Architecture tradeoffs of integrating a mesh generator to partition of unity enriched object-oriented finite element software

Cyrille Dunant[*] — Phu Nguyen Vinh[**] — Mourad Belgasmia[***] Stéphane Bordas[***] — Amor Guidoum[*]

[*] *EPFL STI IMX Laboratoire de Matériaux de Construction*
MXG 241, Station 12 CH-1015 Lausanne
{cyrille.dunant, amor.guidoum}@epfl.ch

[**] *ENISE Laboratoire de Tribologie et de Dynamique des Systèmes*
F-42000 Saint-Etienne
nvinhphu@gmail.com

[***] *EPFL ENAC IS Laboratoire des Structures et des milieux Continus, GC A2 454 (Bâtiment GC) Station 18 CH-1015 Lausanne*
stephane.bordas@alumni.northwestern.edu, mourad.belgasmia@epfl.ch

ABSTRACT. *We explore the tradeoffs of using an internal mesher in a* XFEM *code. We show that it allows an efficient enrichement detection scheme, while retaining the ability to have well-adapted meshes. We provide benchmarks highlighting the considerable gains which can be expected from a well designed architecture. The efficiency of the proposed algorithm is shown by solving fracture mechanics problems of densely micro-cracked bodies including adaptive mesh refinement.*

RÉSUMÉ. *Nous explorons les coûts et gains tenant à l'inclusion d'un mailleur au sein d'un code* XFEM. *Cette architecture permet de rendre efficace la détermination de l'enrichissement, en conservant la possibilité de disposer de maillages bien adaptés. Des évaluations mettent en évidence les gains considérables en temps de calcul qui peuvent être atteints.*

KEYWORDS: XFEM, *multiple enrichment, partition of unity, architecture ; meshing, mesh-geometry interaction, multiple cracks; local mesh refinement, implementation.*

MOTS-CLÉS : XFEM, *enrichissement multiple, partition de l'unité, architecture, maillage, interaction maillage-géométrie, fissures multiples, affinement local du maillage, implémentation.*

REMN – 16/2007. X-FEM, pages 237 to 258

1. Introduction

The extended finite element method is a numerical method for modeling discontinuities and singularities within a standard finite element framework (Moës *et al.*, 1999; Babuška *et al.*, 1997; Melenk *et al.*, 1996). The XFEM has been successfully applied to numerous solid mechanics problems (Sukumar *et al.*, 2003). The XFEM has also been used to model computational phenomena in areas such as fluid mechanics, phase transformations (Chessa *et al.*, 2002), material science and biofilm growth (Bordas, 2003; Bordas *et al.*, 2005a), among others. The XFEM can be implemented within a finite element code with relatively small modifications: variable number of degrees of freedom (DOFS) per node; mesh-geometry interaction; enriched stiffness matrices; numerical integration and derivation, linear and non-linear solvers. The high degree of reflexivity and dynamism required by the the FEMand XFEM methods are best served by an object oriented approach, such as that described by (Zimmermann *et al.*, 1992; Bordas *et al.*, n.d.c). However, compared to FEM, XFEM entails a higher degree of intricacy between the various components of the code, which necessitates a well-designed architecture, allowing for a high degree of reflexivity. A typical usage would be adding cracks to a standard elastic problem (Ventura *et al.*, 2005). The solving of such a problem would be done the usual way: (1) mesh the problem, (2) assemble the discretized form into a system of linear equations, (3) apply boundary conditions, (4) solve the system. In XFEM, only (2) is modified. The additional steps are:

1) find the elements affected by the cracks,

2) modify their elementary matrices accordingly.

The second step has been described extensively (Belytschko *et al.*, 1999; Areias *et al.*, 2005; Strouboulis *et al.*, 2000), but the first much less. In fact, for a very complex problem, it can be quite computationally costly to find all the elements affected by a crack by simply iterating on all the elements. It is also inelegant, and difficult to extend to a more rich and varied enrichment schemes. Using the information *already* available internally to the mesher, the element-finding step can be performed in $O(e \log(n))$, where e is the number of enrichment items, and n the number of elements.

In this paper, we will cover the specifics of computational geometry, meshing schemes and enrichment in the optic of highlighting the architectural and computational advantages of using an integrated approach, as opposed to a more traditional toolbox-type setup. We will first explore the specifics of geometrical representations, then describe the meshing algorithm we have developed and the computational facilities it provides with respect to computing geometrical interactions. We will then show how this allows for considerable computing gains in operations of the XFEM method. Finally, we will present a few examples of setups made easy to simulate with such an architecture as well as crack growth in 2D. Recent papers (Budyn, 2004) describing multiple enrichment features go no higher than a few tens. We suggest an architecture which scales up to thousands easily – the limiting factor being the large amount

of degrees of freedom then required for the modelisation. We exercise the proposed algorithm through micro-cracking examples with hundreds of cracks, which, to the authors' knowledge, is a unique result in computational mechanics, be it approached with the standard FEMor the XFEM.

