize enrichment container and tree data structure*
  for $d_i \in \mathcal{D}$ **do** – *Loop over discontinuities*
    $\{\mathbf{I}, \mathbf{T}\} \leftarrow$ Interactor.interact($\mathbf{S}, \mathbf{G}, d_i, \mathbf{I}, \mathbf{T}, \eta$) – *Interact the background mesh with the discontinuity*
  $\mathbf{M} \leftarrow \mathbf{M}$.modify($\mathcal{D}, \mathcal{P}_{\text{new}}, \mathbf{I}, \mathbf{T}$) – *Modify mesh with enriched nodes, elements and physical groups*
  return $\mathbf{M}$
end function

Output: Modified mesh $\mathbf{M} := \{\mathcal{N}, \mathcal{E}, \mathcal{P}\}$

2.1 | Discontinuities

Both implicit and explicit descriptions are allowed for representing the geometry of discontinuities. While the former can be achieved, for instance, by means of a level set function, the latter is attained by using geometric entities in the form of a finite element mesh—even for fundamental geometric primitives. Since these two representation methods are fundamentally different, a separate implementation is required to deliver the same functionality, namely for determining intersections between the background mesh and discontinuities, and for detecting background and integration elements inside or outside discontinuities.

2.1.1 | Level set functions

Level sets provide a flexible way to model moving/evolving boundaries.[108] They have been widely used in many fields, such as multi-phase flows,[109] image processing,[110] and topology optimization.[111,112] A level set function $\phi(\boldsymbol{x})$ describes boundaries in an implicit form as the iso-contour of a smooth function, for instance the zeroth level contour. Then, $\phi(\boldsymbol{x})$ defined in domain D is expressed as

$$\begin{cases} \phi(\boldsymbol{x}) = 0 \Leftrightarrow \boldsymbol{x} \in \partial\Omega, \\ \phi(\boldsymbol{x}) < 0 \Leftrightarrow \boldsymbol{x} \in \Omega, \\ \phi(\boldsymbol{x}) > 0 \Leftrightarrow \boldsymbol{x} \in (D \setminus \overline{\Omega}), \end{cases} \tag{1}$$

where $\boldsymbol{x}$ is the Cartesian coordinate, and Ω represents a subset of D with closure $\overline{\Omega}$ (see Figure 3A). Level sets can be used to represent a straight boundary, a circular material interface (shown in Figure 3B) and even the structural topology with more complex geometric configurations.[113] In the proposed geometric engine, level sets are used to describe both static material interfaces in multi-phase materials, and the evolving boundaries in topology optimization. For the latter, radial basis functions (RBFs) are used to discretize the level set function,[114,115] and coefficients multiplying RBFs are considered as design variables that are updated during the optimization process.

2.1.2 | Surface mesh

Explicit representations based on geometric entities, such as segments, triangles, rectangles, and polygons, are used to describe both weak and strong discontinuities in form of a surface mesh with lower dimension compared to that of the background mesh. As shown in Figure 4A, a 1-D mesh consisting of line elements (marked by red segments) is used to describe eight cracks intersecting a 2-D square domain. Figure 4B shows the representations of discontinuities in 3-D, whereby a surface mesh with four rectangular elements that can either represent a crack or a material interface, splits a cubic domain completely.

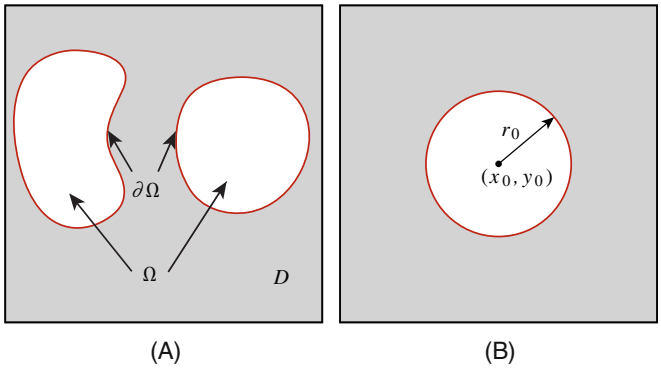

FIGURE 3 (A) Level set function $\phi(\boldsymbol{x})$ defined in a domain D with an embedding domain Ω, and the shape of interface $\partial\Omega$ (marked by a red curve) represented as $\phi(\boldsymbol{x}) = 0$; (B) Level set function $\phi(x, y) = \sqrt{(x - x_0)^2 + (y - y_0)^2} - r_0 = 0$ used to describe a circular interface in 2-D

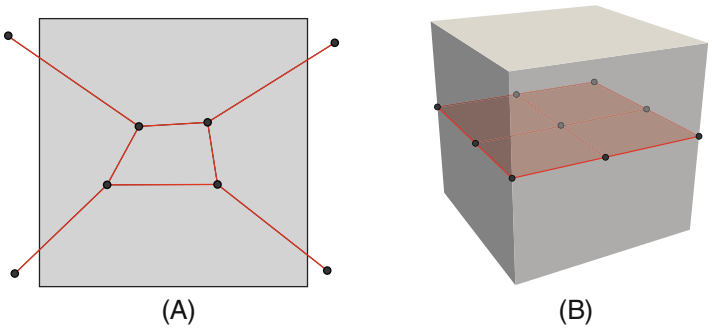

FIGURE 4 (A) A 1-D mesh with eight line elements and eight nodes that intersects within a square domain; (B) A 2-D mesh with four rectangular elements and nine nodes that splits a cube completely can be used to describe a material interface or a crack.

2.2 | Space partitioner

`SpacePartitioner` constructs a k-d tree data structure $\mathbf{S}$ for partitioning the k-dimensional space occupied by the background mesh $\mathbf{M}$. As with a general k-d tree for locating points in space, the k axes are cycled as the tree is constructed. Each non-leaf node in the tree is defined as a plane perpendicular to the specific axis, dividing the space into two parts at either side of the plane. It is worth noting that elements intersected with the plane are assigned to both parts. The coordinate associated with non-leaf node is calculated as the average of elemental centers along the corresponding axis. The tree construction is finalized when the element set at either side of the splitting plane remains the same, and then these elements are stored in the tree as leaf nodes. This k-d tree is used for finding a background element enclosing a given point, an operation that is carried out by the function `findElement`. Given a point, we traverse the tree from the root and we use the point coordinates to guide the search until a leaf node is found containing potential enclosing elements. Then geometric predicates are performed to check which element (or elements in case the point lies at an edge shared by two elements) contains the point.

As an example, we show in Figure 5A the space partitioning of a square 2×2 domain discretized by a background mesh with $3 \times 3 \times 2$ triangular elements; the range for both x and y axes is $[-1, 1]$. For the corresponding k-d tree data structure shown in Figure 5B, the root node divides the space with the plane $x = 0$. This plane divides the background mesh into two groups of elements $\{0, 1, 2, 3, 6, 7, 8, 9, 12, 13, 14, 15\}$ and $\{2, 3, 4, 5, 8, 9, 10, 11, 14, 15, 16, 17\}$ to the left and right sides, respectively. Notice there are several elements that belong to both groups since they are cut by the plane, namely $\{2, 3, 8, 9, 14, 15\}$. For the next level in the hierarchy the plane $y = 0$ is used, which further subdivides elements in the previous level to either side. For example, the left subtree of the root is split into $\{0, 1, 2, 3, 6, 7, 8, 9\}$ and $\{6, 7, 8, 9, 12, 13, 14, 15\}$. For the former element group, in the next hierarchical level a plane $x = -0.3333$ is obtained as the elements' centroid. Then this element group is divided into $\{0, 1, 6, 7\}$ and $\{2, 3, 8, 9\}$. For the next hierarchical level the $y = -0.3333$ is computing, further subdividing elements into $\{0, 1\}$ and $\{6, 7\}$, which are finally stored as leaves in the three. The subdivision of other element groups follows the same procedure. Finally, we obtain a k-d tree with five levels, with each leaf node containing two elements. In order to detect a point with coordinates $(-0.25, -0.25)$ (marked with a red circle in Figure 5A), we follow the red path shown in the tree (see Figure 5B), till arriving to the element set $\{8, 9\}$, after which we find the point is contained in the 8th element using simple predicates.

2.3 | Dual graph

`DualGraph` is an object that represents a dual graph data structure $\mathbf{G} := \{\mathcal{V}, \mathcal{S}\}$ (composed of sets of vertices $\mathcal{V}$ and edges $\mathcal{S}$) of a given background mesh $\mathbf{M}$. In this graph, vertices v_i, v_j represent the ith and jth finite elements, respectively, and they are connected with an edge if the elements share an edge. For constructing this graph structure, the background elements are iterated and an auxiliary data structure is used to store elements' edges that are later checked for overlap with other elements. Figure 6 shows a background mesh with $3 \times 3 \times 2$ triangular elements and its corresponding dual graph structure, which clearly displays with edges adjoining elements. This undirected graph is then used to find neighbors of any given target element via the function `findNeighbors`.

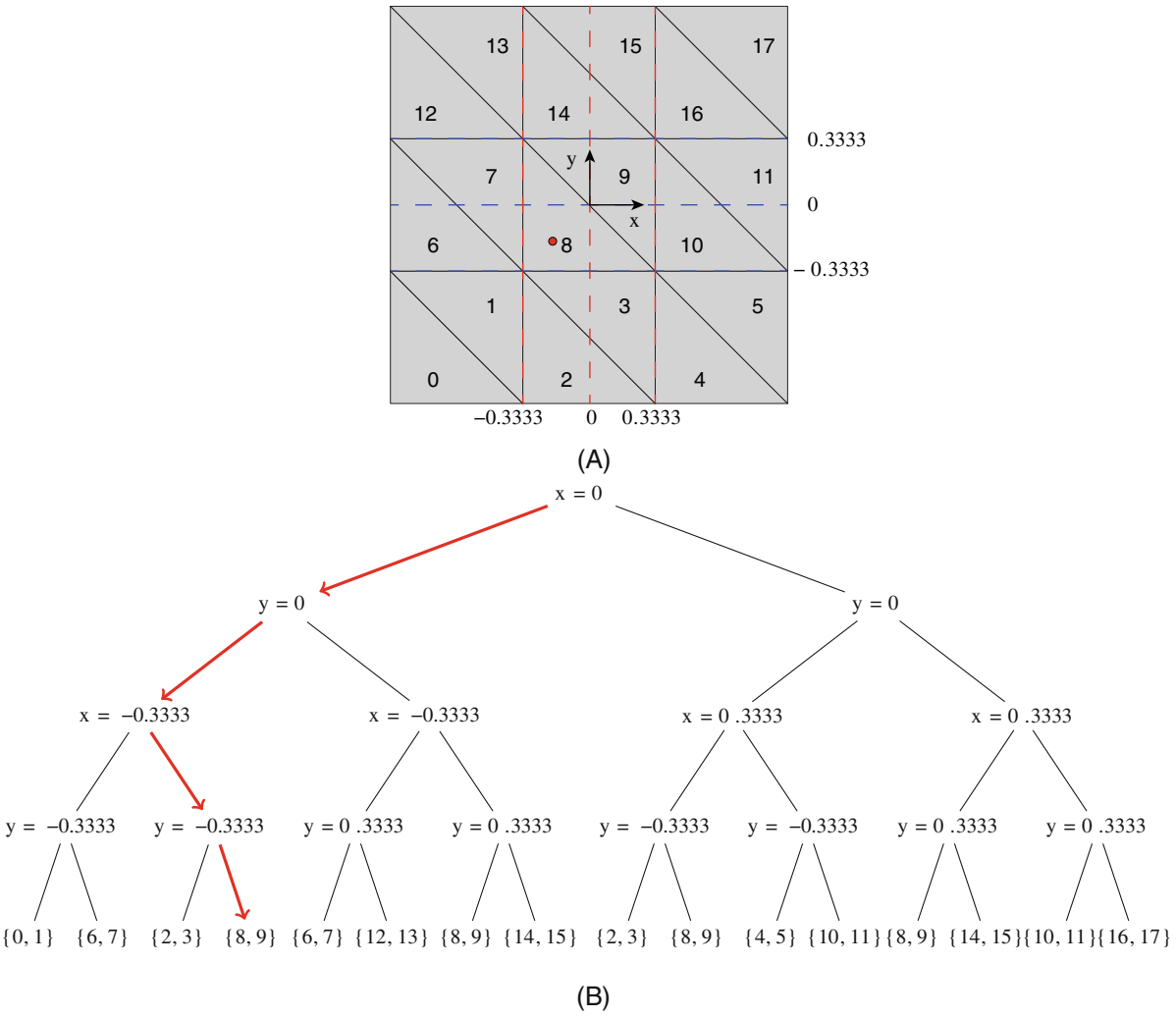

FIGURE 5 (A) The background mesh consisting of $3 \times 3 \times 2$ triangular elements shown with numbers, where a plane perpendicular to different axes is represented as a dashed line with different colors; (B) The corresponding k-d tree data structure, where the red lines with arrows show the path to find the elements enclosing the point $(-0.25, -0.25)$.

