An Efficient Adaptive Procedure for Three-Dimensional Fragmentation Simulations

A. Pandolfi[†] and M. Ortiz[‡]

[†]Dipartimento di Ingegneria Strutturale
Politecnico di Milano, 20133 Milano, Italy

[‡]Graduate Aeronautical Laboratories
California Institute of Technology
Pasadena, CA 91125, USA

June 5, 2001
Submitted to: *Journal of Engineering with Computers*

Abstract

We present a simple set of data structures, and a collection of methods for constructing and updating the structures, designed to support the use of cohesive elements in simulations of fracture and fragmentation. Initially all interior faces in the triangulation are perfectly coherent, i. e., conforming in the usual finite element sense. Cohesive elements are inserted adaptively at interior faces when the effective traction acting on those face reaches the cohesive strength of the material. The insertion of cohesive elements changes the geometry of the boundary and, frequently, the topology of the model as well. The data structures and methods presented here are straightforward to implement and enable the efficient tracking of complex fracture and fragmentation processes. The efficiency and versatility of the approach is demonstrated with the aid of two examples of application to dynamic fracture.

1 Introduction

The essential interplay between geometry and mechanics comes into sharp focus in applications where the topology of the domain may change, oftentimes extensively, in the course of calculations. Fragmentation, which may result in a runaway proliferation of bodies in the form of fragments [1, 2, 3, 4], provides a case in point.

Camacho and Ortiz [5, 6] in two dimensions, and Pandolfi *et al.* [7, 8, 9] in three dimensions, have established the feasibility of: i) accounting explicitly for individual cracks as they nucleate, propagate, branch and possibly link up to form fragments; and ii) simulating explicitly the granular flow which ensues following widespread fragmentation. In this approach, cracks are allowed to form and propagate along element boundaries in accordance with a cohesive-law model [10, 11]. Clearly, it is incumbent upon the mesh to provide a rich enough set of possible fracture paths, an issue which may be addressed within the framework of adaptive meshing. In contrast to other approaches [10, 11] which require interfacial elements to be inserted at the outset along potential fracture paths, Camacho and Ortiz [5], and Pandolfi *et al.* [7, 8, 9], adaptively create new surface as required by the cohesive model by duplicating nodes along previously coherent element boundaries and inserting surface-like *cohesive elements* which encapsulate the fracture behavior of the solid. These elements are surface-like and are compatible with general bulk finite element discretizations of the solid, including those which account for plasticity and large deformations.

Pandolfi and Ortiz [12] have given an enumeration of the ways in which the topology and geometry of a three-dimensional finite-element model may evolve as a consequence of fracture and fragmentation, and have described the

actions which may be taken in order to update the boundary representation, or *Brep*, of the solid. Maintaining an up-to-date Brep is of the essence when meshing methods such as the advancing front [13, 14, 15] are utilized [16]. The Brep may also assist in the implementation of contact algorithms [17]. The geometrical framework proposed by Pandolfi and Ortiz [12] has extensively utilized in a broad range of applications involving fracture and fragmentation [7, 8, 9].

The Brep of a solid can become inordinately complex, and thus computationally costly to maintain, in applications involving profuse fragmentation. In addition, some meshing algorithms, e. g., those based on subdivision [18], and contact algorithms (e. g., [19]) do not require a full Brep for their implementation. In these cases, much simpler data structures suffice in order to account for fragmentation processes.

The purpose of this article is to present one such set of data structures, and a suite of methods for constructing and updating the structures. The data structures and methods are straightforward to implement and enable the efficient tracking of complex fracture and fragmentation processes. We also present two examples of application to dynamic fracture which illustrate the uncanny ability of the method to represent intricate geometrical and topological transitions resulting from crack branching, the nucleation of surfaces and interior cracks, crack coalescence, the detachment of fragments, and other effects.

The organization of the paper is as follows. In Section 2 we introduce three simple data structures where all necessary information pertaining to tetrahedra, faces and edges is stored. In Section 3 we present the suite of methods for evolving the data structures in response to fragmentation. Finally, in Section 4 we present examples of application which demonstrate the scope and the versatility of the approach.

2 Topological Data Structures for a 3D finite element model

We shall restrict our attention to tetrahedral triangulations and regard the computational model as a three-dimension simplicial complex [20, 21, 22]. In addition, we focus on the case of quadratic interpolation and, hence, every tetrahedron in the triangulation gives rise to a ten-node tetrahedral finite-element (e. g., [16]). Extensions to higher-order elements are straightforward but will not be pursued here. The local nodal numbering convention for the tetrahedral elements adopted here is shown in Fig. 1a. Each tetrahedron is bounded by exactly four triangular faces, Fig. 1b, and six edges, Fig. 1c. The faces and the edges can be oriented consistently by an appropriate ordering of the nodes, Figs. 1b and Fig. 1c. A face is bounded by exactly three edges. Each face is incident on exactly one tetrahedron, if the face is on the boundary of the body, or two tetrahedra, if the face is in the interior. Edges are incident on rings of varying numbers of tetrahedra. Likewise, a variable number of faces may be adjacent to an edge.

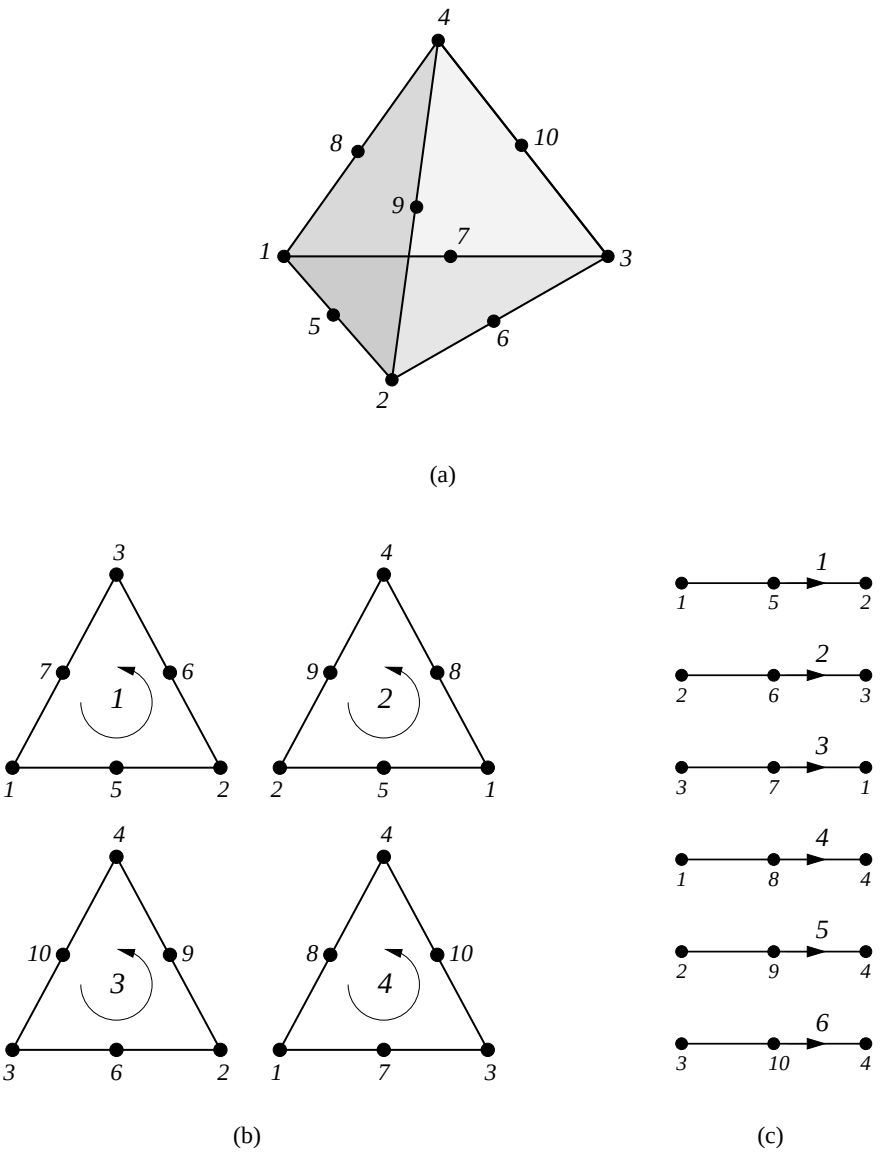
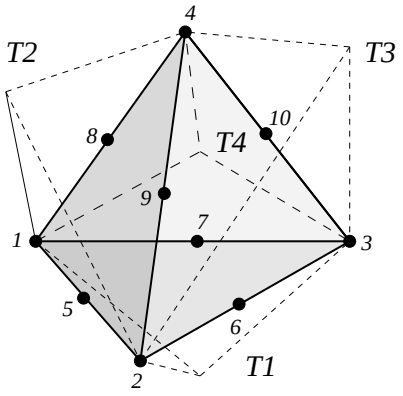


Figure 1: (a) A 10-nodes `tetrahedron`; (b) description of its four `facets`; (c) description of its six segments.