2. Architecture overview

We hae built a setup in which the assembler is strongly coupled to the rest of the program, yielding various advantages in term of performance and code maintainability. Our setup comprises:

1) a problem descriptor,

2) a mesher,

3) an assembler,

4) a solver.

Those items are not individually optimized to the point where they could be, as our interests lies mostly in the interaction between parts and getting a functionning system. Which means, as we study higly geometrically heterogeneous materials a system allowing for high-level, statistical description of a *class* of problems, the generation of a statistical sample of special cases and their subsequent resolution.

This in turn imposes the availability of a fully automatic mesher. Our fully automatic mesher yields unstructured meshes solely from randomly ordered input points[1]. We found that the algorithms used to find point-triangles interaction used could be extended to any geometry-triangle interactions.

In turn, we found it was extremely useful to have a mesher for the integration step of enriched elements. We also found that whith sufficient abstraction, the integration step by subtriangulation could be correctly performed independently of the function integrated.

2.1. *Problem descriptor*

The so-called *problem descriptor* allows the user to input the geometry, the behaviour laws and the precision desired.

It is based on objects such as `Sample`, `Inclusion`, `Pore`, `Crack`, etc. In turn, those have a member `behaviour`, such as `LinearElastic`. An exemple setup would be:

1. Randomly ordering the input points yields optimal performance of the mesher.

```
FeatureTree F(new Sample(/*dims*/)) ;
F.addFeature(new Pore(/*dims*/)) ;
Inclusion * inclusion = new Inclusion(/*dims*/) ;
inclusion->setBehaviour(new Stiffness()) ;
F.addFeature(inclusion) ;
```

Listing 1: *Simple setup with a sigle pore and inclusion*

The problem descriptor is responsible for the generation of the sampling points that are input in the mesher. They are selected in such a way that the mesh obtained be conformant.

2.2. *Mesher*

The inner workings of the mesher are described in more detail further below. The mesher is a simple Delaunay tesselator using an internal tree-like structure allowing point insertion in $O(\log(n))$. The same structure, derived from geometrical considerations, make it possible to also find triangles overlapping a given geometry in $O(m \log(n))$ with m the number of triangles covered by the geometry. This is useful in:

1) localized refinement,

2) mesh-geometry interaction finding,

3) geometry-geometry interaction finding,

4) sub-triangulation for non-polynomial function integration.

2.3. *Assembler and solver*

The assembler and solver are of a classical type. However, the solver could be linked to the mesher so as to yield an adaptative mesh.

3. Finding geometrical interactions

Computational geometry is a well studied field, whose applications in fields as diverse as ray tracing (Various, n.d.), games (Magnenat *et al.*, 2000) and physics engines (Magnenat, 2000) are well explored. It has yielded efficient algorithms for the detection and localization of various geometrical interactions.

3.1. *Representation*

Before finding interactions, geometrical objects must be represented. They can be separated into two categories:

1) simple objects described as analytic functions,

2) general shapes.

A computational geometry module should attempt to provide most of the usual objects (spheres, circles, lines, points...) and their interactions. If one wanted to be general, some assumptions on the shape of the general objects would still be needed, and a general representation in the form of Bézier patches or curves (Chiyokura *et al.*, 1983), or more generally NURBS[2] is chosen. All those objects can be abstracted to a purely virtual interface, which would, in C++, look something like this:

```
class Geometry{ protected:
  GeometryType geometryType ;
public :
  Geometry() ;
  virtual ~Geometry() ;
  GeometryType getGeometryType() const ;
  virtual bool in(const Point *) const;
  virtual bool on(const Point *) const;
  bool intersects(const Geometry *) const ;
}
```

Listing 2: *General geometry interface, for an abstract description of such diverse things as triangles and Bézier patches. You can note the presence of a field* `GeometryType`*: it serves as a very fast geometry type identifier for all operations (such as intersections) that depend on the specific properties of the geometries*

Of course, implementing `intersects` for all possible combinations is tedious, and becomes increasingly cumbersome as the object library grows. To overcome this limitation, one could for example decide to express all shapes as Bézier patches or NURBS. It is interesting to notice that such an implementation makes no assumption on the dimensionality of the described objects. It therefore makes it uniquely suited to represent such concepts as level sets (Sethian, 1999).

Enrichment features are well represented as deriving from general geometries, and providing a means to enrich the elements they affect. On the one hand it is possible to attach to them objects representing physical laws, and the other hand they can be considered as purely geometrical objects when it is efficient to do so.

2. Non Uniform Rational B-Spline.

```
class EnrichmentFeature : public Geometry{
  //like a geometry
  EnrichmentType getEnrichmentType() const ;
  virtual void enrich(ElementSet *) const ;
}
```

Listing 3: *An example interface for enrichment features, which then could be implemented in features like cracks, inclusions...*

Using this structure, the `enrich` member would then look something like that:

```
void EnrichmentFeature::enrich(ElementSet *s) const{
  std::vector<Element *> elements = s->conflicts(this) ;
  for(std::vector<Element *>::iterator i = elements.begin() ;
      i != elements.end() ;
      ++i )  {
      this->enrichElement(*i) ;
  }
}
```

Listing 4: *Implementation of the enrichment loop*

From the listings 3 and 4 we start seeing the intricacy: the element set is responsible for finding the subset of conflicting elements. This is a design choice, as the `EnrichmentFeature` could have done it itself. Simply, if the elements are stored in a clever way, the element set can find the targets much more efficiently. Figure 1 shows a possible inheritance tree for such a setup.