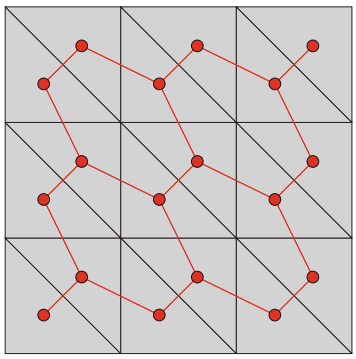

FIGURE 6 The background mesh consisting of $3 \times 3 \times 2$ triangular elements with the corresponding graph structure (shown with red vertices and edges).

2.4 | Interactor

`Interactor` takes care of processing intersections between discontinuities and the background FE mesh. `Intersector` is used to find intersections between discontinuities and elements, forwarding that information to `ElementCreator` to create children elements, and for storing the newly created elements in the ordered tree data structure **T**. For bookkeeping purposes, an enrichment associative container **I** is used, where the keys are cut element edges and/or faces. Their corresponding values include enriched nodes and newly created edges and/or faces (since original element edges and/or faces are partitioned). Keeping these data facilitates the creation of integration elements: If an element side has already been split into several subparts, we directly retrieve these from **I** for creating children of the background element sharing this cut edge or face. Although it is possible to create integration elements of a cut background element immediately after finding it intersects a discontinuity, we defer this operation to a later time so that all integration elements are created with a single function call. To that end, we process all discontinuity-element intersections with the aid of a queue **Q** that is updated as new cut elements are found (by looking at neighboring elements of the cut element being processed), and we keep track of cut elements with a set $\mathcal{W}$. This process ends when the queue is empty, which implies that all cut elements have been processed and that we are ready to create integration elements. The pseudo-code of function `interact` in `Interactor` is given in Algorithm 2. First, looping over a list of points that describe a given discontinuity d_i, we detect elements e that intersect with this discontinuity by calling function `findElement` in `SpacePartitioner` **S**, and store each cut element into the queue **Q** and the set $\mathcal{W}$. Second, the queue **Q** is iterated over cut elements e_i, and `Intersector` is called to calculate the intersections p_i of the cut element e_i with the given discontinuity d_i. Afterwards, enriched nodes are created at these intersections and stored in the enrichment associative container **I** together with their corresponding cut element's edges/faces. The function `findNeighbors` in `DualGraph` **G** is then used to find neighbors of the cut element. If these neighboring elements are not yet in the set $\mathcal{W}$ or are not already processed, they are added subsequently to the queue **Q** and set $\mathcal{W}$. We then continue to process stored components in the queue until **Q** becomes empty. As all intersections between background elements and the discontinuity d_i are detected, the function `createElements` in `ElementCreator` is called to create children elements, which include both integration elements and lower-dimensional elements along the discontinuity. Meanwhile, these elements e_h are stored into the tree data structure **T**, where they are set as children of the parent cut background elements. This tree data structure **T** will be later used to update the background mesh data.

2.5 | Intersector

`Intersector` is the object used to find intersections between element edges (faces in 3-D) and a given discontinuity. Also, if applicable, it also detects surface mesh nodes that belong to discontinuities inside/outside background elements. The corresponding pseudo-code of the function `findIntersections` is given in Algorithm 3, where enriched nodes are created and stored in the enrichment map **I**. Depending on the dimension of the background mesh, different approaches are implemented to detect element-discontinuity intersections. The procedure used is determined by the type of discontinuity since finding intersections for explicit representations follows a different approach than that used for implicit representations. For simplicity, we assume that the background mesh consists of triangular (tetrahedral) elements in 2-D (3-D).

For a level set implicit representation, the approach is quite straightforward as it remains the same regardless of the mesh dimension. First, we iterate over every edge of a background element and calculate level set values of their corresponding nodes. If the level set signs are not the same, the edge is crossed by a discontinuity at the zero level set value. The location of intersection point is then simply determined by interpolation.

For a crack described by a surface mesh, the corresponding procedure is based on algorithms for finding segment-segment (in 2-D) or segment-plane (in 3-D) intersections. To illustrate this type of intersection, consider in Figure 7 a discontinuity represented explicitly via a triangular (T3) element intersects a tetrahedral (T4) element. In this situation, the tetrahedral element is considered as a 3-D geometric entity that is composed of a collection of four triangular faces (2-D geometric entities) and six edges (1-D geometric entities). Therefore, several computational geometry tests are conducted to find the intersections. First, we detect whether any of the discontinuity nodes lie inside the tetrahedron (see a blue node in Figure 7). Then we create the corresponding enriched node at the same location and add it to the enrichment associative container **I**. Second, we iterate over the four triangular faces and determine

Algorithm 2. Interact a background mesh with a discontinuity

Input: Dual graph $\mathbf{G}$, k-d tree data structure $\mathbf{S}$, ith discontinuity d_i, tree data structure $\mathbf{T}$, enrichment associative container $\mathbf{I}$, and tolerance η

```
function interact(G, S, d_i, I, T, η)
    e ← S.findElement(d_i.points)                          – Obtain elements intersected with the discontinuity
    Q, W ← ∅, ∅                                            – Initialize a queue and a set
    Q, W ← addToQueue(e), addToSet(e)                      – Add cut elements to queue and set
    do
        e_i ← getQueue(Q)                                  – Get a cut element and remove it from queue
        x_i, I ← Intersector.findIntersections(e_i, d_i, I, η)   – Obtain intersections and update enrichment container
        if x_i ≠ ∅ then
            e_n ← G.findNeighbors(e_i)                     – Obtain the neighboring elements
            if e_n ∉ W then
                Q, W ← addToQueue(e_n), addToSet(e_n)      – Add neighboring elements to queue and set
    while Q ≠ ∅
    for e_i ∈ W do
        e_h ← ElementCreator.createElements(e_i, d_i, I)  – Create children elements
        T ← addToTree(e_h, e_i, T)                         – Add children elements to the tree data structure
    return I, T
end function
```

Output: Enrichment associative container $\mathbf{I}$ and tree data structure $\mathbf{T}$

Algorithm 3. Find intersections between a background element and a discontinuity

Input: ith background element e_i, ith discontinuity d_i, enrichment associative container $\mathbf{I}$, and tolerance η

```
function findIntersections(e_i, d_i, I, η)
    dim ← e_i.dimension                                    – Get the element dimension
    do
        x_i, I ← d_i.findIntersections(e_i, I, η)          – Obtain the intersections
        dim ← dim − 1                                      – Check the lower-dimensional geometric entity
    while dim ≠ 0
    return x_i, I
end function
```

Output: Intersections $\boldsymbol{x}_i$ and enrichment associative container $\mathbf{I}$

whether they are intersected by any of the three edges of the discontinuity's triangle. Enriched nodes are then created at intersection locations found (see red nodes in Figure 7) and stored into the enrichment associative container $\mathbf{I}$ with linking to the corresponding triangular face; this data will be used when computing intersections in the neighboring element sharing the same face. Finally, intersections between the six edges and the discontinuity's triangular surface element are determined (see a white node in Figure 7). As before, new enriched nodes created are added to the enrichment associative container $\mathbf{I}$ and linked to the corresponding tetrahedral edges. Obviously, it is more involved

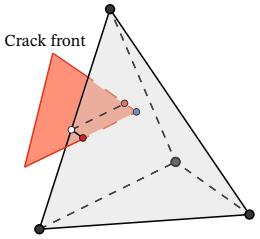

FIGURE 7 A crack represented by a triangular surface element (marked with red) intersects a tetrahedral element. Intersection tests are conducted to determine nodes lying inside the tetrahedron (marked with a blue node) and along its boundaries (see red nodes located at two surfaces and a white node located within an edge).

to handle explicit discontinuities than implicit ones since more operations are required. Noteworthy, this procedure works for any (surface and background) element type, for instance, quadrilateral (Q4) or hexahedral (H8) elements. Although the current version of our geometric engine cannot handle discontinuities with curved edges/faces, this scenario could be handled effortlessly by implementing other `findIntersections` functions. For instance, curved interfaces and cracks described by Non-Uniform Rational B-Splines (NURBS) have already been handled in DE-FEM,[45] whereby intersections between the background mesh and discontinuities were found by means of a Newton–Raphson solver.

Here we show a special situation that highlights the importance of setting the tolerance η when intersecting a background element. Consider in Figure 8 a tetrahedral element with connectivity $\{1,2,3,4\}$ that intersects a rectangular discontinuity with connectivity $\{a,b,c,d\}$, described explicitly by means of a surface mesh. As mentioned in `Intersector`, four triangular faces $\{1,2,3\}$, $\{2,3,4\}$, $\{1,3,4\}$, and $\{1,2,4\}$ are iterated to determine whether they intersect the four edges of the surface element $\{a,b\}$, $\{b,c\}$, $\{c,d\}$, and $\{d,a\}$. First, two intersections are found between $\{d,a\}$ and $\{1,3,4\}$ based on the coordinates of the background and surface mesh nodes. Then, other two intersections are detected when intersecting $\{d,a\}$ with $\{1,2,4\}$ and $\{2,3,4\}$. Since these intersections are found twice, it is necessary to remove duplicate points by checking their coordinates. Although these intersections should overlap exactly—for instance the intersection between $\{d,a\}$ and $\{1,3,4\}$, and between $\{d,a\}$ and $\{1,2,4\}$ (the red/white node in the left side of Figure 8)—the floating precision computation makes their coordinates slightly different. Therefore, we keep one intersection as long as the distance between these intersections is within the given tolerance. Next, intersections between the background element edges $\{1,2\}$, $\{2,3\}$, $\{3,1\}$, $\{1,4\}$, $\{2,4\}$, $\{3,4\}$ and the surface element $\{a,b,c,d\}$ are determined (see red/white and white nodes in Figure 8). Again, note that intersections between the rectangle $\{a,b,c,d\}$ and edges $\{1,4\}$ and $\{3,4\}$ overlap with those detected in the first step. After checking their coordinates, these intersections should not be taken into account when creating enriched nodes. As shown in Figure 8, only three enriched nodes $\{5,6,7\}$ should be created for this case, although more than three intersections were detected.

If we do not set an appropriate tolerance for checking overlapping intersections, enriched nodes with slightly different coordinates could be created (in this example enriched nodes $\{5,8\}$ and $\{6,9\}$); this could cause errors when

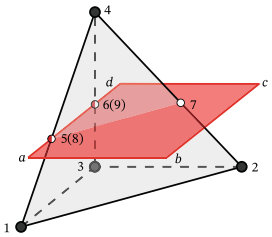

FIGURE 8 A background element with connectivity $\{1,2,3,4\}$ intersects with a discontinuity described by a surface mesh containing a rectangular element with connectivity $\{a,b,c,d\}$. Full and half white nodes are intersections between the background element edges and the rectangular element, and red/white nodes are intersections between the surface of the background element and the surface element edges.

creating integration subdomains. Moreover, both standard and created enriched nodes are considered in creating integration elements, with the possibility of creating tiny elements such as $\{1, 3, 5, 8\}$ and $\{4, 5, 6, 9\}$. This could cause problems when calling the Delaunay algorithm.

2.6 | Element creator

`ElementCreator` uses two different strategies to create elements, that is, lower-dimensional and integration elements are created based on specific rules or via a certain algorithm. The former, represented by the object `Rule`, considers all possible cut situations and sets them as templates for generating lower-dimensional and integration elements. The latter could use, for instance, Delaunay triangulation (tetrahedralization),[105,106] which is represented accordingly by a `Delaunay` object.

We explain the procedure for creating new elements using Delaunay's algorithm which is the only technique to handle complex situations where elements are not fully split by discontinuities. Since creating 2-D triangular sub-domains is included in the procedure that constructs a discretization in 3-D, we focus on the description of the latter. Algorithm 4 shows pseudo-code for the algorithm used to create lower-dimensional and integration elements. For clarity, we denote a cut background element as the *parent element* and integration elements that belong to it as *children* or *integration* elements. Figure 9A shows a tetrahedral element intersected by a discontinuity represented as a surface mesh composed of three quadrilateral (Q4) elements (marked with different colors). We first create lower-dimensional elements along the discontinuity, so surface elements describing the discontinuity are split into smaller subdomains in the function `createDiscontinuityElements`. Since enriched nodes along the discontinuity had already been stored by `Intersector` in the enrichment associative container **I**, it is now straightforward to create these lower-dimensional elements. In addition, these elements are set as internal faces within a parent element for creating children elements using the function `createIntegrationElements`. If these lower-dimensional elements are not considered when creating integration elements, an incorrect situation could emerge, where internal faces do not match the original surface elements (see Figure 9A). Later, triangular faces of this tetrahedron that are cut by surface elements are handled, and the corresponding sub-faces are created by using Delaunay triangulation. Noteworthy, it is necessary to trace intersected segments created between each surface element and the triangular faces, which are shown as red segments in Figure 9B. After obtaining children faces, we store them into the enrichment associative container **I**. For the neighboring element sharing the same cut face, these segments will be retrieved and not recreated ensuring consistency of the discretization. After all cut triangular faces are handled, their corresponding sub-faces, together with non-cut faces, are used to create children tetrahedral elements by calling constrained Delaunay tetrahedralization.