(a)

Figure 2: Tetrahedron connected to four adjacent tetrahedra.

We begin by introducing three data structures which collect these data and relationships, namely, the `tetrahedra`, `facets` and `segments` structures. In describing these structures, we adopt C syntax for definiteness. The data stored in the `tetrahedra` structure consists of the number of the element, the element connectivity, pointers to its six `segments`, pointers to its four `facets`, and pointers to the four adjacent `tetrahedra`, Fig. 3. Some of these pointers can be null. For instance, if a tetrahedron is incident on the boundary one or more of the pointers to adjacent tetrahedra are null. With a view to facilitating fragmentation simulations based on the use of cohesive elements [12], we include in the `tetrahedra` structure the number of the adjacent cohesive elements and the side of the cohesive elements incident on the tetrahedron.

```
/*-----------------------------------------------------------------*
 * Tetrahedron data structure                                      *
 *-----------------------------------------------------------------*/
typedef struct tetra
{
  struct tetra   *Link, *Rink;  /* Links */
  int             el;           /* Element no. */
  int             N[10];        /* Nodes */
  struct segment *S[6];         /* Pointers to Segments */
  struct facet   *F[4];         /* Pointers to Facets */
  struct tetra   *T[4];         /* Pointers to Tets */
  int             C[4];         /* Cohesive element */
  int             L[4];         /* Coh. el. side: 0=bottom, 1=top */
} Tetra;
```

Figure 3: Description of the `tetrahedron` data structure.

The data stored in the `facet` structure consists of an array containing six nodal numbers, ordered cyclically so as define an orientation for the face; pointers to the top and bottom tetrahedra, defined in accordance with the orientation of the face, Fig. 4a; pointers to the three edges incident on the face; and a boolean variable indicative of whether the

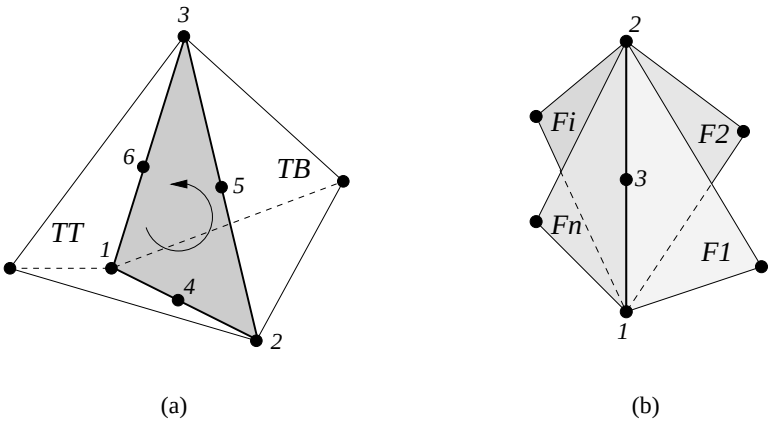


Figure 4: (a) A `facet` connected to two `tetrahedra`; (b) a `segment` connected to several `facets`.

face is interior or exterior, Fig. 5. Specifically, the identify top (bottom) or positive (negative) `tetrahedron` is that tetrahedron from which the nodes of the `facet` appear to be traversed counterclockwise (clockwise). For `facets` which are on the surface, the bottom `tetrahedron` is null. With a view to fragmentation applications, we additional collect the components of the unit normal; the critical traction for the insertion of a cohesive element, and an index designating the cohesive law to be used within the cohesive element.

```
/*----------------------------------------------------------------*
 * Facet data structure                                           *
 *----------------------------------------------------------------*/
typedef struct facet
{
  struct facet   *Link, *Rink;  /* Links */
  int             N[6];         /* Nodes */
  struct tetra   *T[2];         /* Pointers to Tets */
  struct segment *S[3];         /* Pointers to Segments */
  int             posi;         /* 0 = internal, 1 = external */
  double          v[3];         /* Components of the unit normal */
  double          sc;           /* Traction limit value */
  int             cm;           /* Cohesive material index */
} Facet;
```

Figure 5: Description of the `facet` data structure.

The edge structure `segment` contains an array with the numbers of its three nodes, Fig. 6, all of which are shared with one or more adjacent `facets` and with one or more adjacent `tetrahedra`. The local sequential numbering of the nodes defines the orientation of the edge. Owing to the variable environment of the edges in the triangulation, the structure `segments` additionally contains arrays of pointers which need to be allocated dynamically. These are the array of pointers to the tetrahedra and faces adjacent to the edge. The dimension of these arrays is also stored in the structure, as well as an additional boolean variable which designates the edge as interior or exterior.

```
/*--------------------------------------------------------------*
 * Segment data structure                                       *
 *--------------------------------------------------------------*/
typedef struct segment
{
  struct segment *Link, *Rink;  /* Links */
  int             N[3];         /* Nodes */
  int             nt;           /* No. adj tets */
  struct tetra  **T;            /* Pointers to the first Tet */
  int             nf;           /* No. adj Facets */
  struct facet  **F;            /* Pointer to the first Facet */
  int             posi;         /* 0 = internal, 1 = external */
} Segment;
```

Figure 6: Description of the `segment` data structure.

A C procedure for setting up the structures just described from a conventional finite-element connectivity array is shown in Fig. 7. As the connectivity table is traversed, each element is added to the linked list of `tetrahedra`. The corresponding six `segments` of each `tetrahedra` are identified, and the linked list of the `segments` and the list of incident `tetrahedra` of the `segments` are updated. Likewise, the four `facets` of each tetrahedra are identified, and the linked list of the `facets` and the list of incident `tetrahedra` of the `facets` are updated.

```
int sc[6][4] = {{0,4,1},{1,5,2},{2,6,0},    /*  segments topology  */
                {0,7,3},{1,8,3},{2,9,3}};
int fc[4][6] = {{0,1,2,4,5,6},{0,3,1,7,8,4}, /*  faces topology  */
               {1,3,2,8,9,5},{0,2,3,6,9,7}};
int p1[3] = {1,2,0};
/*-----------------------------------------------------------------*
 * void CreateDataStructures                                       *
 *-----------------------------------------------------------------*/
{
  Segment  *S[6], *G;
  Facet    *F[4], *R;
  Tetra    *T,    *Ti;
  int       ie, ic, i, j;
  int       n[10];
  TetraList   = NULL;    /*  Initialize the Tetrahedra list  */
  SegmentList = NULL;    /*  Initialize the Segments list  */
  FacetList   = NULL;    /*  Initialize the Facets list  */
  for (ie = 0; ie < TetraElementNumber; ie++) {
                                  /*  Add Tetrahedron  */
    for (j = 0; j < 10; j++) n[j] = connectivity[ie*nodes_element+j];
    TetraList = T = AddTetra (TetraList, n[0], n[1], n[2], n[3], n[4],
                              n[5], n[6], n[7], n[8], n[9], ie);
    for (ic = 0; ic < 6; ic++) {  /*  Add Segments  */
      S[ic] = SearchInSegments (T, n[sc[ic][0]], n[sc[ic][1]],
                                   n[sc[ic][2]]);
      TetraList->S[ic] = S[ic];
    }
    for (ic = 0; ic < 4; ic++) {  /*  Add Facets  */
      F[ic] = SearchInFacets (T, n[fc[ic][0]], n[fc[ic][1]],
                                 n[fc[ic][2]], n[fc[ic][3]],
                                 n[fc[ic][4]], n[fc[ic][5]]);
      for (j = 0; j < 3; j++) {   /*  Check Segments and Facets  */
         for (i = 0; i < 6; i++) {
            if ((S[i]->N[0] == F[ic]->N[j] &&
                 S[i]->N[2] == F[ic]->N[p1[j]]) ||
                (S[i]->N[2] == F[ic]->N[j] &&
                 S[i]->N[0] == F[ic]->N[p1[j]])) {
            F[ic]->S[j] = S[i];
            SegmentFacet (S[i], F[ic]);
          }
        }
      }
      /*  update the incident Facet  */
      TetraList->F[ic] = F[ic];
    }
  }
```

```
/*  Complete the Tet list with adj Tets  */
  for (T = TetraList; T != NULL; T = T->Rink) {
    for (i = 0; i < 4; i++) {
      R = T->F[i];
      if (R->T[0] == T) T->T[i] = R->T[1];
      if (R->T[1] == T) T->T[i] = R->T[0];
    }
  }
/*  Complete the Facets list defining if inside or outside  */
  for (R = FacetList; R != NULL; R = R->Rink) {
    if (R->T[0] == NULL) R->posi = 1;
    if (R->T[1] == NULL) R->posi = 1;
  }
/*  Complete the Segment list defining if inside or outside  */
  for (G = SegmentList; G != NULL; G = G->Rink) {
    for (i = 0; i < G->nf; i++) {
      if (G->F[i]->posi == 1) G->posi = 1;
    }
  }
  return;
}
```