This shows the need for a mesh database more than for a mesher. It is however already appearing that a lot of the code in the mesher is useful outside as well.

3.2. *Information pertinence*

Computational geometry provides the information pertaining to the relative position of a point and a given geometrical entity. It can also provide the location of intersections. Both those pieces of information are needed to decide whether an element is a target for enrichment or not. Unfortunately, although we can easily check whether a given element *is* a target for enrichment, it is on the other hand much more difficult to have a complete and exhaustive list of *all* the targets. This is because the information given is attached to a geometrical entity, and contains no topological information. In a FEM program, the mesh contains such information, in the form of a connectivity table. But again, although this is sufficient to recreate a table of neighbours it is inefficient to do so. Not only because it is computationally expensive, but also because *the information was already computed and has been discarded*.

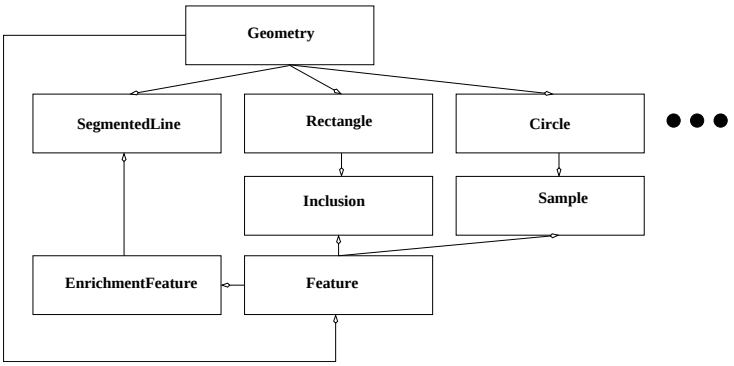


Figure 1. *Multiple inheritance showing a possible scheme separating purely geometrical constructs from physical entities, insuring a high level of code reuse as well as abstraction*

3.3. *Special case for the elements*

The elements are typically also geometrical entities, which means that they also implement the interactions. This is necessary for enrichment, but higher order elements are in fact general, simply connex polygons, which induces potentially computationally expensive checks. If the mesh is well-adapted to the geometry of the problem, that is, it does not rely overly on the additional degrees of freedom provided by the higher-order elements to insure an acceptable approximation of the boundaries, the elements can generally be treated as their low-order counterparts, thus making the process of finding the enriched nodes much faster. This leads to an interesting consideration: the process of finding the enriched elements is, all other things being equal, dominated by the efficiency of the `in()` method. This method would be most efficient if elements were circles or spheres, with only three multiplications and a floating point comparison.

```
bool Sphere::in(Point *p) const{
  double dist = computeSquaredDistance(this->center, p) ;
  return dist < this->r_squared ;
}
```

Listing 5: *Efficient implementation of the `in()` function for a sphere or circle*

This is exactly what is possible if the mesh has been generated by Delaunay triangulation. This comes from the fact that a Delaunay triangulation is defined by the following property: *No four points are within any of the meshes triangles' circumcircles*. Other meshing schemes, which are not based on circles or spheres interactions can still be stored in a efficient tree-like structure, but the exact implementation of the

`in` condition is going to be either less flexible in the case of a purely regular rectangular mesh, or more costly.

4. Description of the meshing scheme used

Two kinds of meshing schemes dominate, the advancing front and the n-Tree family. They both have advantages and disadvantages when considered strictly from the meshing point of view, but we will try to show that for our purposes, the advancing front family is not well-suited. Of course various hybrid methods are used, for example to determine refinement zones. Typically the input consists of only the analytical description of the geometry, and it is up to the mesher to do the sampling. Alternatively, the sampling is given, and the mesher then outputs the elements corresponding to the given sampling points, constrained by the boundaries of the geometrical entities to mesh.

4.1. n-*tree*

Trees are an efficient mean of organizing and retrieving data (Frederickson, 1991). In particular, for space-dividing schemes, a parent-children relationship ordered by the spatial position allows the building of such trees on the fly as the meshing goes. A very simple example of such meshing is a quad-tree.

But a tree is also a possible representation of the current state of a Delaunay type triangulation. Points are pre-generated and then fed randomly into a triangulation algorithm. Each new insertion determines the elimination of a set of triangles, and the creation of new ones (Devillers *et al.*, 1992). The new triangles are the children of the old with which they share the same conflict condition. The conflict condition in the case of Delaunay triangulation is the one described in the Listing 5, with the used circle being the circum circle of the triangle. The Listing 6, adapted from (Devillers *et al.*, 1992), describes a simple version of such an algorithm.