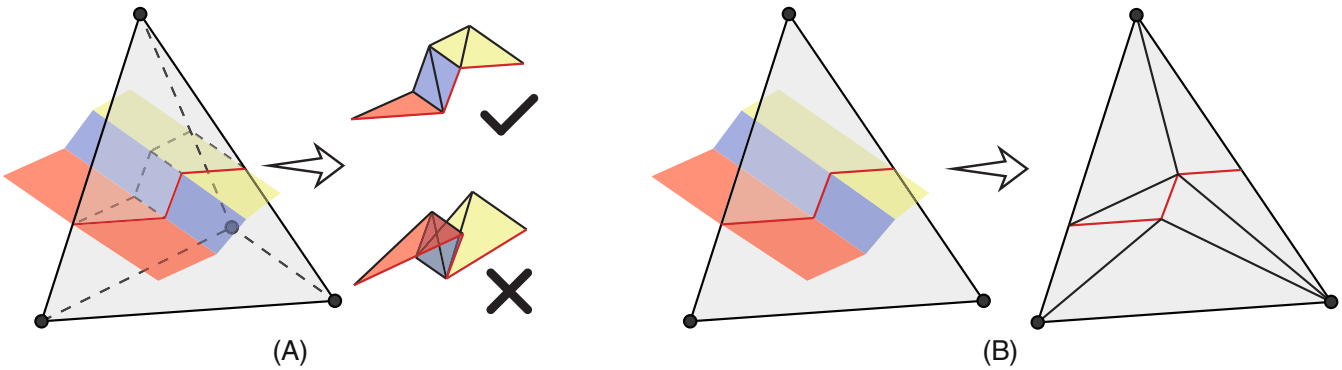

FIGURE 9 An explicit discontinuity composed of three quadrilateral elements intersects a tetrahedral element. (A) Lower-dimensional elements created along the discontinuity are set as internal faces when creating integration elements (top), and wrong internal faces (that do not match the original surface elements) could be created without considering lower-dimensional elements (bottom); (B) When creating the children elements of a triangular face, intersections between the face and this surface mesh (marked in red segments) are considered as constrained segments.

Algorithm 4. Create lower-dimensional elements along the discontinuity and integration elements of a background mesh element

Input: Parent element e_p, ith discontinuity d_i, and enrichment associative container $\mathbf{I}$

function createElements($e_p, d_i, \mathbf{I}$)
  $e_{d_i} \leftarrow$ createDiscontinuityElements(e_p, d_i) – *Create lower-dimensional elements along discontinuity*
  $e_c \leftarrow$ createIntegrationElements($e_p, e_{d_i}, d_i, \mathbf{I}$) – *Create children of the background element*
  return e_{d_i}, e_c
end function

function createIntegrationElements($e_p, e_{d_i}, d_i, \mathbf{I}$)
  $dim \leftarrow$ getDimension(e_p) – *Get dimension of the parent element*
  $\boldsymbol{X}_p \leftarrow$ getNodes(e_p) – *Get original nodes of the parent element*
  $\boldsymbol{x}_p \leftarrow$ getEnrichedNodes($e_p, \mathbf{I}$) – *Get enriched nodes of the parent element*
  if $dim = 1$ **then**
    $l \leftarrow$ createSegments($\boldsymbol{X}_p, \boldsymbol{x}_p$) – *Create segments of the edge (or line element)*
    $e_c \leftarrow$ createElements(l) – *Create children elements*
  if $dim = 2$ **then**
    $edges \leftarrow$ getEdges(e_p) – *Get edges of the parent element*
    for $edge \in edges$ **do**
      $newEdges \leftarrow$ createIntegrationElements($edge, e_{d_i}, d_i, \mathbf{I}$) – *Create children of each edge*
    $t \leftarrow$ Triangle.MeshInfo() – *Initialize Triangle package*
    $t \leftarrow t$.set_points($\boldsymbol{X}_p, \boldsymbol{x}_p$) – *Pass all nodes to Triangle*
    $t \leftarrow t$.set_facets($newEdges$) – *Pass all edges to Triangle*
    $e_c \leftarrow$ Triangle.build(t) – *Create children elements*
  if $dim = 3$ **then**
    $faces \leftarrow$ getFaces(e_p) – *Get faces of the parent element*
    for $face \in faces$ **do**
      $newFaces \leftarrow$ createIntegrationElements($face, e_{d_i}, d_i, \mathbf{I}$) – *Create children of each face*
    $t \leftarrow$ Tetgen.MeshInfo() – *Initialize Tetgen package*
    $t \leftarrow t$.set_points($\boldsymbol{X}_p, \boldsymbol{x}_p$) – *Pass all nodes to Tetgen*
    $t \leftarrow t$.set_facets($newFaces$) – *Pass all faces to Tetgen*
    $e_c \leftarrow$ Tetgen.build(t) – *Create children elements*
  return e_c
end function

Output: Lower-dimensional elements along the discontinuity e_{d_i} and volumetric children elements e_c

2.7 | Mesh modification

The last step of the geometric engine is transforming the background mesh into the enriched discretization. This consists of three main functions as shown in Algorithm 5: `modifyNodes`, `modifyElements`, and `modifyPhysicalGroups`. The purpose of functions `modifyNodes` and `modifyElements` is straightforward, since they update the mesh data structures via adding newly created enriched nodes and integration elements. When dealing with multi-phase problems,

Algorithm 5. Modify the mesh data structure

Input: The original mesh $\mathbf{M} := \{\mathcal{N}, \mathcal{E}, \mathcal{P}\}$ (node, element and physical group sets), discontinuity set $\mathcal{D}$, physical group set $\mathcal{P}_{\text{new}}$, tree data structure $\mathbf{T}$, and enrichment associative container $\mathbf{I}$

```
function modify(D, P_new, I, T)
    N ← modifyNodes(I)                          – Add enriched nodes to the mesh data structure
    E ← modifyElements(T)                       – Add elements and mask the cut background elements
    P ← modifyPhysicalGroups(D, P_new, T)       – Add new physical groups and modify original ones
    return N, E, P
end function
```

Output: Modified mesh $\mathbf{M} := \{\mathcal{N}, \mathcal{E}, \mathcal{P}\}$

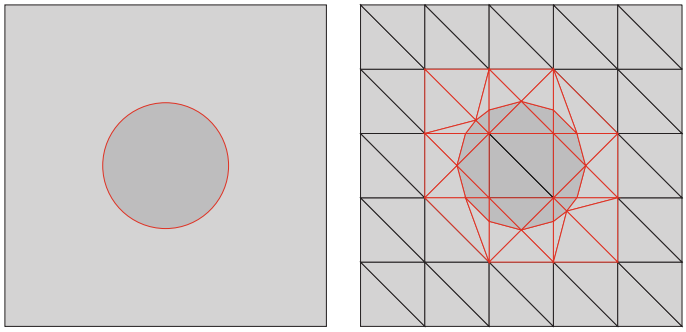

FIGURE 10 (A) A circular inclusion (marked in darker color and red interface) is embedded into a square matrix; (B) A 2-D background mesh with $5 \times 5 \times 2$ triangular elements intersected by the circular inclusion creates integration elements (marked in red triangles) that are assigned to a new physical group, together with 2 uncut elements that lie fully inside the inclusion (marked in darker color).

it is required to specify *physical groups* to different parts of the mesh that correspond to different materials. The function `modifyPhysicalGroups` creates new physical domains associated with discontinuities and changes the original physical domains (see Algorithm 6). Geometric groups, which include elements inside or at either side of a given discontinuity, are a prerequisite for the creation of physical groups. Once geometric groups are defined, Boolean operations are performed on them for creating the physical groups, which could have either the same dimension as that of the background mesh or a lower dimension—for example, lower-dimensional elements created along discontinuities. If there is no need to create new physical groups, for instance, when modeling a fracture problem with a single material phase, all integration and background elements are then assigned to the original physical group of the background mesh.

To explain this last step of the geometric engine, consider in Figure 10A a 2-D square matrix containing a circular inclusion (marked in red) of a different material, which is described by a level set function. The model is discretized by a mesh composed of $5 \times 5 \times 2$ triangular elements. Then, 42 integration elements and 14 lower-dimensional line elements along the material interface are created via cutting 14 background elements (see Figure 10B). For assigning these elements to their corresponding geometric groups, we start with any integration element and determine its group by checking its nodes' level set values. Next, neighboring integration elements are also assigned to their corresponding geometric group based on the same strategy. As only one discontinuity is considered in this example, the inclusion's geometric group coincides with a physical group associated with different material properties. As shown in Figure 10B, 18 integration and 2 original elements are added to this new physical group—and are removed from their original physical group.

In the case of handling models that contain multiple interfaces and/or cracks, the function `selectDiscontinuities` is used to select the discontinuities associated with new physical groups $\mathcal{P}_{\text{new}}$. For instance, when solving a problem with a crack and an inclusion, only the discontinuity representing the material interface is required to create its corresponding physical group. However, when modeling a problem with multiple phases, geometric groups could

Algorithm 6. Create and modify physical groups

Input: Background mesh $\mathbf{M}$, discontinuity set $\mathcal{D}$, physical group set $\mathcal{P}_{\text{new}}$, and tree data structure $\mathbf{T}$

```
function modifyPhysicalGroups(M, D, P_new, T)
    D_new ← selectDiscontinuities(D, P_new)          – Split original discontinuities if needed
    G_group ← propagateGeoGroups(M, D_new, P_new, T)  – Create geometric groups for discontinuities in D_new if needed
    P ← assignPgroups(G_group, P_new, M)              – Create new physical groups and add them to M if needed
    return P
end function

function propagateGeoGroups(D_new, P_new, T, M)
    dims ← list(dimensions(P_new))                    – Get dimensions of the new physical groups
    for d_i ∈ D_new do                                – Loop over discontinuities to create geometric groups
        for dim ∈ dims do
            if dim == M.dim then                      – For volumetric geometric groups
                e_p ← intersectDiscontinuity(d_i, M)  – Find a background element intersecting d_i
                e_c ← getTreeLeaves(e_p, T)           – Get children elements
                e_in ← findElementsInside(d_i, e_c)   – Find all elements inside d_i by going through neighbors of e_c
                G_group ← addToGroup(e_in)
            if dim < M.dim then                       – For lower-dimensional geometric groups
                e_di ← getDiscontinuityElements(d_i)  – Get lower dimensional elements along the discontinuity
                G_group ← addToGroup(e_di)            – Add elements to this geometric physical group
    return G_group
end function

function assignPgroups(G_group, P_new)
    for p_i ∈ P_new do
        p_i ← boolean(G_group)                        – Create new physical groups with boolean operations
        P ← update(p_i)                               – Update physical groups
    return P
end function
```

Output: Updated $\mathcal{P}$

be combined into a single physical group with a single material property. The geometric engine makes this possible by means of Boolean operations performed on geometric groups associated with different discontinuities. For instance, Figure 11 shows two overlapping material interfaces represented implicitly. In order to collect elements shared by them into a single physical group, as shown in Figure 11A, the Boolean operation `Intersection` is used to choose elements from the corresponding geometric groups. Other operations, such as `Subtraction` or `Union`, are also available to create complex physical groups (see Figures 11B,C, respectively).