Figure 7: This procedure fills the data structures described in Fig. 3-6. A new `segment` or `facet` insertion follows from the failure of a search in the corresponding linked lists. Subroutines are described in Fig. 8-10.

```
/*------------------------------------------------------------------*
 * Tetra *AddTetra                                                  *
 *------------------------------------------------------------------*/
Tetra
*AddTetra (Tetra *beg, int n1, int n2, int n3, int n4, int n5,
   int n6, int n7, int n8, int n9, int n10, int ie)
{
  Tetra *T;      /*  Add a Tet to the linked list  */
  T       = calloc (1, sizeof(Tetra));
  T->el   = ie + 1;
  T->N[0] = n1;  T->N[1] = n2;
  T->N[2] = n3;  T->N[3] = n4;
  T->N[4] = n5;  T->N[5] = n6;
  T->N[6] = n7;  T->N[7] = n8;
  T->N[8] = n9;  T->N[9] = n10;
  T->C[0] = 0;   T->C[1] = 0;  T->C[2] = 0;   T->C[3] = 0;
  T->Link = NULL;
  T->Rink = beg;
  if (beg != NULL) beg->Link = T;
  return T;
}
/*------------------------------------------------------------------*
 * Segment *SearchInSegments                                        *
 *------------------------------------------------------------------*/
Segment
*SearchInSegment (Tetra *T, int n1, int n2, int n3)
{
  Segment  *S;    /*  Search for a Segment in SegmentList  */
  int       nt;
  for (S = SegmentList; S != NULL; S = S->Rink) {
    if ((n1 == S->N[0] && n3 == S->N[2]) ||
        (n1 == S->N[2] && n3 == S->N[0])) {
      nt = S->nt;
      nt++;
      S->T = realloc (S->T, nt * sizeof(Tetra *));
      S->T[nt - 1] = T;
      S->nt = nt;
      return S;
    }
  }
  SegmentList = S = AddSegment (SegmentList, n1, n2, n3, T);
  return S;
}
```

Figure 8: Insertion of a new `tetrahedron` and check of the existence of a `segment` in the corresponding linked lists.

```
/*------------------------------------------------------------------*
 * Segment *AddSegment                                              *
 *------------------------------------------------------------------*/
Segment
*AddSegment (Segment *beg, int n1, int n2, int n3, Tetra *T)
{
  Segment *S;     /*  Add a Segment to the linked list  */
  S       = calloc (1, sizeof(Segment));
  S->posi = 0;
  S->N[0] = n1;  S->N[1] = n2;  S->N[2] = n3;
  S->nt   = 1;
  S->T    = calloc (1, sizeof(Tetra *));
  S->T[0] = T;
  S->nf   = 0;
  S->F    = calloc (1, sizeof(Facet *));
  S->Rink = beg;
  S->Link = NULL;
  if (beg != NULL) beg->Link = S;
  return S;
}
/*------------------------------------------------------------------*
 * Segment *SearchInFacets                                          *
 *------------------------------------------------------------------*/
Facet
*SearchInFacets (Tetra *T, int n1, int n2, int n3,
                 int n4, int n5, int n6)
{
  Facet *F;      /*  Search for a Facet in the FacetList  */
  for (F = FacetList; F != NULL; F = F->Rink) {
    if ((n1 == F->N[0] || n1 == F->N[1] || n1 == F->N[2]) &&
        (n2 == F->N[0] || n2 == F->N[1] || n2 == F->N[2]) &&
        (n3 == F->N[0] || n3 == F->N[1] || n3 == F->N[2])) {
      F->T[1] = T;
      return F;
    }
  }
  FacetList = F = AddFacet (FacetList, n1, n2, n3, n4, n5, n6, T);
  return F;
}
```

Figure 9: Insertion of a new `segment` and check of the existence of a `facet` in the corresponding linked lists.

```
/*------------------------------------------------------------------*
 * Facet *AddFacet                                                  *
 *------------------------------------------------------------------*/
Facet
*AddFacet (Facet *beg, int n1, int n2, int n3,
           int n4, int n5, int n6, Tetra *T)
{
  Facet *F;     /* Add a Facet to the linked list */
  F       = calloc (1, sizeof(Facet));
  F->posi = 0;
  F->N[0] = n1;  F->N[1] = n2;  F->N[2] = n3;
  F->N[3] = n4;  F->N[4] = n5;  F->N[5] = n6;
  F->T[0] = T;
  F->Rink = beg;
  F->Link = NULL;
  if (beg != NULL) beg->Link = F;
  return F;
}
/*------------------------------------------------------------------*
 * void *SegmentFacet                                               *
 *------------------------------------------------------------------*/
void
SegmentFacet (Segment *S, Facet *F)
{
  int nf = S->nf;
  int j;
  for (j = 0; j < nf; j++) if (S->F[j] == F) return;
  AddFaceToSegment (S, F);
  return;
}
/*------------------------------------------------------------------*
 * void *AddFaceToSegment                                           *
 *------------------------------------------------------------------*/
void
AddFaceToSegment (Segment *S, Facet *F)
{
  int nf = S->nf;
  nf++;
  S->F  = realloc (S->F, nf*sizeof(Facet));
  S->F[nf - 1] = F;
  S->nf = nf;
  return;
}
```

Figure 10: Insertion of a new `facet` in the corresponding linked list, update of the list of `facet` in the `segment` data structure.

3 Fragmentation

Ortiz and Pandolfi [23] developed a class of three-dimensional cohesive elements consisting of two six-node triangular facets (Fig. 11a). The opening displacements are described by quadratic interpolation within the element. The element is fully compatible with –and may be used to bridge– pairs of ten-node tetrahedral elements (Fig. 11b). The elements

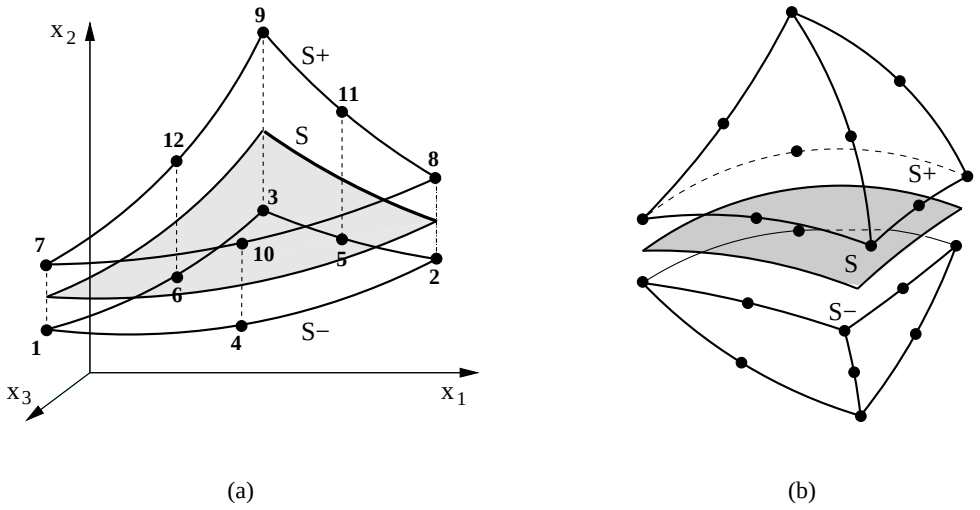


Figure 11: (a) Geometry and connectivities of 12-nodes cohesive element; (b) Assembly of 12-node triangular cohesive element and two 10-node tetrahedral elements.

are endowed with full finite-deformation kinematics and, in particular, are exactly invariant with respect to superposed rigid body translations and rotations.