The algorithm works thus:

1) if we already checked this element, return,

2) if this element is not conflicting with the point, return,

3) add self to return,

4) attempt insertion in all children, insert result,

5) attempt insertion in all neighbours, insert result,

6) return.

```
std::vector<TreeItem *> * TreeItem::conflicts(const Point *p){
  std::vector<TreeItem *> * ret = new std::vector<TreeItem *>() ;
  std::vector<TreeItem *> * temp ;
  this->visited = true ;   // mark as checked
  if(!inCircumCircle(p))   // essential conflict ?
    return ret ;
  if(this->isAlive())      //else, if "alive"
    ret->push_back(this) ; //put it in list.
  //check the children -- note the recursion
  for (size_t i  = 0 ;  i < child.size() ; i++)  {
    if( !child[i]->visited )
      ret->insert(ret->end(),
                  son[i]->conflicts(p)->begin(),
                  son[i]->conflicts(p)->end()
                 ) ;
  }
  //check the neighbours -- note the recursion
  for (size_t i  = 0 ;  i < neighbour.size() ; i++)  {
    if( !neighbour[i]->visited )
      ret->insert(ret->end(),
                  neighbour[i]->conflicts(p)->begin(),
                  neighbour[i]->conflicts(p)->end()
                 ) ;
  }
  return ret ; // return the list of conflicting triangle
}
```

Listing 6: *Recursive building of the list of affected elements. The `inCircumCircle()` method is used instead of a more classical `in()`. This is the advantage of Delaunay triangulation: it only uses cheap checks for its formation*

Once a list of affected triangles has been obtained, one simply deletes them and replaces them with new triangles defined by the boundary of the set of affected triangles, and the inserted point. The combination of the conflict-checking algorithm and the point insertion scheme generates a set of triangles, arranged in a tree whose leaves form the actual triangulation which will be used.

4.2. *Mesher efficiency*

The mesher we coded is fairly efficient. Compared to gmesh, we get the following numbers:

Gmsh is three times as fast, but the order of the algorithms is the same. We conclude that our mesher is *good enough*, though it could certainly be optimised. The tradeof is offset by having full control over the API and the additional usage we get from the mesher, using its readily available data strucures.

Table 1. *Time taken for a mesh with ≈ 50 000 points. This finally yields 200000 degrees of freedom*

gmsh	own mesher
5 s	15 s

This is further reinforced by the fact that the meshing step is in fact a relatively low-time-consuming one, as is highlighted in the Figure 2.

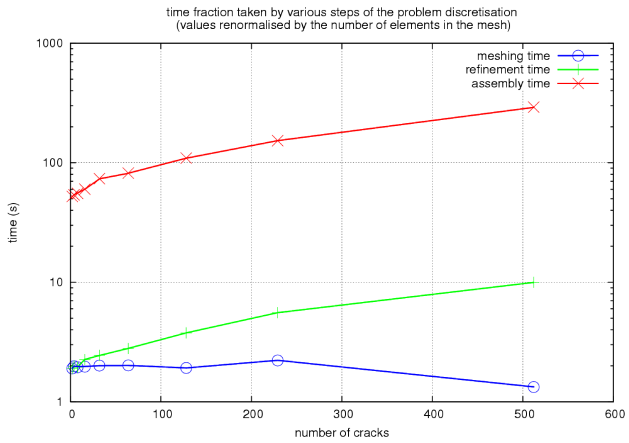


Figure 2. *Localised refinement and matrix assembly are much costlier operation than the meshing itself. The renormalisation highlights the asymptotic behaviour of the various algorithms involved*

4.3. *Mesh quality*

The quality of the initial mesh depends a lot on the choice of discretization points, this independently of the meshing algorithm chosen. To enhance the quality of the mesh, two options are available: (1) refinement, (2) relocation, based on some quality criterion. Relocation is equally feasible independently of the meshing algorithm initially chosen, but refinement is not, necessitating the insertion of points *in* the already meshed domain, and thus requiring an algorithm capable of this. This limitation can be overcome by using an hybrid meshing technique allowing for both modes. A simple hybrid mesher could generate the original mesh using the advancing front technique, and also provide a quad-tree like refinement algorithm.

5. Comparison with a mesh database

5.1. *Similarities and limitations*

A mesh database, such as AOMD (Remacle *et al.*, 2000), would provide the same accessors and iterators as the mesher, thus from the point of view of efficiency of the mesh-geometry interaction detection step, it is equivalent. However, the database would need to be rebuilt each time the mesh is rebuilt, leading to unnecessary overhead. The algorithmic advantages are however the same for static meshes. But the overall algorithmic efficiency can never be better than $O(n)$, where n is the number of elements, as each element is to be entered at least once.

5.2. *Exported information* vs. *internal representation*

Once a mesh has been generated, it is fully defined by (1) the coordinates of the nodes, (2) the list of elements defined by a list of nodes. In a typical FEM code, only this information is exported and loaded in the assembler-solver. This is the *external representation* of the mesh. However, as we have just seen, much more information is used in the mesher. This information is typically discarded in the toolbox approach case, as it is useless: separate programs provide separate services and communicate through data files containing only the minimum necessary. However, one can build all sorts of element-finding routines on the model of the `conflicts(Point *)`. This only requires the programmer to provide a function checking for geometrical conflict.