3 | COMPUTATIONAL COMPLEXITY

The complexity of the geometric engine depends on that of each individual component. It is well known that finding the nearest neighbor in a k-d tree for any given point has time complexity $\mathcal{O}(\log|\mathcal{E}|)$ on average, where $|\mathcal{E}|$ is the number of

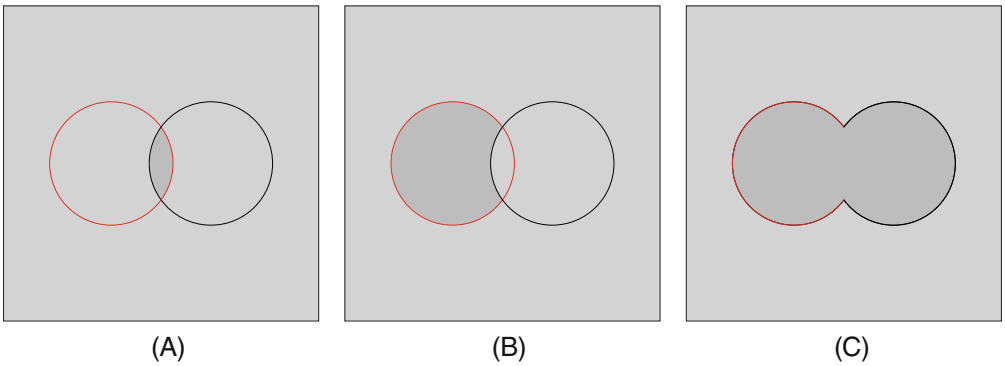

FIGURE 11 Two overlapping circular material interfaces, where different parts are assigned to a new physical group: (A) Intersection between discontinuities; (B) Subtraction of the right discontinuity; (C) Union of discontinuities

elements in the background mesh. `DualGraph` takes constant time (i.e., $\mathcal{O}(1)$) to obtain neighbors of the target element, where a maximum number of neighbors equals to the number of element sides. The time complexity of `Interactor` depends on the number of background elements intersected by discontinuities. Every cut element is tested for intersection against all discontinuities for constructing new children elements; this is done by iterating over all element edges and faces, which asymptotically has constant time complexity $\mathcal{O}(1)$. Considering Delaunay's algorithm for the creation of children elements, the time complexity is $\mathcal{O}(|\mathcal{N}_e| \log |\mathcal{N}_e|)$ in 2-D[116] and $\mathcal{O}(|\mathcal{N}_e|)$ in 3-D,[117] respectively, where $|\mathcal{N}_e|$ is the number of original and enriched nodes in a cut background element. Noteworthy, as $|\mathcal{N}_e|$ is relatively small, asymptotically `ElementCreator` has constant time complexity both in 2-D and 3-D. Since discontinuities have one dimension less than that of the background mesh—that is, they are lower-dimensional manifolds—as problems increase in size, the number of intersected (and integration) elements becomes less significant compared to the total number of background mesh elements. We can therefore estimate the total complexity of the `Interactor` as $\mathcal{O}(|\mathcal{E}|^{1/2})$ and $\mathcal{O}(|\mathcal{E}|^{2/3})$ in two and three dimensions, respectively.

2-D and 3-D models with discontinuities are used to corroborate the time complexity of the geometric engine. For 2-D, a square domain contains 9 discontinuities (represented by 9 line elements) with arbitrary orientations (see Figure 12A). This computational domain is then discretized with increasingly finer meshes with 882, 3362, 13,122, 51,842, 206,082, 821,762, and 3,281,922 constant strain triangles. The cubic 3-D model contains 6 discontinuities described by 12 triangular elements (see Figure 12B), and is discretized by meshes containing 750, 4374, 29,478, 215,622, and 1,647,750 linear tetrahedra. In order to show the efficiency of the geometric engine, we first compare the computational complexity for finding cut elements via a brute-force approach that loops over *all* background elements to the method that uses `SpacePartitioner` to locate elements that contain given points of a discontinuity. The computational time $\varkappa$ has been normalized by the maximum value obtained by both methods. From Figures 13A,B, it can be seen that, asymptotically, the brute-force approach and `SpacePartitioner` associated with the problem size scale as $\mathcal{O}(|\mathcal{E}|)$ and $\mathcal{O}(\log |\mathcal{E}|)$, respectively. Next, the computational time for interacting discontinuities with the background mesh is studied via three different strategies: (i) A brute-force approach that iterates over all background elements; (ii) The brute-force approach to find cut elements but then calls `Dualgraph` to find neighbors of cut elements; (iii) Adopting both `SpacePartitioner` and `Dualgraph`. The normalized computational time $\varkappa$ for 2-D and 3-D examples with respect to the number of background elements $|\mathcal{E}|$ is shown in Figures 14A,B, respectively. For both examples, the first strategy has complexity $\mathcal{O}(|\mathcal{E}|)$ and is therefore the most time-consuming; the second strategy has the same complexity, but takes less time with the help of `Dualgraph`; the last approach is the least expensive one, scaling as estimated above as $\mathcal{O}(|\mathcal{E}|^{1/2})$ in 2-D and $\mathcal{O}(|\mathcal{E}|^{2/3})$ in 3-D.

4 | NUMERICAL EXAMPLES

In this section we demonstrate the capability of the proposed geometric engine with several complex 2-D and 3-D discontinuous problems. We set a tolerance of 10^{-10} for detecting intersections in all examples. Two examples focus on creating a new discretization without conducting analysis, whereas the rest of the examples uses IGFEM and DE-FEM to solve

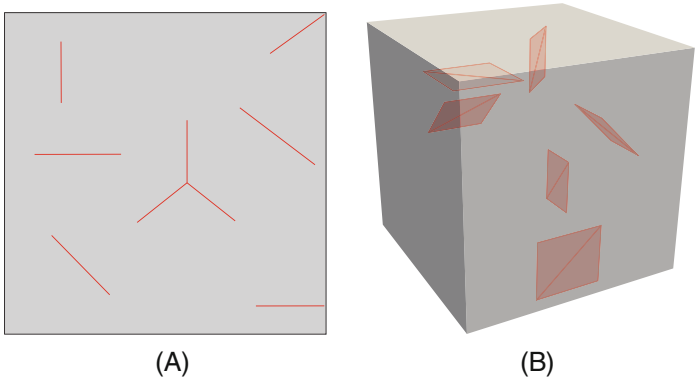

FIGURE 12 (A) 2-D structure including 9 cracks represented by line elements; (B) 3-D model with 6 discontinuities described as triangular elements

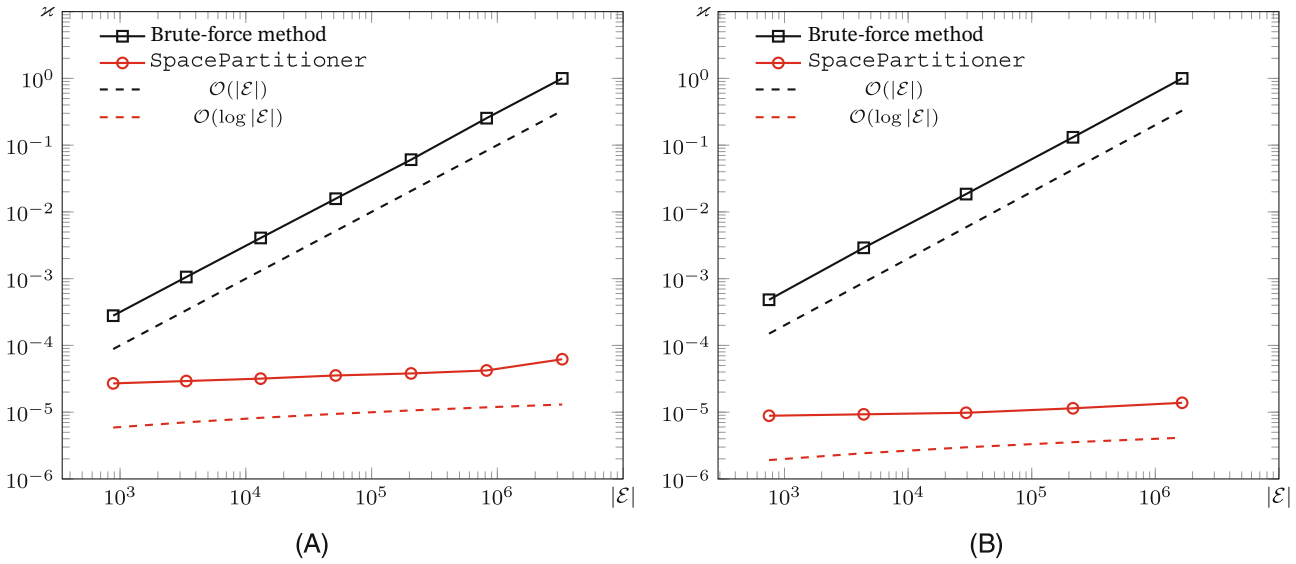

FIGURE 13 Complexity for finding cut elements with given points of discontinuities via brute-force method and `SpacePartitioner` in (A) 2-D and (B) 3-D

weakly and strongly discontinuous problems, respectively. Since no units are given, geometrical parameters, Young's moduli, and magnitudes of applied tractions can be taken from any consistent unit system.

4.1 | Topology optimization

The first example considers evolving material interfaces described by a level set in the context of topology optimization. Figure 15A shows the schematic of a 3-D cantilever beam problem with the corresponding initial hole seed design used; the back surface is clamped and a downward line traction $\bar{\boldsymbol{t}}$ is applied along the middle of the front surface. The cantilever beam is discretized by a structured background mesh with $40 \times 20 \times 10 \times 6$ tetrahedral elements of size a. Young's moduli $E_s = 1$ and $E_v = 10^{-6}$ are assigned for solid and void domains, respectively. Both of them have the same Poisson's ratio of 0.3. Although it is possible to directly use nodal level set values as design variables—thus the level set function is discretized by means of standard finite element shape functions—we use instead compact radial basis functions (RBFs)[114] to parameterize the level set function. This choice, whereby the design space is decoupled from the FE discretization, shows benefits comparable to a filter in density-based topology optimization.[33] The design space with the same dimension as that of the cantilever beam is discretized using a grid of $21 \times 11 \times 6$ RBFs, with a supported radius of $r_s = \sqrt{2} \cdot a$, where

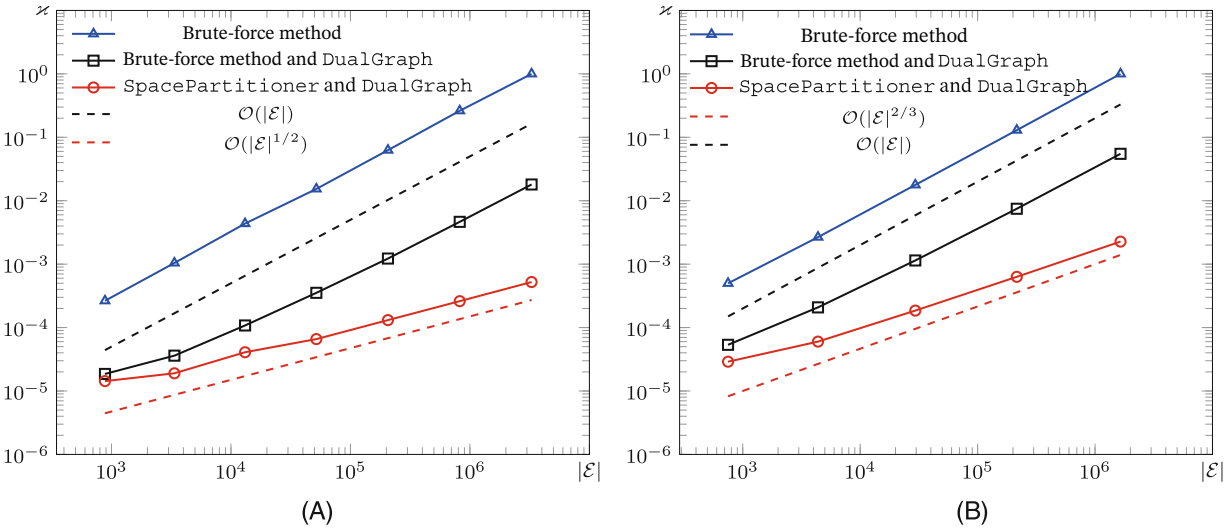

FIGURE 14 Complexity for interacting discontinuities and the background mesh via looping over elements, brute-force method with `DualGraph`, and `SpacePartitioner` with `DualGraph` in (A) 2-D and (B) 3-D

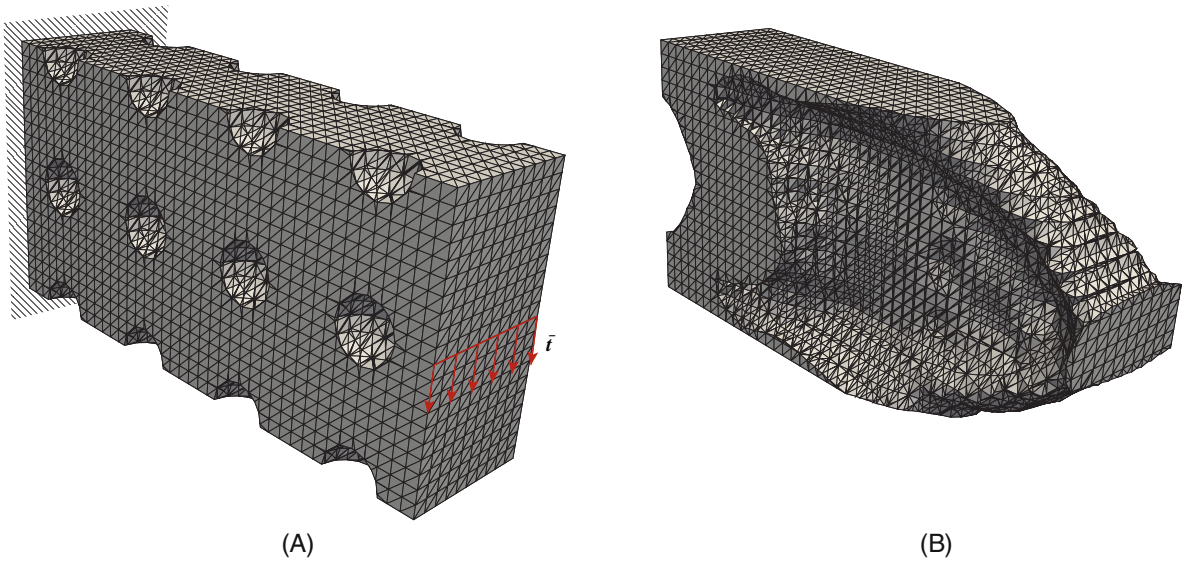

FIGURE 15 3-D Cantilever beam topology optimization problem: (A) Schematic where the back surface is clamped and a downward line traction $\bar{\boldsymbol{t}}$ is applied along the middle of the front surface. The figure also shows the initial hole seed design; (B) Optimized design for minimal structural compliance after 200 iterations

a is the grid size. The cantilever beam is optimized for minimum structural compliance (i.e., maximum stiffness) with solid material constrained at 55% of the total volume of the design domain. Figure 15B shows the optimized design with satisfied volume constraint after 200 iterations. This example shows the geometric engine's ability to handle moving level set—thus implicit—weak discontinuities in 3-D. For more information on level set-based topology optimization using IGFEM we refer to van den Boom *et al.*[33]