We are here specially concerned with dynamic fragmentation, although static applications may be treated similarly. The analysis proceeds incrementally in time, e. g., by explicit dynamics. Following Camacho and Ortiz [5], cohesive elements are introduced adaptively at element interfaces as required by a fracture –or spall– criterion. For instance, fracture may be supposed to initiate at a previously coherent element interface when a suitably defined *effective traction* attains a critical value [5, 24, 23]. When the fracture criterion is met at an element interface, a cohesive element is inserted, leading to the creation of new surface. In this manner, the shape and location of successive crack fronts is itself an outcome of the calculations.

Next we discussed how the data structures defined in the preceding section may be used to support fragmentation simulations of the type just described. In particular, we specifically address the issue of how to update the data structures in response to fragmentation. The update procedure consists of two basic operations:

1. Selection of interior faces (`facets`) for the insertion of new cohesive elements.
2. Updating the data structures based on the selected `facets`.

We proceed to discuss these two steps in turn.

3.1 Selection of faces to be fractured

The selection of an interior face for the insertion of a cohesive element is based on the attainment of a suitable fracture criterion. The specific form of this criterion depends on the type of cohesive model used in calculations. For definiteness, we consider the class of cohesive laws proposed by [5, 24, 23], which are based on the introduction of an effective opening displacement and the corresponding work-conjugate effective traction:

$$t_{eff} = \sqrt{t_n^2 + \beta^{-2}|\boldsymbol{t}_s|^2} \tag{1}$$

where β is a tension-shear coupling parameter, t_n denotes the traction component normal to the `facet`, and $\boldsymbol{t}_s$ is the corresponding tangential traction, Fig. 12. If, in addition, cohesive surfaces are assumed to be rigid, or perfectly

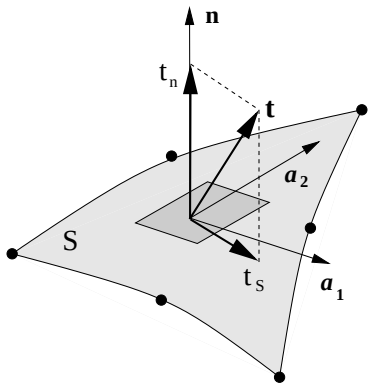


Figure 12: Normal and tangential components of the traction acting on an interface.

coherent, below a certain critical traction σ_c, or spall strength, then the appropriate form of the fracture criterion is:

$$t_{eff} \geq \sigma_c \tag{2}$$

This condition is checked for each internal face at the conclusion of a prespecified number of time steps in the calculations, and the faces where the criterion is met are flagged for subsequent processing. Evidently, criterion (2), which arises directly from the mechanics of cohesive fracture, drives the evolution of the geometrical description of the model. This connection exemplifies the tight coupling between mechanics and geometry which is characteristic of fragmentation simulations.

3.2 Data-structure update

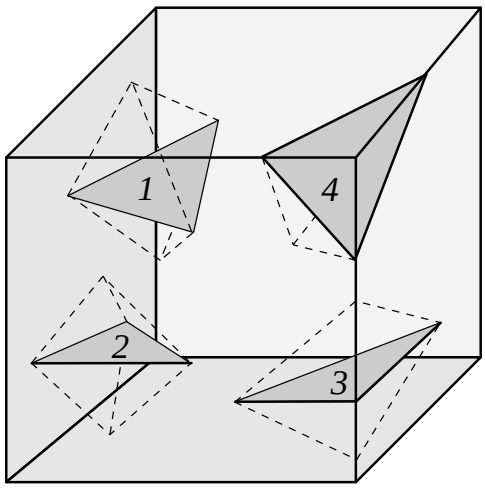


Figure 13: Classification of cases according to whether the fractured `facet` has zero (case 1), one (case 2), two (case 3) or three (case 4) `segments` on the boundary.

The operations to perform in order to process a fractured face depend critically on its position with respect to the external boundaries of the domain. Four main cases may be identified depending on whether the fractured `facet` has zero, one, two or three `segments` resting on the boundary, Fig. 13. This information is supplied by the flag *posi* of each `segment` contained in the `facet`. In all cases the fractured `facet` is duplicated and a new `facet` is added to the corresponding linked list. The remaining operations to be performed are:

1. No `segments` are on the boundary: No further operations, Fig. 14a.
2. One `segment` is on the boundary: the `segment` is duplicated by doubling the mid-side node, Fig. 14b.
3. Two `segments` are on the boundary: the `segments` are duplicated by doubling the mid-side nodes; the corner node is duplicated when it represents the sole remaining connection between the top and bottom `tetrahedra`, Fig. 14c.
4. Three `segments` are on the boundary: the `segments` are duplicated by doubling the mid-side nodes; a corner node is duplicated when it represents the sole remaining connection between the top and bottom `tetrahedra`, Fig. 14d. The new node is added to the top elements of the fractured `facet`.

The commented C code reported in listing 15 gives a detailed account of the operations to be performed in order to update the data structures and the connectivities. Steps referred to in the code are graphically displayed in Figs. 16-22.