6. Finding elements affected by enrichment schemes

Enrichment schemes are determined on a geometrical basis. Elements in and around enrichment features are selected, and additional degrees of freedom are added to their nodes, based on the nature of the feature. This typically means that a given feature should be located in terms of elements. This is easy in the case of a so-called *structured mesh* where the spatial position of nodes determines their numbering. It is however much harder in the case of unstructured meshes, as those typically produced by the meshing scheme described above, used in practice. With only a list of elements, the only way to find those within a given geometrical limit is to run through the list, and check every element. This is dramatically inefficient. The slightly better case involves having also a list of neighbours for each element. Then, one needs only to find a single affected element and proceed through the neighbours. This is faster, but *is still asymptotically the same order of computation* because the first element still needs to be found, and this is $O(n)$ for randomly listed elements. Indeed, on average one will still have to check for half the elements before finding one that matches the condition and, in effect, only the constant part of the iteration is reduced.

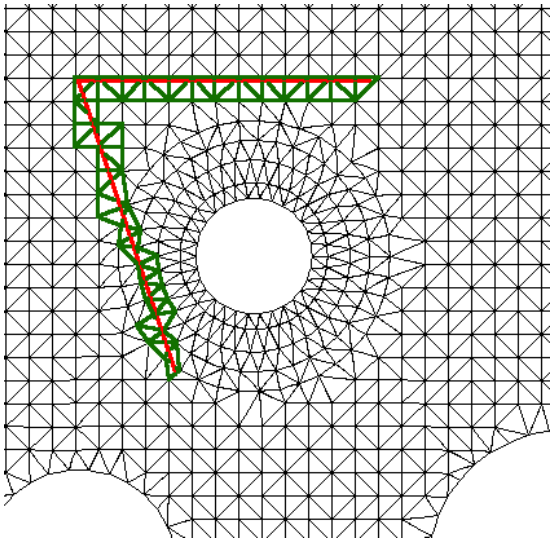
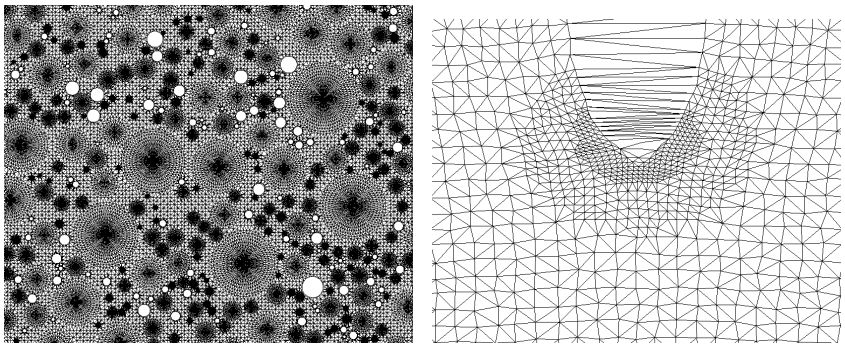


Figure 3. *Detecting a geometry in a complex case. the bold line interacting with the elements, in bold, the elements found to be interacting*

Finally if the elements are arranged in a tree, the order of the computation is dramatically reduced from $O(en)$ where e is the number of features and n the order of elements, to $O(e \log(n))$.

6.1. *Using the information provided by the mesher*

A mesh produced by a traditional, separate mesher only provides the list of nodes in each elements and the coordinates of the nodes. This means that the data provided are very inadequate for the purpose of enrichment. What one can do is rebuild the list of neighbours before the enrichment target finding step. This is done simply by counting the common nodes for each elements. It is however impossible to re-create a tree structure from the simple connectivity data, except in certain trivial cases. Rather, it is possible, but only by discarding the element information and regenerating a mesh from the sampling points. If a mesher is integrated with the FEM/ XFEM program, it becomes possible to enrich a mesh with a large amount of features, because the mesher provides all the information. It is also possible to setup complex experiments using various geometry generators (see for example Figure 4a).



(a) complex microstructure (b) local mesh refinement

Figure 4. *Detail of a mesh autogenerated with a random arrangement of inclusions and pores (a) and deformed mesh around a crack tip with local mesh refinement (b)*

6.2. *Fracture mechanics and* a posteriori *error applications*

Modern fracture mechanics applications and schemes require a range of geometrical routines which are very easily provided using the architecture suggested. It is necessary to evaluate domain integrals for the computation of stress intensity factors. This requires finding elements within a ball of given radius, or, alternatively, elements cut by a sphere. This is trivially accomplished here. But even more interesting is that when using special shape functions, the integration step involves generating a submesh on the elements, and summing the sub-integrals. This is typically more precise than simply using a very large amount of Gauß points. In addition, the idea of "fixed area enrichment" (Laborde *et al.*, 2004) rests on enriching sets of elements within a given surface area, a ball centered on the crack tip. Using the above general geometry description, the implementation and integration of such a scheme becomes very easy. *A posteriori* error estimations such as ZZ-type error estimates typically require knowledge about nodes' neighbouring elements, in order to construct the enhanced stress and strain fields on the computational mesh. Other error estimators (Bordas *et al.*, n.d.d) also require identifying nodes within a ball centered on the computational point where the smoothed field or derivative is desired. Again, the mesher integration is the easy answer to the program architecture challenge.