4.2 | Intersecting discontinuities

4.2.1 | Fracture junction

In this example DE-FEM is used to resolve multiple intersecting cracks. As shown in Figure 16A, a star-shaped fracture is placed at the center of a finite plate with dimensions 1.25×1.25, where unit magnitude bi-axial tractions $\bar{\boldsymbol{t}}$ are

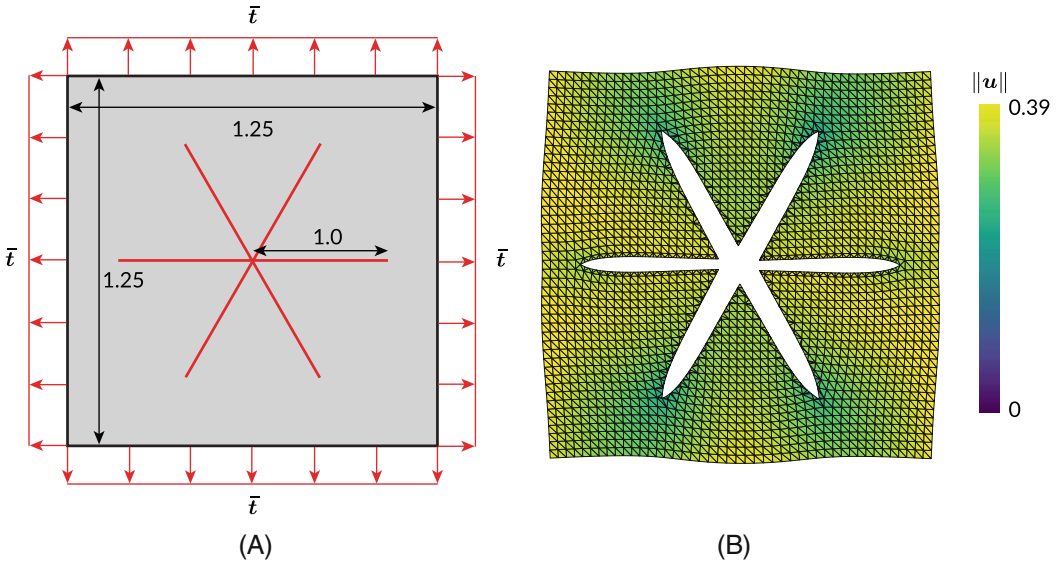

FIGURE 16 (A) Schematic diagrams for star-shaped cracks in a finite plate; (B) The displacement $\boldsymbol{u}$ distribution with the deformed model discretized by the background mesh consisting of $45 \times 45 \times 2$ triangular elements.

prescribed normal to the boundary. Cracks, represented explicitly as line elements of unit length, intersect at the center of the domain. The background mesh used has $45 \times 45 \times 2$ triangular elements. After detecting intersections between cracks and background elements and creating integration elements, a new discretization with 712 integration elements is obtained. For modeling crack junctions using DE-FEM, the number of strong nodes (for resolving the jump in displacement) added in the junction depends on the number of intersecting cracks. In other words, we need to create a displacement jump between each two intersecting cracks, and if n cracks intersect, we need at least $n - 1$ strong enriched nodes, since the last displacement jump can be formed by a linear combination of all other displacement jumps. In our example, since six line segments intersect, five strong nodes are generated. The resulting displacement field is shown in Figure 16B. More details on the use of DE-FEM for resolving multiple intersecting discontinuities can be found in Liu *et al.*[43]

4.2.2 | Resolving both weak and strong discontinuities

Figure 17A shows a 2×2 plate that contains a material interface (marked with blue) and a crack (marked with red). While the former is represented using a level set function, for the latter we use a one-dimensional surface mesh. An elastic material with Young's modulus $E_1 = 1$ and Poisson's ratio $\nu_1 = 0$ is assigned to the domain at the left side of the interface, and $E_2 = 2$ and $\nu_2 = 0$ is assigned to the rest of the plate. While the left boundary is fully constrained, only the vertical displacement is restricted along top and bottom edges. Horizontal tractions $\bar{\boldsymbol{t}}_1 = \boldsymbol{e}_1$ and $\bar{\boldsymbol{t}}_2 = 2\boldsymbol{e}_1$ are imposed on the right edge below and above the crack, respectively. For this example, which aims at illustrating the hierarchical data structure used to store the information of cut elements and their corresponding subdomains, a simple background mesh with $5 \times 5 \times 2$ triangular elements is used. The result obtained via DE-FEM is shown in Figure 17B, where it can be seen that two independent kinematic fields with constant state of stress $\boldsymbol{\sigma}$ are recovered at either side of the crack. The resulting discretization after processing the interaction between both discontinuities with the non-fitted mesh is given in Figure 18, where a representation of the tree data structure shows the three hierarchical levels created to store integration elements. For instance, the background element (hatched in Figure 18) intersects with both the material interface and the crack. The material interface first splits this element (added to the first level of the tree data structure) into three children elements, which are created at the second level of the hierarchy. Later, these elements intersect with the crack, and their corresponding integration elements are stored at the last level (leaves of the tree). More details about this example can be found in van den Boom *et al.*[36]

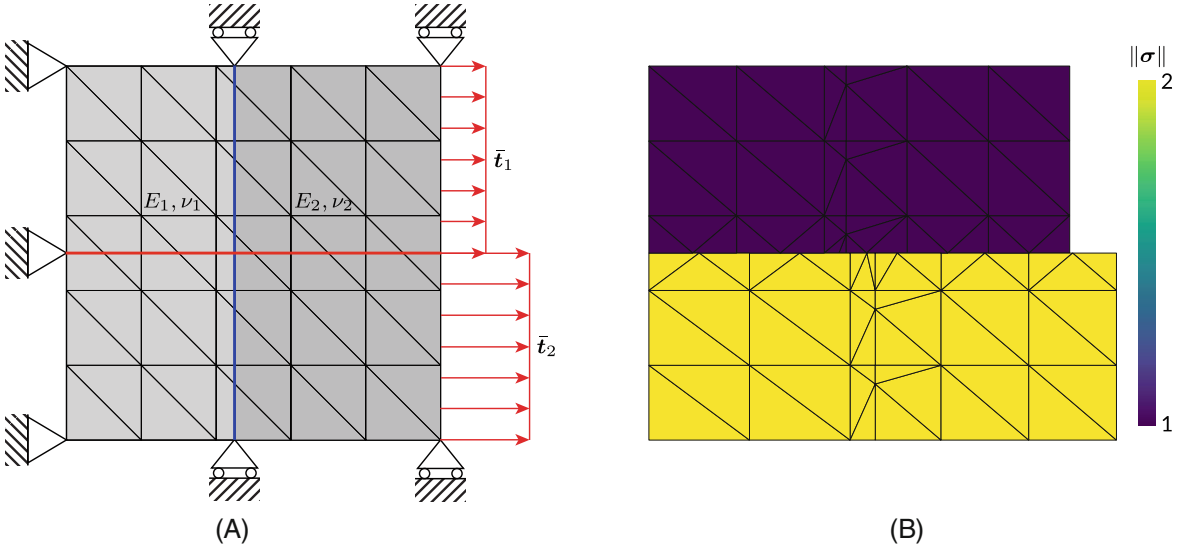

FIGURE 17 (A) A square domain with dimensions 2×2 includes both a material interface (marked with blue) and a crack (marked with red); horizontal tractions $\left\|\bar{t}_1\right\| = 1$ and $\left\|\bar{t}_2\right\| = 2$ are imposed on the right boundary below and above the crack, respectively. (B) Two independent kinematic fields with constant state of stress $\boldsymbol{\sigma}$ are generated in either side of the crack.

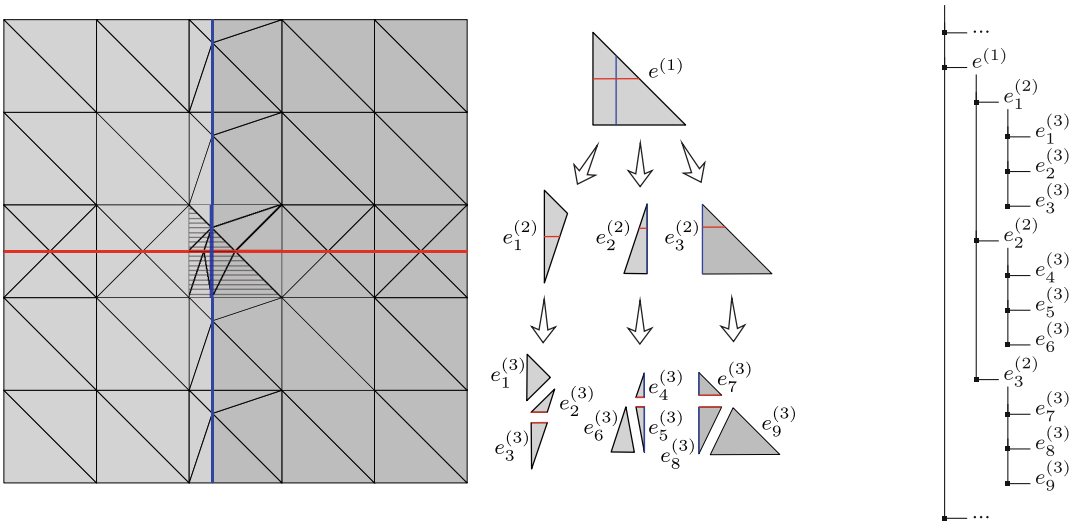

FIGURE 18 The new discretization created after interacting discontinuities with the background mesh, where a cut element (hatched) intersects with both material interface (marked with blue) and crack (marked with red); the corresponding children elements are created and stored into a hierarchical tree data structure with three levels.

4.3 | Polycrystalline materials

In this example, subdividing the computational domain into grains is aggravated by the correct assignment of material properties. Here grains are represented explicitly with polygons in 2-D and polyhedra in 3-D. Polygons are represented by segments forming a closed surface. This is achieved by defining a proper nodal connectivity—similarly to the way we store the node connectivity of finite elements. Analogously, surface polygons are required to describe the corresponding polyhedra.

We showcase the geometric engine's capability to cope with polycrystalline models in 2-D and 3-D. The 2-D model shown in Figure 19A is composed of 30 polygonal grains with different material properties (every color corresponds to

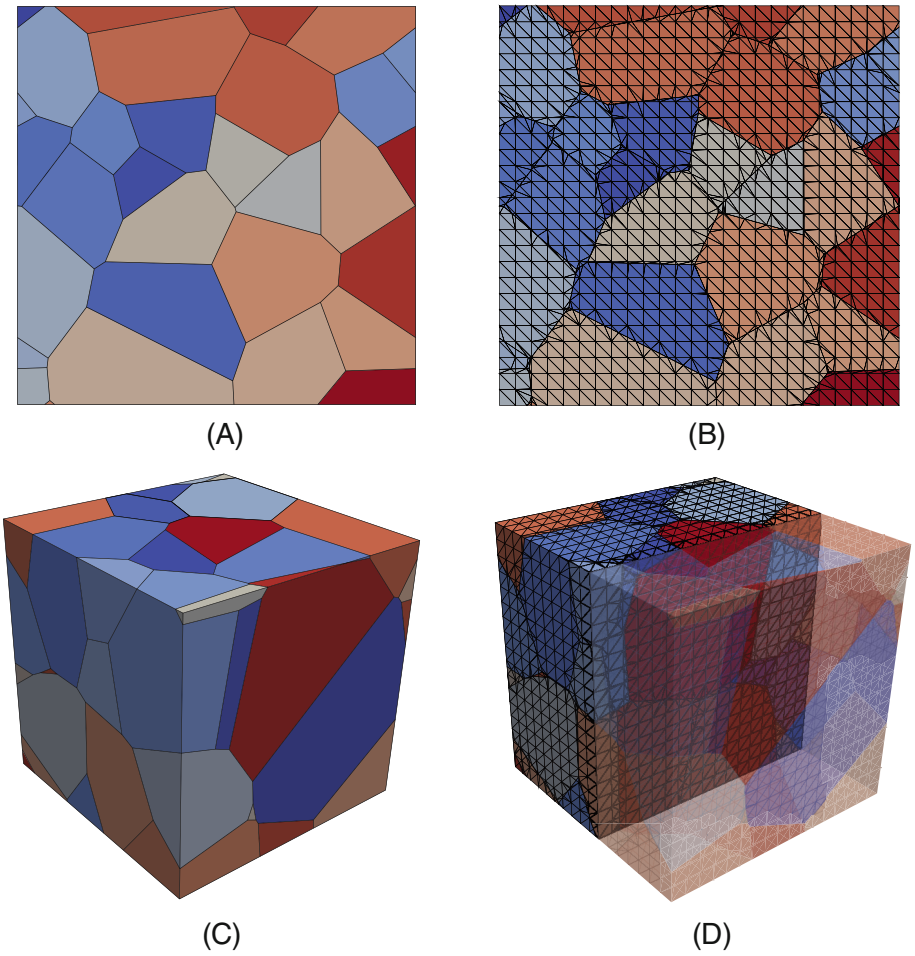

FIGURE 19 Polycrystalline models (A,C) and their finite element discretizations after interacting with the geometric engine (B,D). The figure shows that the geometric engine is able to properly construct integration elements and that the right material properties are assigned (color-coded regions).

a different material), and a structured mesh of $25 \times 25 \times 2$ triangular elements is used. As apparent from Figure 19B, the geometric engine is able to correctly assign the appropriate material properties to each grain. Obtaining the same results in 3-D is more challenging. Figure 19C shows 157 polyhedra used to describe the 3-D polycrystalline model, on a background mesh composed of $17 \times 17 \times 17 \times 6$ tetrahedral elements. The result of interacting the polycrystalline model with the background mesh in Figure 19D shows the proper creation of integration elements and assignment of material properties.