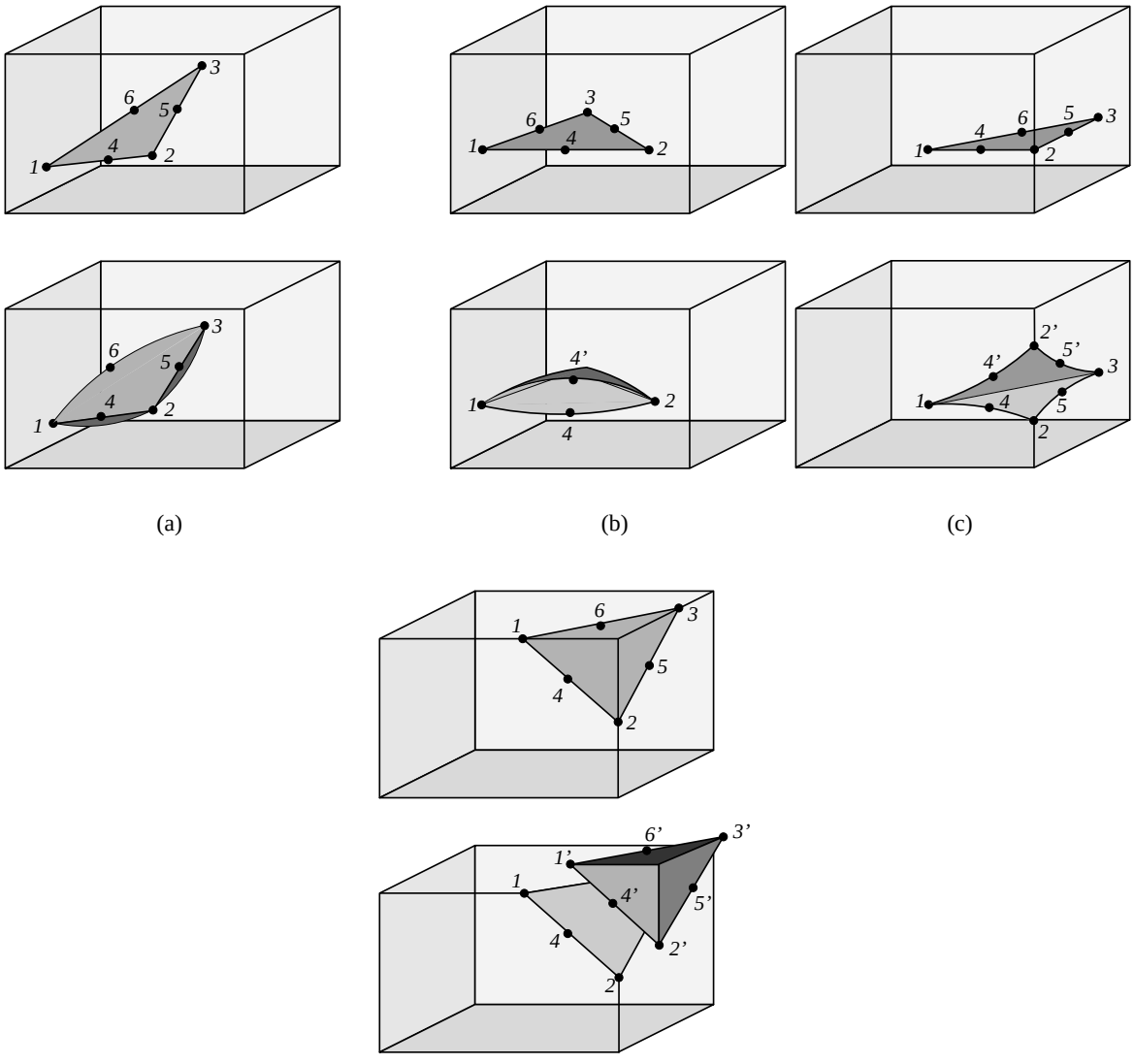


Figure 14: Topological changes induced by the fracturing of a `facet`. The primed numbers indicate the new inserted nodes. (a) Case 1: no `segments` on the boundary. No action; (b) Case 2: one `segment` on the boundary. The mid-side node is duplicated. (c) Case 3: two `segments` on the boundary. The corner node is duplicated and the mid-side nodes are duplicated; (d) Case 4: three `segments` on the boundary. All six nodes are duplicated.

```
/*------------------------------------------------------------*
 * void CreateSurface                                         *
 *------------------------------------------------------------*/
void
CreateSurface (int cohmate, Facet *facet)
{
  Tetra   *TEB, *TET;   /*  Top and Bottom tetrahedra  */
  int      open[3];     /*  Boundary sides counters  */
  Segment *S[3];        /*  Facet Segments  */
  Facet   *facetnew;    /*  New Facet  */
  int      i, j;
                        /* Get the six nodes on the Facet */
  for (i = 0; i < 6; i++) node[i] = nold[i] = facet->N[i];
  TET  = facet->T[0];
  TEB  = facet->T[1];
  for (i = 0; i < 3; i++) {
    S[i] = facet->S[i];
    open[i] = S[i]->posi;
  }                                  /* add a new facet */
  FacetList = facetnew = AddFacet (FacetList, node[0], node[2], node[1],
                                   node[5], node[4], node[3], TET);
  UpdateNewFacet (facet, facetnew, TEB, TET, cohmate);
  for (i = 0; i < 3; i++) {      /* Check the midside node */
    if (open[i] == 0) continue;
    nodes++;
    node[i + 3] = nodes;
    UpdateMidNodes (TET, S[i], node[i + 3], cohmate);
    DefineNodeVectors (node[i + 3], nold[i + 3]);
  }                                /* Check the corner node */
  counted = calloc (elements + 1, sizeof(int));
  for (i = 0; i < 3; i++) {
    if (open[i] == 0 || open[perm[i]] == 0) continue;
    for (j = 1; j <= TetraElementNumber; j++) counted[j] = 0;
    if (CornerToOpen (TET, TEB, nold[i]) == 0) continue;
    nodes++;
    node[i] = nodes;
    UpdateCornerNodes (TET, node[i], nold[i], cohmate);
    DefineNodeVectors (node[i], nold[i]);
  }
  free (counted);
  if (cohmate > 0) AddCohesive (cohmate, trli);
  return;
}
```

Figure 15: Main for the insertion of a new surface. The subroutines are reported in Figs. 16-22.

```
/*----------------------------------------------------------------*
 * void UpdateNewFacet                                            *
 *----------------------------------------------------------------*/
void
UpdateNewFacet (Facet *facet, Facet *facetnew,
                Tetra *TEB, Tetra *TET, int cohmate)
{
  Segment *S[3];
  int      move[3] = {2,1,0};
  int      i;
/*  Update the old Facet */
  facet->T[1] = NULL;
  facet->posi = 1;
/*  Update the new Facet */
  facetnew->posi = 1;
/*  Add Segments to new Facet and new Facet to Segments */
  for (i = 0; i < 3; i++) {
    S[i] = facet->S[i];
    S[i]->posi = 1;
    facetnew->S[move[i]] = S[i];
    S[i] = AddFaceToSegment (S[i], facetnew);
  }
/*  Update the Facet and adjacency of the Tets */
  for (i = 0; i < 4; i++) {
    if (TEB->F[i] == facet) {
      TEB->T[i] = NULL;
      if (cohmate > 0) {
        TEB->C[i] = elements + 1;
        TEB->L[i] = 0;
      }
    }
    if (TET->F[i] == facet) {
      TET->F[i] = facetnew;
      TET->T[i] = NULL;
      if (cohmate > 0) {
        TET->C[i] = elements + 1;
        TET->L[i] = 1;
      }
    }
  }
  return;
}
```

Figure 16: Insertion of a new `facet` into the data structures.

```
/*--------------------------------------------------------------*
 * void UpdateMidNodes                                          *
 *--------------------------------------------------------------*/
void
UpdateMidNodes (Tetra *TET, Segment *SO, int nn, int cohmate)
{
  Segment *SN;
  Tetra   *T;
  int      n1 = SO->N[0];
  int      n2 = SO->N[2];
  int      ot = SO->nt;
  int      of = SO->nf;

/*  Insert a new Segment */
  SegmentList = SN = AddSegment (SegmentList, n1, nn, n2, TET);
  SN->posi = 1;
  SN->nt   = 0;

/*  Transfer Tets from the old to the new Segment */
  SN->T = realloc (SN->T, ot*sizeof(Tetra));
  TetraInSegments (TET, SO, SN, cohmate);
  SN->T = realloc (SN->T, SN->nt*sizeof(Tetra));
  SO->T = realloc (SO->T, SO->nt*sizeof(Tetra));

/*  Transfer Facets between the two Segments */
  SN->F = realloc (SN->F, of*sizeof(Facet));
  FacetInSegments (SO, SN);
  SN->F = realloc (SN->F, SN->nf*sizeof(Facet));
  SO->F = realloc (SO->F, SO->nf*sizeof(Facet));
  return;
}
```

Figure 17: Changes in the data structures required by the duplication of a mid node. First, a new `segment` is added (see Fig. 8). Then, the list of the `tetrahedra` incident to the original `segment` is traversed and the `tetrahedra` are moved from the old to the new `segment` (Fig. 18). Finally, the list of the `facets` incident to the original `segment` is traversed and the `facets` are moved from the old to the new `segment` (Fig. 19).

```
/*------------------------------------------------------------------*
 * void TetraInSegments                                             *
 *------------------------------------------------------------------*/
void
TetraInSegments (Tetra *TET, Segment *SO, Segment *SN, int cohmate)
{
  Tetra *T;
  int    el = (TET->el - 1)*nodes_element;
  int    no = SO->N[1];
  int    nn = SN->N[1];
  int    ot = SO->nt;
  int    nt = SN->nt;
  int    i, j;
  for (i = 4; i < 10; i++) { /*  Change the connectivity  */
    if (TET->N[i] != no) continue;
    TET->N[i] = connectivity[el + i] = nn;
    if (cohmate > 0) UpdateCohesive (TET, nn, no, 3);
  }
  for (j = 0; j < 6; j++) {  /*  Change the Segment in the Tet  */
    if (TET->S[j] != SO) continue;
    TET->S[j] = SN;
  }
  for (i = 0; i < ot; i++) { /*  Move Tet from old to new Segment  */
    T = SO->T[i];
    if (T != TET) continue;
    SN->T[nt] = T;
    nt++;
    SO->T[i] = NULL;
  }
  SN->nt = nt;
  for (j = i + 1; j < ot; j++, i++) SO->T[i] = SO->T[j];
  ot--;
  SO->nt = ot;
  for (i = 0; i < ot; i++) { /*  Look at adjacent Tets  */
    T = SO->T[i];
    for (j = 0; j < 4; j++) {
      if (T->T[j] != TET) continue;
      TetraInSegments (T, SO, SN, cohmate);
      return;
    }
  }
  return;
}
```

Figure 18: Update the `tetrahedra` connected to the `segment`.

```
/*------------------------------------------------------------------*
 * void FacetInSegments                                             *
 *------------------------------------------------------------------*/
void
FacetInSegments (Segment *SO, Segment *SN)
{
  Tetra *T;
  Facet *F;
  int    nn = SN->N[1];
  int    of = 0;
  int    nf = 0;
  int    i, j, k;
  for (i = 0; i < SN->nt; i++) { /*  Update connectivities  */
    T = SN->T[i];
    for (j = 0; j < 4; j++) {
      F = T->F[j];
      for (k = 0; k < 3; k++) {
        if (F->S[k] != SO) continue;
        F->S[k]     = SN;
        F->N[3 + k] = nn;
        SN->F[nf]   = F;
        nf++;
        break;
      }
    }
  }
  SN->nf = nf;
  for (i = 0; i < SO->nt; i++) { /*  Update the incident Facets  */
    T = SO->T[i];
    for (j = 0; j < 4; j++) {
      F = T->F[j];
      for (k = 0; k < 3; k++) {
        if (F->S[k] != SO) continue;
        SO->F[of] = F;
        of++;
        break;
      }
    }
  }
  SO->nf = of;
  return;
}
```