7. Usage considerations

From a usage point of view, one can find many advantages in having an integrated mesher. It allows for greater abstraction. It allows for greater extensibility. It per-

mits the easy implementation of various optimization technique which require a link between the solver and the mesh and allows for very complex enrichment schemes.

7.1. *Auto-meshing*

One of the most important features allowed by the integration of the mesher in the core of the XFEM code is the possibility to have an auto-mesher. Although auto-meshing cannot be certain to give good results, it is extremely practical to test various configurations, which might be generated based on a set of rules. For example, one might seek the effects on the apparent mechanical properties of a sample, of a given distribution of inclusions or/and cracks. With an auto-mesher, it is simply a matter of generating the geometry of the features, then the program can mesh, assemble and solve at once.

7.2. *Auto-refinement*

Criteria exist to evaluate both the general quality of the mesh, and the local quality of each element. Based on those criteria, the mesh can be refined or smoothed automatically. A tree-like structure in the mesher allows for not only adding sampling points, but also removing them when they are no longer needed. This permits the limitation of the total number of sampling points in an efficient way. Of course, as illustrated in Figure 4b, simple *a priori* refinement can easily be added. More involved error-based refinement are seamlessly introduced in the proposed architecture.

Very powerful also is the possibility of error-based remeshing (Bordas *et al.*, n.d.d). For time-dependent calculations, a once well-adapted mesh might becomes inadequate and yields faulty results. This leads on one hand to a necessary remeshing, and on the other to the recomputation of the mesh-geometry interactions. This cannot be done efficiently by using a mesh database, as it would have to be rebuilt with the new, modified mesh, an operation which is needlessly costly.

7.3. *Enrichment*

Enrichment is a relatively new method, and though new schemes of enrichment are regularly published (Legrain *et al.*, 2005), the question of the performance of the actual enrichment step is rarely mentioned in the literature. However, if one imagines a case where thousands of enrichment features are needed – dense micro-fissuration, complex micro-structures, cavitation and bubble/fluid interaction – the cost of finding the enriched node becomes prohibitive. The trivial method is in the order of $n \times e$ where, n is the number of elements, and e the number of enrichment items. This leads to catastrophic computing times, especially if interactions *between* enrichment items need also be calculated – crack junction, crack-boundary interaction, crack-material interface interaction… Indeed, we believe, it is *impossible*, without a mesh database,

to efficiently perform such computations. The downside of our approach, however, is an increased memory consumption.

8. Architectural considerations

8.1. *Integration*

When following an object-oriented design approach (Mackerle, 2000), the facilities used by each object to perform its task are available for the others to use. The following data is needed for a calculation:

1) the geometrical setup of the problem,

2) the neighbourhood relationships between elements,

3) the geometrical information for the elements,

4) the assembled matrices of the problem,

5) the boundary conditions, per node, or per geometrical definition.

This data is provided by different objects, or entities. The geometrical data and boundary conditions are provided by the user through the interface of the program itself. The mesher provides the geometry of the elements, and the neighbourhood relationships. The assembled matrices are the responsibility of an assembler, and then passed to a solver. Figure 5 illustrates the interactions and data flows between the different elements of the program.

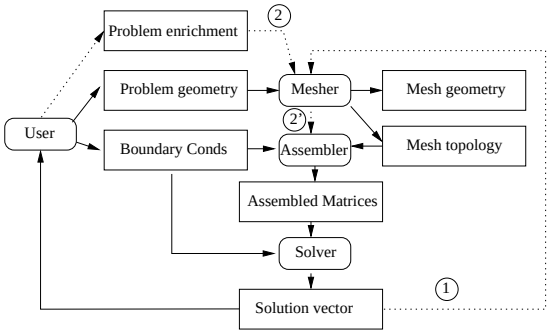


Figure 5. *The additional data paths made possible by the integration of the mesher:* 1*, for adaptive meshing, and* 2 *and* 2′ *for computationally efficient enrichment*

8.2. *Functionality provided by the API*

The API or application programming interface is the collection of functions and data-structures available for the programmer's use. Their richness and design make the

difference between some extensibility or no extensibility (Zimmermann *et al.*, 1992). The API of the integrated mesher must be able to provide the required services. However, for architectural reasons, it needs only know about *geometry*, and not specifically about elements or physics. The most important service provided is getting the list of elements affected by a geometrical feature. An item of the tree must therefore implement a `conflicts(Geometry *)` method, which will be implemented following the template of the `conflicts(Point *)` already present. The condition `inCircumCircle()` is replaced by a more generic condition, as described in the Listing 7. This is all that is necessary to get computationally efficient enrichment strategies.

```
bool DelaunayTreeItem::isAffected(Geometry *g){
  for(size_t i = 0 ; i < this->boundingPoint.size() ; i++)  {
    if(g->in(boundingPoint[i]))
      return true ;
  }
  return false ;
}
```

Listing 7: *Generic checking of whether an item of the triangulation is affected by a given geometry*

9. Numerical examples

9.1. *Densely micro-cracked body and high number of enrichment features*

The suggested architectures allows for the simulation of densely micro-cracked materials, as presented in Figure 6, but also more classical problems can be solved conveniently.