4.4 | Immersing lower-dimensional manifolds

Here we demonstrate the geometric engine's ability to immerse a 2-D surface mesh (composed of triangular elements) into a 3-D tetrahedral mesh. To this end we use the well-known Stanford bunny in the computer graphics literature,[118] which is discretized by a coarse surface mesh containing 1,006 triangular elements (see Figure 20A). This surface mesh is then fully immersed into a structured mesh composed of $20 \times 20 \times 20 \times 6$ tetrahedral elements, as shown in Figure 20B. A total of 3,768 background elements are cut by the surface mesh (see Figure 20C), resulting in 57,172 newly created integration elements. The resulting new volumetric discretization of the Stanford bunny is composed of 2,844 background and 28,066 integration elements, and the original surface mesh is split into 20,319 triangular elements (see Figure 20D).

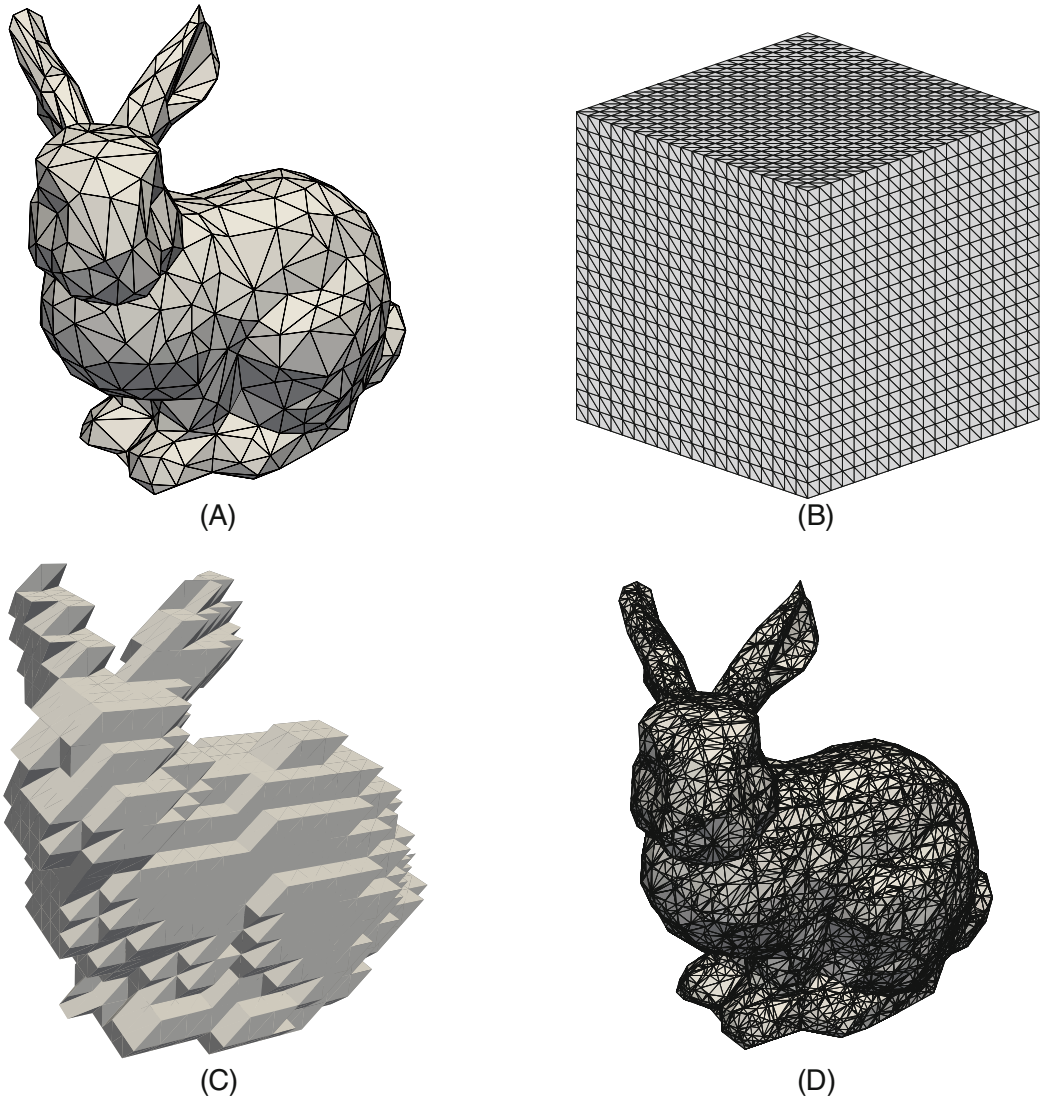

FIGURE 20 (A) The Stanford bunny surface mesh discretized by 1006 triangular elements; (B) The background mesh consisting of $20 \times 20 \times 20 \times 6$ tetrahedral elements; (C) 3768 background tetrahedral elements intersected by the surface mesh; and (D) The new volumetric discretization of the Stanford bunny model with 2844 background and 28,066 integration elements

5 | EXTENSIONS FOR OTHER UNFITTED/IMMERSED/ENRICHED FEMS

The proposed geometric engine could be readily applied to unfitted FEMs. Similarly to IGFEM, unfitted FEMs that are used to solve discontinuous problems need to detect subdomains at either side of material interfaces. A tessellation of cut elements is also required to create integration elements that are used for calculating local arrays. These steps are currently considered in the proposed geometric engine: cut elements, together with their corresponding integration elements (and also lower-dimensional elements along discontinuities), are stored in the ordered tree data structure. Finite elements that are not cut by interfaces can be found in either original volumetric and newly created physical groups. Therefore, the proposed geometric engine can be used as is for unfitted FEMs.

As with IGFEM, immersed FEMs separate the background mesh into two classes: non-cut and cut elements. The former uses the standard shape functions for the assembly. For cut elements, the original basis functions are modified to piecewise functions for capturing discontinuous behaviors in cut elements, which requires detecting intersections between discontinuities and elements. Since no integration elements are created in this method, only the information obtained from `Intersector` (stored in container **I**) is needed. Therefore, the geometric engine can also be used as is for immersed FEMs.

Extending the proposed geometric engine so that it can be used with X/GFEM is straightforward. Similarly for X/GFEM, intersections between the background mesh elements and discontinuities need to be detected and integration elements need to be created—to ensure the integration of smooth functions. However, contrary to IGFEM and DE-FEM, where new enriched nodes are created along discontinuities, in X/GFEM newly created integration elements are also connected to *"dummy nodes"* that have no associated degrees of freedom; this is because in X/GFEM enriched nodes are added to nodes of the original finite element mesh. Dummy nodes can nevertheless be added to the original nodes array as long as we use another mechanism that performs the bookkeeping of enriched DOFs. For instance, an associative container could be used to store information of enriched nodes associated to a given standard mesh node—the latter's id becoming the key in the data structure with the corresponding value being the associated enriched DOFs. This container would then be used during the computation of local elemental arrays, and for obtaining element freedom tables that are used to map DOFs from local to global matrices and vectors. It should also be mentioned that, when using X/GFEM for solving fracture mechanics problems, appropriate enrichments are commonly used to capture singularities near crack fronts. Both topological and geometrical enrichment strategies have been used in this regard;[119] while the former only enriches nodes of elements cut by the crack front, the latter enriches a region contiguous to the crack front that is independent of mesh size—which remains roughly the same as the finite element mesh is refined and therefore yields a more accurate solution. The geometric engine proposed in this article could easily be extended to work with either type of enrichment strategy. For a topological enrichment strategy, the detection of elements that contain the crack front was outlined in Section 2.5. This could be combined with a search for neighboring elements by means of the dual graph, following Section 2.7, for finding all elements within the geometrical enrichment region. An X/GFEM implementation could also benefit from the proposed geometric engine by means of the explicit representation of crack fronts, for example, by means of a mesh, so that crack propagation simply entails advancing the front by simply adding new elements. Notice also that an explicit representation of the crack front makes it straightforward to place a local cylindrical coordinate system, which is required for the definition of the singular enrichments and for computing stress intensity factors.

6 | SUMMARY AND CONCLUSION

In this work we proposed an object-oriented geometric engine especially designed for unfitted/immersed/enriched FEMs, which aims at interacting a background mesh with discontinuities for generating a new discretization. Considering the representation of discontinuities, both implicit and explicit methods are implemented to describe their corresponding geometric configurations. The geometric engine adheres to the requirements of generality, robustness, and efficiency. The geometric engine design contains several main modules: `SpacePartitioner` creates a k -d tree data structure for partitioning the background mesh and is used to identify elements enclosing a given point. `DualGraph` builds a graph structure for finding neighbors of any given element. These two components are called in other modules for facilitating geometric operations. `Interactor` focuses on creating enriched nodes at intersections between discontinuities and the background mesh via calling `Intersector`. Integration elements are created within `ElementCreator` and stored in a hierarchical data structure used for the assembly. Later, enriched nodes and integration elements are added to the original mesh data in `Mesh`, where new physical groups associated with discontinuities are created, for instance, when solving multi-phase problems. Together with detailed explanations, the corresponding pseudo-code of each module was also provided. We investigated the computational complexity of the geometric engine through several 2-D and 3-D models including weak and strong discontinuities, where the behavior of the computational time associated with the number of background elements $|\mathcal{E}|$ is asymptotically approaching $\mathcal{O}(|\mathcal{E}|^{1/2})$ (in 2-D) and $\mathcal{O}(|\mathcal{E}|^{2/3})$ (in 3-D).

Furthermore, several challenging numerical examples were used to investigate the capability of the geometric engine with IGFEM and DE-FEM. First, a 3-D topology optimization example for minimizing structural compliance was used, where a level set function describes the structural boundary and the structural analysis is performed via IGFEM. Next, a fracture junction problem was solved by means of DE-FEM, where several cracks intersect with each other within a single background element. Later, an example including intersected weak and strong discontinuities was proposed to show the hierarchical data structure that stores cut background elements and integration elements. Moreover, we incorporated polycrystalline materials that are represented as polygons in 2-D or polyhedra in 3-D into a background mesh, where new physical groups associated with the corresponding grains are created for assigning different material properties. Finally, the Stanford bunny model with a complex geometry configuration, which is discretized as a surface mesh with 1,006 triangular elements, was immersed into a 3-D background mesh, where the new volumetric discretization conforming to the surface of the bunny was generated.