Figure 19: Update the `facets` connected to the `segment`.

```
/*------------------------------------------------------------------*
 * void UpdateCohesive                                              *
 *------------------------------------------------------------------*/
void
UpdateCohesive (Tetra *T, int nn, int no, int add)
{
  int ce, cc, i, j;
  for (i = 0; i < 4; i++) {
    ce = T->C[i] - 1;
    if (ce < 0 || ce == elements) continue;
    cc = ce*nodes_element + T->L[i]*6 + add;
    for (j = 0; j < 3; j++) {
      if (connectivity[cc + j] != no) continue;
      connectivity[cc + j] = nn;
    }
  }
  return;
}
/*------------------------------------------------------------------*
 * void CornerToOpen                                                *
 *------------------------------------------------------------------*/
int
CornerToOpen (Tetra *TET, Tetra *TEB, int no)
{
  Tetra *T;
  int    i, j, res;
  counted [TET->el] = 1;
  for (i = 0; i < 4; i++) {
    T = TET->T[i];
    if (T == NULL) continue;
    for (j = 0; j < 4; j++) {
      if (T->N[j] != no) continue;
      if (T == TEB) return 0;
      if (counted[T->el] > 0) continue;
      if (CornerToOpen (T, TEB, no) == 0) return 0;
    }
  }
  return 1;
}
```

Figure 20: Update of the cohesive element connectivity. The recursive subroutine `CornerToOpen` checks the existence of additional connections between the Top and the Bottom `tetrahedron` before duplicating a corner node.

```
/*-----------------------------------------------------------------*
 * void UpdateCornerNodes                                          *
 *-----------------------------------------------------------------*/
void
UpdateCornerNodes (Tetra *TET, int nn, int no, int cohmate)
{
  Segment *S;
  Facet   *F;
  Tetra   *T;
  int      el = (TET->el - 1)*nodes_element;
  int      j, i;

  counted[TET->el] = 0;       /*  Flag the Tet as checked  */

  for (j = 0; j < 4; j++) {   /*  Check Tet and connectivity  */
    if (TET->N[j] != no) continue;
    TET->N[j] = nn;
    connectivity[el + j] = nn;
    if (cohmate > 0) UpdateCohesive (TET, nn, no, 0);
    break;
  }
  for (j = 0; j < 6; j++) {   /*  Update Segments  */
     S = TET->S[j];
     if (S->N[0] == no) S->N[0] = nn;
     if (S->N[2] == no) S->N[2] = nn;
  }
  for (i = 0; i < 4; i++) {   /*  Update Facets  */
    F = TET->F[i];
    if (F->N[0] == no) F->N[0] = nn;
    if (F->N[1] == no) F->N[1] = nn;
    if (F->N[2] == no) F->N[2] = nn;
  }
   for (i = 0; i < 4; i++) {  /*  Next adjacency  */
    T = TET->T[i];
    if (T == NULL) continue;
    if (counted[T->el] == 0) continue;
    UpdateCornerNodes (T, nn, no, cohmate);
  }
  return;
}
```

Figure 21: Recursive subroutine to perform changes in the data structures required by the duplication of a corner node. First, the `tetrahedron` connectivity is update. Then the incident `segments` and `facets` are updated. Finally, the adjacent `tetrahedra` are checked.

```
/*------------------------------------------------------------------*
 * void DefineNodeVectors                                           *
 *------------------------------------------------------------------*/
void
DefineNodeVectors (int node, int nold)
{
  int nd = nodes*dof_node;
  int nn = (node - 1)*dof_node;
  int no = (nold - 1)*dof_node;
  int j;
  coordinates = realloc (coordinates, nd*REAL);
  for (j = 0; j < dof_node; j++) {
    coordinates[nn + j] = coordinates[no + j];
  }
  return;
}

/*------------------------------------------------------------------*
 * void AddCohesive                                                 *
 *------------------------------------------------------------------*/
void
AddCohesive (int cohmate, double trli)
{
  int     k, i;
  k = nodes_element * elements;
  elements++;
  connectivity     = realloc (connectivity,     (k + 1)*sizeof(int));
  element_material = realloc (element_material, elements*sizeof(int));
  for (i = 0; i < 6; i++) {
    connectivity[k + i    ] = nold[i];
    connectivity[k + i + 6] = node[i];
  }
  element_material [elements - 1] = cohmate;
  return;
}
```

Figure 22: Reallocation of the nodal vectors (`DefineNodeVectors`) and element vectors (`AddCohesive`). These subroutine must be completed with the reallocation of all the node and element based vectors.

4 Examples of application

In this section we demonstrate the scope and versatility of the procedures described above with the aid of two examples of application. The fracture criterion (2) is adopted in order to determine the onset of fracture in a `facet` [5, 24, 23]. In all calculations, the material is modeled as nonlinear elastic obeying a Neohookean constitutive law [25]. In particular, full finite kinematics is taken into account.

The first application concerns the simulation of the dynamic fragmentation of a three-point bend PMMA specimen with a sharp precrack contained within its symmetry plane. The central point of the top side of the specimen is suddenly imparted a uniform velocity 40 m/s at time $t = 0$, and the velocity is held constant thereafter. The length of the specimen is 8.4 mm, its width 1 mm and its height 1.4 mm (Fig. 23).

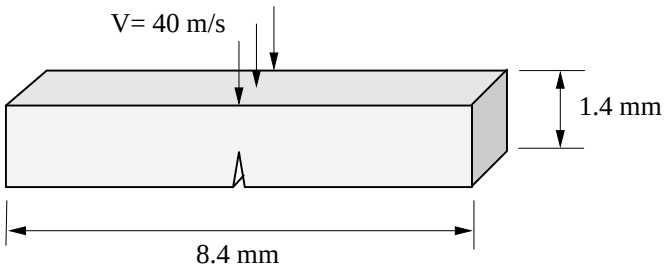


Figure 23: Three point bend test in PMMA specimen, loading conditions. The impact is simulated imposing a uniform and constant velocity V along the central line. Owing to the high impact speed, the supports are not simulated.

The model is meshed into 4,260 ten-node tetrahedra and 6,420 nodes. The mesh is finer in the central part and is gradually coarsened away from the crack (Fig. 24a).

The material parameters employed in the calculations are: specific fracture energy per unit area, or critical energy release rate, $G_c = 210$ N/m; critical cohesive stress, or spall strength, $\sigma_c = 100$ MPa; tension-coupling constant $\beta = 1$; Young's modulus $E = 3$ GPa; Poisson's ratio $\nu = 0.38$; and mass density $\rho = 1180$ kg/m^3. The equations of motion are integrated in time by recourse to Newmark's explicit algorithm with parameters $\beta = 0$, $\gamma = 1/2$ [26, 27]. The time step used in the calculations is $\triangle t = 1.7$ x $10^{-4}\mu$s.

Figs. 24b and c show the computed fracture and fragmentation pattern after 20 μs. The ability of the approach to track the evolution of complex crack geometries is noteworthy. As may be seen in the figure, the geometrical update procedure effectively deals with the intricate geometrical and topological transitions which result from crack branching, the nucleation of surfaces and interior cracks, crack coalescence, the detachment of fragments, and others.

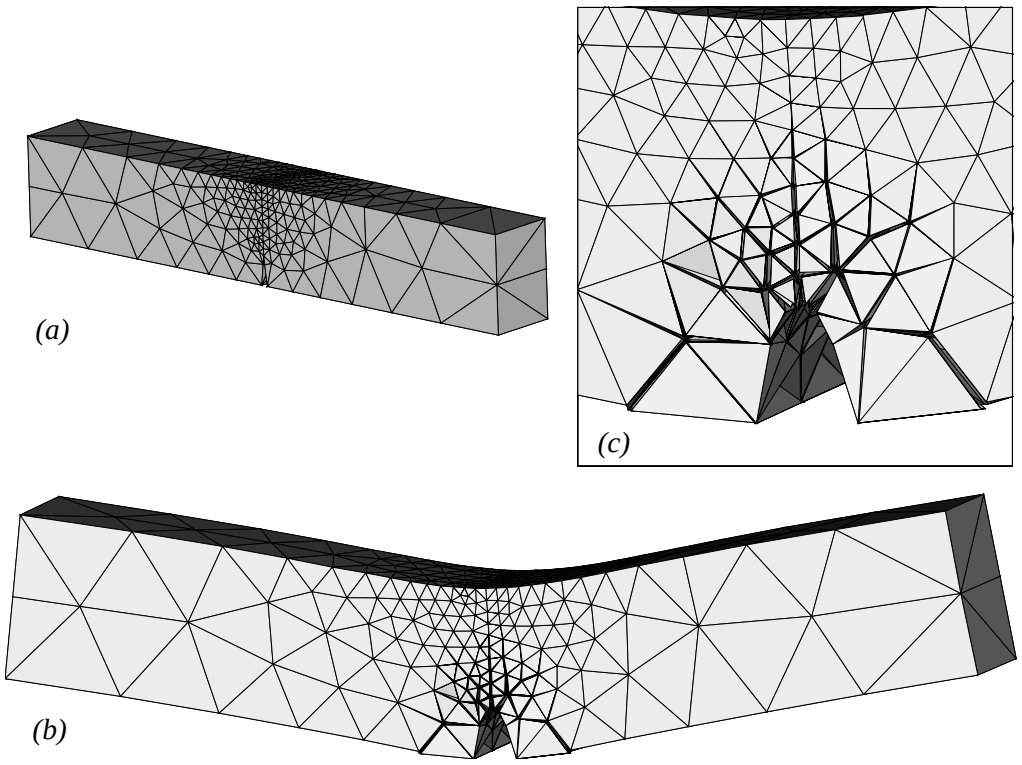


Figure 24: Fragmentation algorithm applied to the three-point bend dynamic test in PMMA: (a) initial mesh; (b) final configuration; (c) detail of the fracture and fragmentation pattern in the final configuration.