The advantage of the method lies in the fact that it allows for very large number of enrichment features, and therefore the simulation of complex setups. The feature size has an important impact on the computing time, as each element to enrich has to be tested. In this benchmark, the discovery step is in two phases: (1) find a possible subset of the elements, using the `conflict` method and (2) an exhaustive search in this subset. This is of course much less efficient than using the neighbourhood relationships, but only by a linear factor, as the original element is found in $O(n)$. XFEM is extremely efficient in itself to model very complex setups. The meshing of a complex geometry can be a daunting task, if a very-well adapted mesh is required. Using area enrichment, one can dispense with moving refinement of the mesh along the crack tip, thanks to asymptotic enrichment[3]. However, the mesh-geometry detection step remains, as all elements within a ball centered on the crack tip must be asymptotically enriched. This step is made very simple with the proposed architecture.

3. In linear elastic fracture mechanics, enrichment with the Westergaard solution.

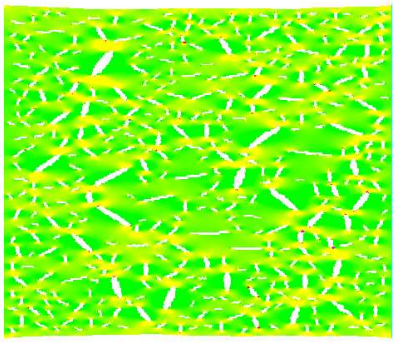


Figure 6. *One of the principal stresses, computed around approximatively 400 cracks. The mesh-geometry computation step took less than a second. Such a case uses approximatively 200 000 degrees of freedom. In this case, the displacements are set on the left and right of the sample, symetrically, with the top and bottom frontiers left free. The behaviour is purely linear-elastic. In this case, the longest step is the solving of the linear system itself, taking 30 min. on a desktop computer. The assembly step takes 20% longer, than an equivalently sized assembly with no enrichment*

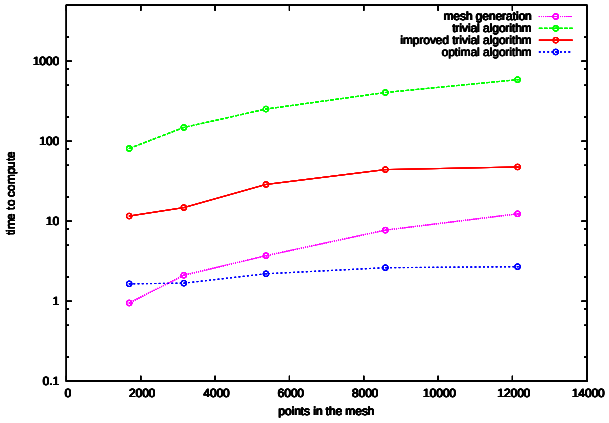
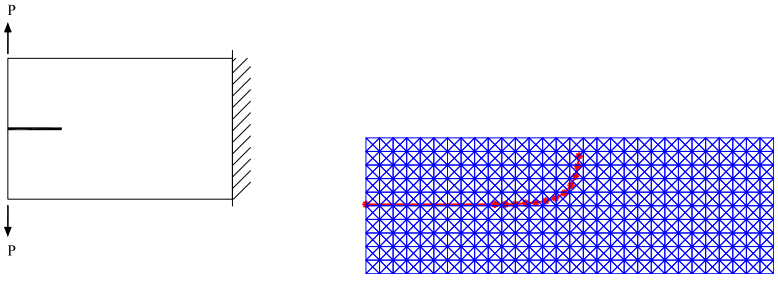


Figure 7. *Time taken respectively for the mesh-generation step and the enrichment step, using various searching algorithm. There are 1000 cracks with a kink (each crack having a length of approximately 12% of the sample characteristic length). The* $n\log(n)$ *behavior of meshing algorithm is highlighted. The* trivial algorithm *uses no mesher provided information, it is an* $O(en)$ *algorithm. The* improved naive algorthim *is a naive approach using the mesher provided facilities – the search method, but not the neighbourhood relationships; the* $e\log(n)$ *behavior is already an improvement over the "obvious" method. The* optimal algorithm *shows the purely* $\log(n)$ *behavior of a search algorithm using all the facilities*

In references (Nguyen, 2004; Bordas *et al.*, 2005b; Bordas *et al.*, n.d.c), we present 2d crack growth results and parameter studies of the XFEM. State-of-the-art 3d crack growth simulation in complex three-dimensional components are presented in other papers (Bordas *et al.*, n.d.b; Bordas *et al.*, n.d.a). We assume plane strain conditions and linear elements. Domain interaction integrals are used to compute the stress intensity factors (Shih *et al.*, 1986; Nikishkov *et al.*, 1987; Moran *et al.*, 1987). Quasi-static crack growth is governed by the maximum hoop stress criterion, and the crack growth increment is chosen in advance and remains constant.