A summary of the main findings of this work is listed in the following:

- Delaunay triangulation (tetrahedralization) behaves well for creating the corresponding integration elements in 2-D and 3-D. However, if only considering the background mesh with triangular or tetrahedral elements, it is also practical to set specific routines to construct integration elements. For instance, there are only a few scenarios when a triangle is split with a line segment completely. Although more cases should be taken into account for splitting a tetrahedron, it is still doable to implement the corresponding scheme to perform tetrahedralization. Although this alternative could increase the difficulty of the implementation and could require more efforts in considering all possible cut situations, the resulting geometric engine would not require any extra package based on Delaunay algorithm.
- Discontinuities can be handled in either hierarchical or unified way within the geometric engine. For instance, when only one surface mesh is used to represent several cracks, where each surface element is considered as a crack, these cracks can be processed as a single discontinuity. However, these cracks can also be seen as a collection of surface meshes, each containing only one element, where each crack (surface mesh) is one discontinuity that is processed in a hierarchical way. Generally, the latter would create more integration/children elements, as children elements associated with previous discontinuities would be split by the remaining cracks. Therefore, in terms of efficiency, the former method is recommended when handling discontinuous models with multiple cracks.
- As explicit representations can be used to describe discontinuities, computational geometry operations among geometric entities are conducted to detect intersections and create integration elements. For instance, intersections between background and surface elements are detected via segment/segment and segment/triangle tests based on coordinates in `Intersector`. Moreover, it is important to make sure that no duplicated enriched nodes are created at the same location by checking their coordinates. It is thus critical to set an appropriate tolerance value for intersection tests to ensure the robustness of the proposed procedure. The value of this tolerance should be related to the background/surface mesh size. For instance, if the average length of intersecting primitives is 0.1, the difference between overlapping intersections could be chosen (arbitrarily) to be within 10^{-8}. This value of tolerance should be reduced if the mesh size is reduced for the geometric engine to yield the same intersections.
- Considering efficiency, when handling any cut background element, it is important to store the information about sub-domains of cut element edges and/or faces, which can save the computational cost and avoid the precision issue related to check the location of intersections. On the one hand, this data can be directly used to create children of any background elements sharing these edges/faces. On the other hand, it is useful for handing discontinuities in a hierarchical way, as any sub-edges and/or sub-faces created by the handled discontinuities could be split by the following ones.

Although the proposed geometric engine is general, robust, and efficient when dealing with discontinuities, there is still room for improvement:

- Parallelization could make the geometric engine more efficient. A possible parallel implementation could consider the following aspects: (i) A domain decomposition technique could be used to subdivide the background mesh in several subdomains of contiguous elements; information about subdomain boundaries would need to be stored as well to detect duplicate nodes (which overlap in space but that are otherwise shared by different processes); (ii) The geometric engine in this manuscript could then be applied sequentially to each process independently to detect intersections between discontinuities and background elements in each subdomain. Integration elements and enriched nodes would then be created independently; and (iii) Instead of communicating to ensure that duplicate enriched nodes have a unique id, we could use multi-point constraints (MPCs) or another form of constraint enforcement to deal with duplicate nodes. This would necessitate communicating enriched nodes that intersect with subdomain boundaries and checking for coordinates within tolerance.
- Although evolving interfaces described implicitly can readily be handled, the evolution of explicit discontinuities is still missing in the current geometric engine. For instance, crack propagation could be implemented by representing crack fronts explicitly. It would be straightforward to extend the geometric engine to handle crack propagation in 2-D, whereby new crack segments are added to the lower-dimensional mesh, connecting the old tips with the new ones. Crack propagation in 3-D would be more involved, as discussed elsewhere.[120] Extending the proposed geometric engine for 3-D crack propagation should therefore properly consider the geometry of the front after advancing it: New vertices along the crack front should be checked for overlap to ensure no duplicates are present. These nodes should be properly ordered (for instance, after projecting them to a plane) for the creation of new surface elements and to ensure

that new segments created along the crack front do not intersect with one another and have the proper orientation. Finally, depending on the problem, the crack front could intersect with itself, so intersection tests should envision such cases. Newly created nodes along the crack front, which would be added to the nodes array of the surface mesh, would then be used together with nodes of the original crack front to create new surface elements (for instance by means of constrained Delaunay's algorithm); these newly created surface elements would then be added to the updated surface mesh.

ACKNOWLEDGMENT

Jian Zhang would like to thank China Scholarship Council (CSC NO. 201606060130) for the financial support.

DATA AVAILABILITY STATEMENT

All data is contained in the article and the references cited therein.

ORCID

Jian Zhang https://orcid.org/0000-0002-8872-7348
Sanne J. van den Boom https://orcid.org/0000-0002-5547-2596
Dongyu Liu https://orcid.org/0000-0001-9803-1865
Alejandro M. Aragón https://orcid.org/0000-0003-2275-6207

REFERENCES

1. Cottrell JA, Hughes TJ, Bazilevs Y. *Isogeometric Analysis: Toward Integration of CAD and FEA*. 1st ed. John Wiley & Sons, Ltd; 2009.
2. Du Q, Wang D. Recent progress in robust and quality Delaunay mesh generation. *J Comput Appl Math*. 2006;195(1-2):8-23.
3. Swenson DV, Ingraffea AR. Modeling mixed-mode dynamic crack propagation using finite elements: theory and applications. *Comput Mech*. 1988;3(6):381-397.
4. Bouchard PO, Bay F, Chastel Y, Tovena I. Crack propagation modelling using an advanced remeshing technique. *Comput Methods Appl Mech Eng*. 2000;189(3):723-742.
5. Bouchard P-O, Bay F, Chastel Y. Numerical modelling of crack propagation: automatic remeshing and comparison of different criteria. *Comput Methods Appl Mech Eng*. 2003;192(35-36):3887-3908.
6. Bugeda G, Oliver J. A general methodology for structural shape optimization problems using automatic adaptive remeshing. *Int J Numer Methods Eng*. 1993;36(18):3161-3185.
7. Liu Z, Korvink JG. Adaptive moving mesh level set method for structure topology optimization. *Eng Optim*. 2008;40(6):529-558.
8. Salazar de Troya MA, Tortorelli DA. Three-dimensional adaptive mesh refinement in stress-constrained topology optimization. *Struct Multidiscip Optim*. 2020;62(5):2467-2479.
9. Saksono PH, Dettmer WG, Perić D. An adaptive remeshing strategy for flows with moving boundaries and fluid–Structure interaction. *Int J Numer Methods Eng*. 2007;71(9):1009-1050.
10. Masud A, Bhanabhagvanwala M, Khurram RA. An adaptive mesh rezoning scheme for moving boundary flows and fluid–Structure interaction. *Comput Fluids*. 2007;36(1):77-91.
11. Rohit V, Devendra D, Bhaskar P. The study of elastic modulus and thermal conductivity of different fiber cross-section of fiber-reinforced composite in ANSYS. Proceedings of the IOP Conference Series: Materials Science and Engineering; Vol. 1013; 2021:012015; IOP Publishing.
12. Riemselagh K, Vierendeels J, Dick E. Two-dimensional incompressible Navier-Stokes calculations in complex-shaped moving domains. *J Eng Math*. 1998;34(1-2):57-73.
13. Wu L, Bogy DB. Unstructured triangular mesh generation techniques and a finite volume numerical scheme for slider air bearing simulation with complex shaped rails. *IEEE Trans Mag*. 1999;35(5):2421-2423.
14. Li Z, Lin T, Lin Y, Rogers RC. An immersed finite element space and its approximation capability. *Numer Methods Part Differ Equ*. 2004;20(3):338-367.
15. Kafafy R, Lin T, Lin Y, Wang J. Three-dimensional immersed finite element methods for electric field simulation in composite materials. *Int J Numer Methods Eng*. 2005;64(7):940-972.
16. Barrett JW, Elliott CM. Fitted and unfitted finite-element methods for elliptic equations with smooth interfaces. *IMA J Numer Anal*. 1987;7(3):283-300.
17. Noble DR, Newren EP, Lechman JB. A conformal decomposition finite element method for modeling stationary fluid interface problems. *Int J Numer Methods Fluids*. 2010;63(6):725-742.
18. Kramer RM, Noble DR. A conformal decomposition finite element method for arbitrary discontinuities on moving interfaces. *Int J Numer Methods Eng*. 2014;100(2):87-110.
19. Fries TP. Higher-order conformal decomposition FEM (CDFEM). *Comput Methods Appl Mech Eng*. 2018;328:75-98.

20. Burman E, Claus S, Hansbo P, Larson MG, Massing A. CutFEM: discretizing geometry and partial differential equations. *Int J Numer Methods Eng*. 2015;104(7):472-501.
21. Moës N, Dolbow J, Belytschko T. A finite element method for crack growth without remeshing. *Int J Numer Methods Eng*. 1999;46(1):131-150.
22. Duarte CA, Hamzeh ON, Liszka TJ, Tworzydlo WW. A generalized finite element method for the simulation of three-dimensional dynamic crack propagation. *Comput Methods Appl Mech Eng*. 2001;190(15-17):2227-2262.
23. Melenk JM, Babuška I. The partition of unity finite element method: basic theory and applications. *Comput Methods Appl Mech Eng*. 1996;139:289-314.
24. Babuška I, Melenk JM. The partition of unity method. *Int J Numer Methods Eng*. 1997;40:727-758.
25. Chessa J, Wang H, Belytschko T. On the construction of blending elements for local partition of unity enriched finite elements. *Int J Numer Methods Eng*. 2003;57(7):1015-1038.
26. Fries TP. A corrected XFEM approximation without problems in blending elements. *Int J Numer Methods Eng*. 2008;75(5):503-532.
27. Babuška I, Banerjee U, Osborn JE. Survey of meshless and generalized finite element methods: a unified approach. *Acta Numer*. 2003;12:1-125.
28. Moës N, Béchet E, Tourbier M. Imposing Dirichlet boundary conditions in the extended finite element method. *Int J Numer Methods Eng*. 2006;67(12):1641-1669.
29. Babuška I, Banerjee U. Stable generalized finite element method (SGFEM). *Comput Methods Appl Mech Eng*. 2012;201–204:91-111.
30. Gupta V, Duarte CA, Babuška I, Banerjee U. A stable and optimally convergent generalized FEM (SGFEM) for linear elastic fracture mechanics. *Comput Methods Appl Mech Eng*. 2013;266:23-39.
31. Kergrene K, Babuška I, Banerjee U. Stable generalized finite element method and associated iterative schemes; application to interface problems. *Comput Methods Appl Mech Eng*. 2016;305:1-36.
32. Soghrati S, Aragón AM, Armando DC, Geubelle PH. An interface-enriched generalized FEM for problems with discontinuous gradient fields. *Int J Numer Methods Eng*. 2012;89:991-1008.
33. van den Boom SJ, Zhang J, van Keulen F, Aragón AM. An interface-enriched generalized finite element method for level set-based topology optimization. *Struct Multidiscip Optim*. 2020;63(1):1-20.
34. Soghrati S. Hierarchical interface-enriched finite element method: an automated technique for mesh-independent simulations. *J Comput Phys*. 2014;275:41-52.
35. Aragón AM, Liang B, Ahmadian H, Soghrati S. On the stability and interpolating properties of the hierarchical Interface-enriched finite element method. *Comput Methods Appl Mech Eng*. 2020;362:112671.
36. van den Boom SJ, Zhang J, van Keulen F, Aragón AM. A stable interface-enriched formulation for immersed domains with strong enforcement of essential boundary conditions. *Int J Numer Methods Eng*. 2019;120(10):1163-1183.
37. Cuba-Ramos A, Aragón AM, Soghrati S, Geubelle PH, Molinari J. A new formulation for imposing Dirichlet boundary conditions on non-matching meshes. *Int J Numer Methods Eng*. 2015;103(6):430-444.
38. van den Boom SJ, van Keulen F, Aragón AM. Fully decoupling geometry from discretization in the Bloch–Floquet finite element analysis of phononic crystals. *Comput Methods Appl Mech Eng*. 2021;382:113848.
39. Liu D, van den Boom SJ, Simone A, Aragón A. An interface-enriched generalized finite element formulation for locking-free coupling of non-conforming discretizations and contact; 2022.
40. Zhang J, van Keulen F, Aragón AM. On tailoring fracture resistance of brittle structures: a level set interface-enriched topology optimization approach. *Comput Methods Appl Mech Eng*. 2022;388:114189.
41. Aragón AM, Simone A. The discontinuity-enriched finite element method. *Int J Numer Methods Eng*. 2017;112(11):1589-1613.
42. Zhang J, Boom SJ, Keulen F, Aragón AM. A stable discontinuity-enriched finite element method for 3-D problems containing weak and strong discontinuities. *Comput Methods Appl Mech Eng*. 2019;355:1097-1123.
43. Liu D, Zhang J, Aragón AM, Simone A. The discontinuity-enriched finite element method for multiple intersecting discontinuities. *Int J Numer Methods Eng*.
44. Yan Y. *A Discontinuity-Enriched Finite Element Method for Dynamic Fracture in Brittle Materials*. Master's thesis; 2021.
45. De Lazzari E, Boom SJ, Zhang J, Keulen F, Aragón AM. A critical view on the use of non-uniform rational B-splines to improve geometry representation in enriched finite element methods. *Int J Numer Methods Eng*. 2021;122(5):1195-1216.
46. Sabet FA, Raeisi NA, Hamed E, Jasiuk I. Modelling of bone fracture and strength at different length scales: a review. *Interface Focus*. 2016;6(1):20150055.
47. Erden S, Ho K. Fiber reinforced composites. In: Seydibeyoglu MO, Mohanty AK, Misra M, eds. *Fiber Technology for Fiber-Reinforced Composites, Woodhead Publishing Series in Composites Science and Engineering*. Vol 3. Woodhead Publishing; 2017:51-79.
48. Sukumar N, Chopp DL, Moës N, Belytschko T. Modeling holes and inclusions by level sets in the extended finite-element method. *Comput Methods Appl Mech Eng*. 2001;190(46-47):6183-6200.
49. Tranquart B, Ladevèze P, Baranger E, Mouret A. A computational approach for handling complex composite microstructures. *Compos Struct*. 2012;94(6):2097-2109.
50. Soghrati S, Geubelle PH. A 3D interface-enriched generalized finite element method for weakly discontinuous problems with complex internal geometries. *Comput Methods Appl Mech Eng*. 2012;217:46-57.
51. Simone A, Duarte CA, Giessen E. A generalized finite element method for polycrystals with discontinuous grain boundaries. *Int J Numer Methods Eng*. 2006;67(8):1122-1145.
52. Sietsma J. Nucleation and growth during the austenite-to-ferrite phase transformation in steels after plastic deformation. In: Pereloma E, Edmonds DV, eds. *Phase Transformations in Steels*. Elsevier; 2012:505-526.