The second example of application concerns the simulation of dynamic crack branching in a PMMA thin square plate. The plate is 3 mm in length and 0.3 mm in thickness. The specimen contains an initial 0.25 mm sharp notch. Constant normal velocities, tending to open the crack symmetrically in mode I, are prescribed on the top and bottom edges of the specimen. The magnitude of the prescribed velocities corresponds to a nominal strain rate of $0.002/\mu$s. In addition, the initial velocity field is assumed to be linear in the coordinate normal to the crack and to correspond to a uniform rate of deformation of $0.002/\mu$s throughout the specimen, Fig. 25.

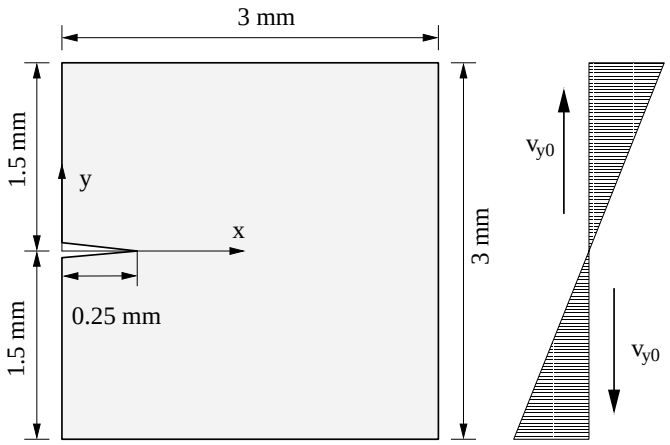


Figure 25: Geometry of the PMMA square plate and mode-I loading conditions.

The model is meshed into 14,319 ten-node tetrahedra and 25,936 nodes, Fig. 26a. The equations of motion are integrated using Newmark's explicit algorithm with time step $\triangle t = 6$ x $10^{-}5\mu$s. The material parameters used in the calculation are: specific fracture energy per unit area $G_c = 176.15$ N/m; critical cohesive stress $\sigma_c = 324$ MPa; tension-shear coupling constant $\beta = 2$; Young's modulus $E = 3.29$ GPa; Poisson's ratio $\nu = 0.35$; and mass density $\rho = 1190$ kg/m^3. Figs. 26b and c show the fracture pattern at the conclusion of the test. As may be seen, at a low prescribed strain rate the crack tends to grow within its plane, and it branches only when it senses the proximity of the free surface on the right side of the specimen. Results from a similar calculation at a higher nominal strain rate of $0.01/\mu$s are shown in Figs. 27a-c. The mesh in this case contains 6,363 ten-node tetrahedra and 11,569 nodes, and the stable time step is $\triangle t$ = 6 x $10^{-}5$ μs. Figs. 27b and c show the crack patterns at the conclusion of the test. Initially the crack remains within its plane and accelerates steadily. As a certain crack speed is attained, the crack begins to issue lateral branches. These branches consume additional fracture energy, thereby limiting the mean crack speed. As the crack approaches the free surface, the extent of branching increases steadily.

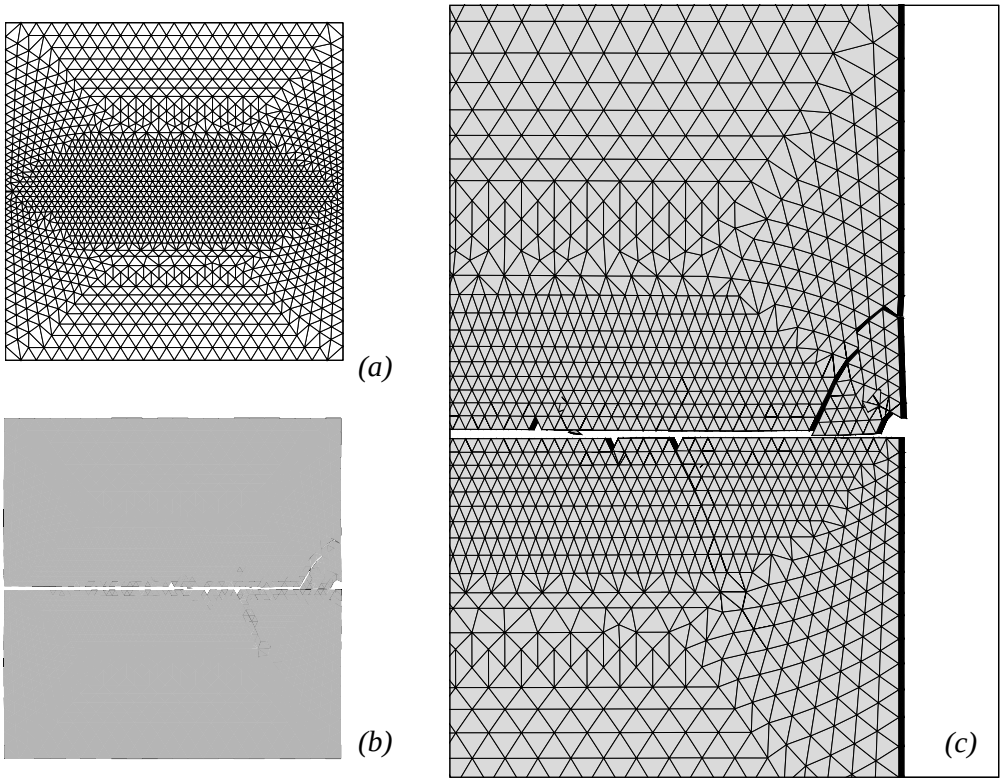


Figure 26: Mode-I dynamic fracture test in PMMA at nominal strain rate $0.002/\mu$s. (a) Initial mesh and prescribed and initial velocities; (b) final configuration; (c) detail of the branching in the final configuration at time 5 μs.

As in the previous example, the ability of the method to deal with complex geometrical and topological transitions simply and effectively is noteworthy. In particular, cracks are allowed to branch unimpeded, connect with free surfaces or with other cracks to form fragments.

We conclude this section by emphasizing that the simulations presented above, while representative of a broad class of engineering materials and loading conditions, are not intended as a validation of the cohesive model but as a demonstration of the computational methodology. Detailed validation studies, including extensive comparisons with experiment, based on test configurations similar to those just described may be found elsewhere [7, 8, 9, 28].