53. Shen J. *Advanced Ceramics for Dentistry*. Butterworth-Heinemann; 2013.
54. Yi D, Wang Y, Erve OMJ, *et al*. Emergent electric field control of phase transformation in oxide superlattices. *Nat Commun*. 2020;11(1):1-8.
55. Ruan Y, Liu JC, Richmond O. A deforming finite element method for analysis of alloy solidification problems. *Finite Elem Anal Des*. 1993;13(1):49-63.
56. Ji H, Chopp D, Dolbow JE. A hybrid extended finite element/level set method for modeling phase transformations. *Int J Numer Methods Eng*. 2002;54(8):1209-1233.
57. Merle R, Dolbow J. Solving thermal and phase change problems with the extended finite element method. *Comput Mech*. 2002;28(5):339-350.
58. Cosimo A, Fachinotti Víctor, Cardona A. An enrichment scheme for solidification problems. *Comput Mech*. 2013;52(1):17-35.
59. Stapór P. The XFEM for nonlinear thermal and phase change problems. *Int J Numer Methods Heat Fluid Flow*. 2015;25(2):400-421.
60. Li M, Chaouki H, Robert J-L, Ziegler D, Martin D, Fafard M. Numerical simulation of Stefan problem with ensuing melt flow through XFEM/level set method. *Finite Elem Anal Des*. 2018;148:13-26.
61. Faghri A, Zhang Y. In: Faghri A, Zhang Y, eds. *Transport phenomena in multiphase systems*. Elsevier; 2006.
62. Pan L, Webb SW, Oldenburg CM. Analytical solution for two-phase flow in a wellbore using the drift-flux model. *Adv Water Resour*. 2011;34(12):1656-1665.
63. Wallis GB. *One-Dimensional Two-Phase Flow*. Dover Publications; 2020.
64. Massoud M. *Engineering Thermofluids. Thermodynamics, Fluid Mechanics, and Heat Transfer*. Springer; 2007.
65. Chessa J, Belytschko T. An enriched finite element method and level sets for axisymmetric two-phase flow with surface tension. *Int J Numer Methods Eng*. 2003;58(13):2041-2064.
66. Chessa J, Belytschko T. An extended finite element method for two-phase fluids. *J Appl Mech*. 2003;70(1):10-17.
67. Sauerland H, Fries T-P. The stable XFEM for two-phase flows. *Comput Fluids*. 2013;87:41-49.
68. Zhuang Z, Liu Z, Cheng B, Liao J. *Extended Finite Element Method: Tsinghua University Press Computational Mechanics Series*. Academic Press; 2014.
69. Zienkiewicz OC, Taylor RL, Nithiarasu P. Ch. 13 - Fluid–Structure interaction. In: Zienkiewicz OC, Taylor RL, Nithiarasu P, eds. *The Finite Element Method for Fluid Dynamics*. 7th ed. Butterworth-Heinemann; 2014:423-449.
70. Takashi N. ALE finite element computations of fluid-structure interaction problems. *Comput Methods Appl Mech Eng*. 1994;112(1-4):291-308.
71. Kuhl E, Hulshoff S, De Borst R. An arbitrary Lagrangian Eulerian finite-element approach for fluid–Structure interaction phenomena. *Int J Numer Methods Eng*. 2003;57(1):117-142.
72. Rubenstein D, Yin W, Frame MD, eds. *Biofluid Mechanics: An Introduction To Fluid Mechanics, Macrocirculation and Microcirculation*. Academic Press; 2015.
73. Gerstenberger A, Wall WA. An extended finite element method/Lagrange multiplier based approach for fluid–Structure interaction. *Comput Methods Appl Mech Eng*. 2008;197(19-20):1699-1714.
74. Gerstenberger A, Wall WA. Enhancement of fixed-grid methods towards complex fluid–Structure interaction applications. *Int J Numer Methods Fluids*. 2008;57(9):1227-1248.
75. Jenkins N, Maute K. Level set topology optimization of stationary fluid-structure interaction problems. *Struct Multidiscip Optim*. 2015;52(1):179-195.
76. Cantwell WJ, Morton J. The significance of damage and defects and their detection in composite materials: a review. *J Strain Anal Eng Des*. 1992;27(1):29-42.
77. Naebe M, Abolhasani MM, Khayyam H, Amini A, Fox B. Crack damage in polymers and composites: a review. *Polym Rev*. 2016;56(1):31-69.
78. Sukumar N, Moës N, Moran B, Belytschko T. Extended finite element method for three-dimensional crack modelling. *Int J Numer Methods Eng*. 2000;48(11):1549-1570.
79. Moës N, Belytschko T. Extended finite element method for cohesive crack growth. *Eng Fract Mech*. 2002;69(7):813-833.
80. Cox JV. An extended finite element method with analytical enrichment for cohesive crack modeling. *Int J Numer Methods Eng*. 2009;78(1):48-83.
81. Hull D, Bacon DJ. *Introduction to Dislocations*. Butterworth-Heinemann; 2011.
82. Lagerlof P. Crystal Dislocations: Their Impact on Physical Properties of Crystals. *Crystals*. 2018;8(11):413.
83. Ventura G, Moran B, Belytschko T. Dislocations by partition of unity. *Int J Numer Methods Eng*. 2005;62(11):1463-1487.
84. Gracie R, Ventura G, Belytschko T. A new fast finite element method for dislocations based on interior discontinuities. *Int J Numer Methods Eng*. 2007;69(2):423-441.
85. Robbins J, Voth TE. Modeling dislocations in a polycrystal using the generalized finite element method. *Int J Theor Appl Multisc Mech*. 2011;2(2):95-110.
86. Shuttle DA, Smith IM. Numerical simulation of shear band formation in soils. *Int J Numer Anal Methods Geomech*. 1988;12(6):611-626.
87. Rudnicki JW, Rice JR. Conditions for the localization of deformation in pressure-sensitive dilatant materials. *J Mech Phys Solids*. 1975;23(6):371-394.
88. Meyers MA, Nesterenko VF, LaSalvia JC, Xue Q. Shear localization in dynamic deformation of materials: microstructural evolution and self-organization. *Mater Sci Eng A*. 2001;317(1-2):204-225.
89. Humphreys FJ, Hatherly M. *Recrystallization and Related Annealing Phenomena*. Elsevier; 2012.
90. Gilman JJ. Micromechanics of shear banding. *Mech Mater*. 1994;17(2-3):83-96.

91. Desrues J, Lanier J, Stutz P. Localization of the deformation in tests on sand sample. *Eng Fract Mech*. 1985;21(4):909-921.
92. Samaniego E, Belytschko T. Continuum–Discontinuum modelling of shear bands. *Int J Numer Methods Eng*. 2005;62(13):1857-1872.
93. Mikaeili E, Liu P. Numerical modeling of shear band propagation in porous plastic dilatant materials by XFEM. *Theor Appl Fract Mech*. 2018;95:164-176.
94. Areias PMA, Belytschko T. Two-scale shear band evolution by local partition of unity. *Int J Numer Methods Eng*. 2006;66(5):878-910.
95. Areias PMA, Belytschko T. Two-scale method for shear bands: thermal effects and variable bandwidth. *Int J Numer Methods Eng*. 2007;72(6):658-696.
96. Fabri A, Giezeman G-J, Kettner L, Schirra S, Schönherr S. On the design of CGAL a computational geometry algorithms library. *Softw Pract Exp*. 2000;30(11):1167-1202.
97. Ren G, Younis RM. A Coupled XFEM-EDFM numerical model for hydraulic fracture propagation. ECMOR XVI-16th European Conference on the Mathematics of Oil Recovery, European Association of Geoscientists & Engineers; Vol. 1, 2018:1-16.
98. Massing A, Larson MG, Logg A. Efficient implementation of finite element methods on nonmatching and overlapping meshes in three dimensions. *SIAM J Sci Comput*. 2013;35(1):C23-C47.
99. Popinet S. *The GNU Triangulated Surface Library*. GTS website; 2004. https://gts.sourceforge.net/.
100. Sukumar N, Prévost J-H. Modeling quasi-static crack growth with the extended finite element method Part I: computer implementation. *Int J Solids Struct*. 2003;40(26):7513-7537.
101. Pereira JP, Duarte CA, Guoy D, Jiao X. hp-Generalized FEM and crack surface representation for non-planar 3-D cracks. *Int J Numer Methods Eng*. 2009;77(5):601-633.
102. Soghrati S, Ahmadian H. 3D hierarchical interface-enriched finite element method: implementation and applications. *J Comput Phys*. 2015;299:45-55.
103. Březina J, Exner P. Fast algorithms for intersection of non-matching grids using Plücker coordinates. *Comput Math Appl*. 2017;74(1):174-187.
104. Yang M, Nagarajan A, Liang B, Soghrati S. New algorithms for virtual reconstruction of heterogeneous microstructures. *Comput Methods Appl Mech Eng*. 2018;338:275-298.
105. Shewchuk JR. Triangle: engineering a 2D quality mesh generator and Delaunay triangulator. Proceedings of the Workshop on Applied Computational Geometry; 1996:203-222; Springer, New York.
106. Si H. TetGen, a Delaunay-based quality tetrahedral mesh generator. *ACM Trans Math Softw (TOMS)*. 2015;41(2):1-36.
107. Geuzaine C, Remacle J-F. Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities. *Int J Numer Methods Eng*. 2009;79(11):1309-1331.
108. Osher S, Sethian JA. Fronts propagating with curvature-dependent speed: algorithms based on Hamilton-Jacobi formulations. *J Comput Phys*. 1988;79(1):12-49.
109. Sethian JA, Smereka P. Level set methods for fluid interfaces. *Annu Rev Fluid Mech*. 2003;35(1):341-372.
110. Osher S, Paragios N. *Geometric Level Set Methods in Imaging, Vision, and Graphics*. Springer Science & Business Media; 2003.
111. Wang MY, Wang X, Guo D. A level set method for structural topology optimization. *Comput Methods Appl Mech Eng*. 2003;192(1-2):227-246.
112. Allaire G, Jouve F, Toader A-M. Structural optimization using sensitivity analysis and a level-set method. *J Comput Phys*. 2004;194(1):363-393.
113. Chern IL, Shu YC. A coupling interface method for elliptic interface problems. *J Comput Phys*. 2007;225(2):2138-2174.
114. Wendland H. Piecewise polynomial, positive definite and compactly supported radial functions of minimal degree. *Adv Comput Math*. 1995;4(1):389-396.
115. Wang S, Wang MY. Radial basis functions and level set method for structural topology optimization. *Int J Numer Methods Eng*. 2006;65(12):2060-2090.
116. Chew LP. Constrained Delaunay triangulations. *Algorithmica*. 1989;4(1):97-108.
117. Si H. *A Quality Tetrahedral Mesh Generator and Three-Dimensional Delaunay Triangulator*. Weierstrass Institute for Applied Analysis and Stochastic; 2006:81.
118. Turk G, Levoy M. Zippered polygon meshes from range images. *In Proceedings of the 21st annual conference on Computer graphics and interactive techniques*. 1994;311-318.
119. Gupta V, Duarte CA, Babuška I, Banerjee U. Stable GFEM (SGFEM): improved conditioning and accuracy of GFEM/XFEM for three-dimensional fracture mechanics. *Comput Methods Appl Mech Eng*. 2015;289:355-386.
120. Garzon J, O'Hara P, Duarte CA, Buttlar WG. Improvements of explicit crack surface representation and update within the generalized finite element method with application to three-dimensional crack coalescence. *Int J Numer Methods Eng*. 2014;97(4):231-273.

How to cite this article: Zhang J, Zhebel E, van den Boom SJ, Liu D, Aragón AM. An object-oriented geometric engine design for discontinuities in unfitted/immersed/enriched finite element methods. *Int J Numer Methods Eng*. 2022;123(21):5126-5154. doi: 10.1002/nme.7049