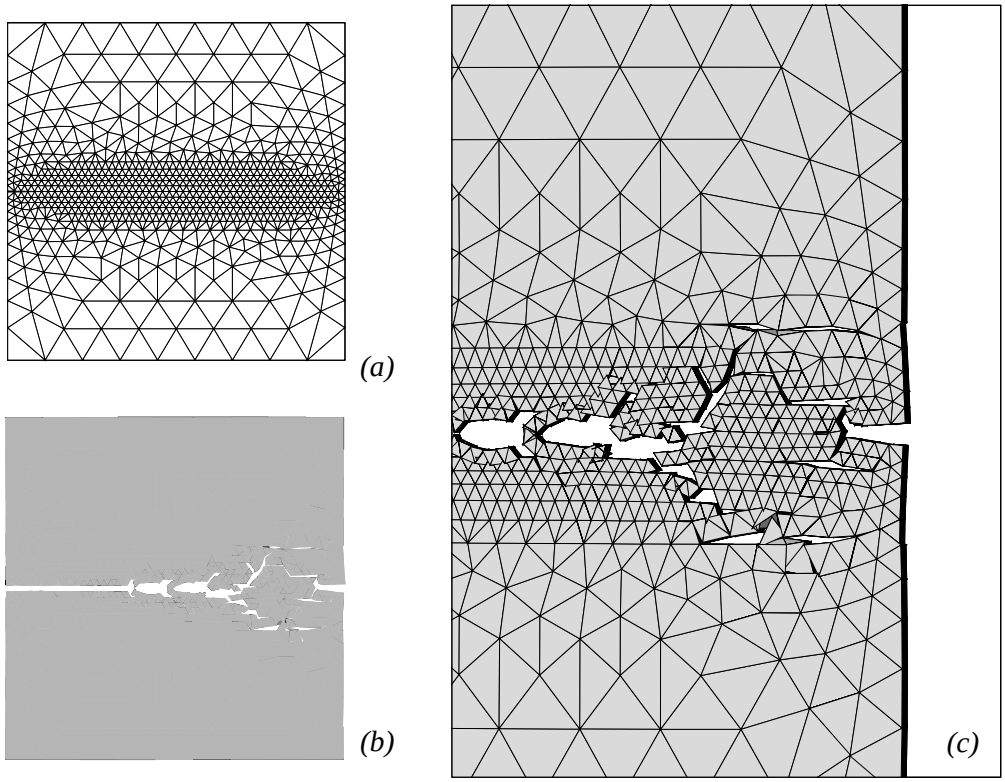


Figure 27: Mode-I dynamic fracture test in PMMA at a nominal strain rate of $0.01/\mu$s. (a) Initial mesh; (b) final configuration; (c) detail of the branching in the final configuration at time 5 μs.

5 Summary and Conclusions

In cohesive theories of fracture, material separation is governed by a suitable cohesive law. In finite-element simulations based on a tetrahedral triangulation of the domain of analysis, decohesion and opening may conveniently be restricted to interior triangular faces. The cohesive laws considered here are rigid up to the attainment of the cohesive strength of the material. Consequently, initially all the faces in the triangulation are perfectly coherent, i. e., conforming in the usual finite element sense. Cohesive elements are inserted adaptively at interior faces when the effective traction acting on those face reaches the cohesive strength of the material. The insertion of cohesive elements changes the geometry of the boundary and, frequently, the topology of the model as well.

We have presented a simple set of data structures, and a collection of methods for constructing and updating the structures, designed to support the use of cohesive elements in simulations of fracture and fragmentation. The data structures and methods are straightforward to implement and enable the efficient tracking of complex fracture and fragmentation processes. The examples of application discussed here illustrate the uncanny ability of the method to represent intricate geometrical and topological transitions resulting from crack branching, the nucleation of surfaces and interior cracks, crack coalescence, the detachment of fragments, and others.

Acknowledgments

The support of the Army Research Office through grant DAA-H04-96-1-0056 is gratefully acknowledged. We are also grateful for support provided by the DoE through Caltech's ASCI/ASAP Center for the Simulation of the Dynamic Behavior of Solids. The assistance provided by Dr. Chengxiang Yu with the numerical examples is gratefully acknowledged.

References

[1] J.E. Field, Q. Sun, and D. Townsend. Ballistic Impact of Ceramics. *Inst. Phys. Conf. Ser. No 102: Session 7, Paper presented at Int. Conf. Mech.Prop. Materials at High Rates of Strain, Oxford, 1989*, 1989.

[2] M. E. Kipp, D. E. Grady, and J. W. Swegle. Numerical and Experimental Studies of High-Velocity Impact Fragmentation. *International Journal of Impact Engineering*, 14:427–438, 1993.

[3] R. L. Woodward, W. A. Gooch, R. G. O'Donnell, W. J. Perciballi, B. J. Baxter, and S. D. Pattie. A Study of Fragmentation in the Ballistic Impact of Ceramics. *International Journal of Impact Engineering*, 15(5):605–618, 1994.

[4] A.J Piekutowski. Fragmentation of a Sphere Initiated by Hypervelocity Impact with a thin sheet. *International Journal of Impact Engineering*, 17:627–638, 1995.

[5] G. T. Camacho and M. Ortiz. Computational modelling of impact damage in brittle materials. *International Journal of Solids and Structures*, 33(20-22):2899–2938, 1996.

[6] M. Ortiz. Computational Micromechanics. *Computational Mechanics*, 18:321–338, 1996.

[7] A. Pandolfi, P. Krysl, and M. Ortiz. Finite element simulation of ring expansion and fragmentation: The capturing of length and time scales through cohesive models of fracture. *International Journal of Fracture*, 95:1–18, 1999.

[8] G. Ruiz, M. Ortiz, and A. Pandolfi. Three dimensional finite-element simulation of the dynamic brazilian tests on concrete cylinders. *International Journal for Numerical Methods in Engineering*, 48(7):963–994, 2000.

[9] G. Ruiz, A. Pandolfi, and M. Ortiz. Three-dimensional cohesive modeling of dynamic mixed-mode fracture. *International Journal for Numerical Methods in Engineering*, 2000. submitted.

[10] M. Ortiz and S. Suresh. Statistical Properties of Residual Stresses and Intergranular Fracture in Ceramic Materials. *Journal of Applied Mechanics*, 60:77–84, 1993.

[11] X. P. Xu and A. Needleman. Numerical Simulations of Fast Crack Growth in Brittle Solids. *Journal of the Mechanics and Physics of Solids*, 42:1397, 1994.

[12] A. Pandolfi and M. Ortiz. Solid modeling aspects of three-dimensional fragmentation. *Engineering with Computers*, 14(4):287–308, 1998.

[13] J. Peraire, M. Vahdati, K. Morgan, and O. C. Zienkiewicz. Adaptive Remeshing for Compressible Flow Computations. *Journal of Computational Physics*, 72:449–466, 1987.

[14] J. Peraire, J. Peiro, L. Formaggia, K. Morgan, and O.C. Zienkiewicz. Finite Element Euler Computations in Three Dimensions. *International Journal for Numerical Methods in Engineering*, 26:2135–2159, 1988.

[15] R. Löhner and P. Parikh. Generation of Three-Dimensional Unstructure Grids by the Advancing-Front Method. *International Journal for Numerical Methods in Fluids*, 8:1135–1149, 1988.

[16] R. Radovitzky and M. Ortiz. Tetrahedral mesh generation based on node insertion in crystal lattice arrangements and advancing-front-Delaunay triangulation. *Computer Methods in Applied Mechanics and Engineering*, 187(3–4):543–569, 2000.

[17] C. Kane, E. A. Repetto, M. Ortiz, and J. E. Marsden. Finite element analysis of nonsmooth contact. *Computer Methods in Applied Mechanics and Engineering*, 180:1–26, 1999.

[18] J. F. Molinari and M. Ortiz. Three-dimensional adaptive meshing by subdivision and edge-collapse in finite-deformation dynamic-plasticity problems with application to adiabatic shear banding. 2001.

[19] A. Pandolfi, C. Kane, M. Ortiz, and J. E. Marsden. Time-discretized variational formulation of nonsmooth frictional contact. *International Journal for Numerical Methods in Engineering*, 2001. in press.

[20] A. A. G. Requicha. Representations for Rigid Solids: Theory, Methods and Systems. *Computing Surveys*, 12:437–465, 1980.

[21] M. Mantyla. *An Introduction to Solid Modeling*. Computer Science Press, Rockwille, Maryland, 1988.

[22] C. M. Hoffmann. *Geometric and Solid Modeling*. Morgan Kaufmann Publishers, San Mateo, California, 1989.

[23] M. Ortiz and A. Pandolfi. A class of cohesive elements for the simulation of three-dimensional crack propagation. *International Journal for Numerical Methods in Engineering*, 44:1267–1282, 1999.

[24] A. De-Andrés, J. L. Pérez, and M. Ortiz. Elastoplastic finite element analysis of three-dimensional fatigue crack growth in aluminum shafts subjected to axial loading. *International Journal of Solids and Structures*, 36(15):2231–2258, 1999.

[25] J. E. Marsden and T. Hughes. *Mathematical Foundations of Elasticity*. Prentice Hall, 1983.

[26] T. Belytschko. An overview of semidiscretization and time integration procedures. In T. Belytschko and T. J. R. Hughes, editors, *Computational Methods for Transient Analysis*, pages 1–65. North-Holland, 1983.

[27] T. J. R. Hughes. Analysis of transient algorithms with particular reference to stability behavior. In T. Belytschko and T. J. R. Hughes, editors, *Computational Methods for Transient Analysis*, pages 67–155. North-Holland, 1983.

[28] C. Yu, A. Pandolfi, and M. Ortiz. 3d cohesive investigation on branching for brittle materials. 2001.