9.2. *Double cantilever beam*

In Figure 8a, a 6×2 double cantilever beam (DCB) is illustrated. The initial crack with length of $a = 2.05$ is placed slightly above the mid-plane of the beam. The material properties are $E = 100$, and $\nu = 0.3$, and $P = 1$. In Figure 8b, for a growth increment $\Delta a = 0.15$, a typical crack path is shown. The mesh had $1,200$ elements. The SIFs are computed with domain size taken to be of $r_d = 2.5 h_{local}$[4]. The result agrees with other published works (Belytschko *et al.*, 1999).



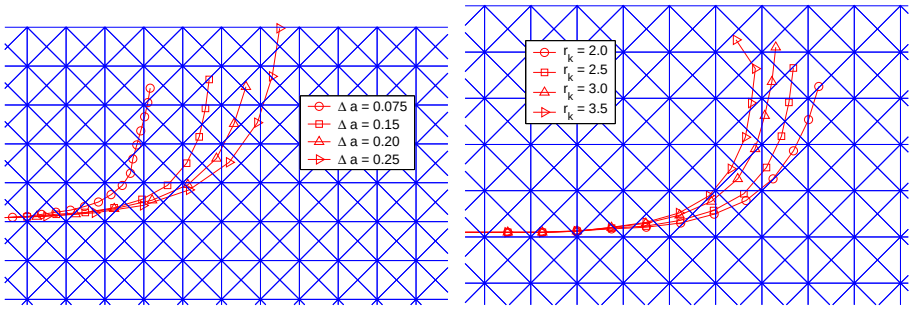
(a) (b)

Figure 8. *Geometry-loads of a double cantilever beam and crack paths specimen*

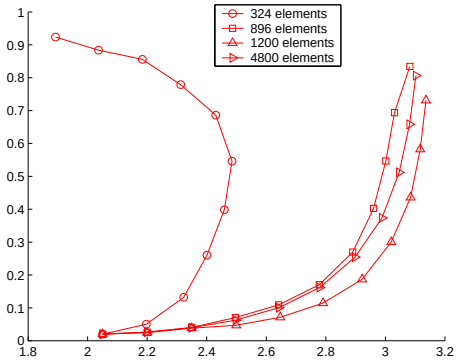
Parameters which affect the accuracy of the simulated crack path are (1) mesh refinement; (2) the domain size $r_d = r_k h_{local}$; and (3) the crack growth increment Δa. A series of computations were performed to study these effects. The numerical results are illustrated in Figure 9. In Figure 9a, crack growth paths obtained with four different crack growth increments are presented (1,200 elements, tip element size $h_{local} = 0.1$). It is observed that a converged crack path is obtained for crack increments between $h_{local}/2$ and $3 \times h_{local}/2$. Accuracy is improved by using smaller Δa's. However, if the crack increment is too small compared to the element size,

4. h_{local} being the square root of the area of the element holding the tip.

multiple changes in the direction of the crack path may occur within an element. In addition, the element partitioning for numerical integration also becomes time consuming. Also, the J integral is not path-independent for curved cracks. Therefore, the domain size r_d also affects the simulated crack path. This effect is shown in Figure 9b. Although the effect of the domain size is small, the adoption of appropriate path-independent integrals for curved cracks would certainly improve the crack growth capabilities of the XFEM. Results on mesh refinement are shown in Figure 9 and show that, for a sufficiently refined mesh, the crack path is not mesh-sensitive.



(a) (b)



(c)

Figure 9. *Parametric study of simulated crack paths in double cantilever beam specimen: (a) influence of crack growth increment Δa (1200 elements, $r_k = 2.5$); (b) influence of domain radius (1200 elements, $\Delta a = 0.15$); and (c) influence of mesh refinement ($r_k = 2.5$, $\Delta a = 0.15$)*

10. Conclusion

Using object oriented approaches to design and implement finite element code allows us to have both the advantages of integration and modularity. The exposure of functionality and data through complete API designed to do more than simply make the transfer of the minimal amount of data possible bear the promise of very large efficiency gains, more than offsetting the use of higher-level programming paradigms. The complete design of the present XFEM code shall be presented in subsequent papers. The prospect of solving for the growth of hundreds of cracks is an exciting outcome of the present work, which shall be presented in a forthcoming paper.

Some phenomena, such as the Alkali-Aggregate-Reaction (Poyet, 2003; Larive, 1998) in concrete are coupled problems: cracks are initiated through a chemical process and are propagated by a mixed chemical diffusion-fatigue process. The interaction between the various zones is easily computed through the integrated meshing facilities, and allows the user to concentrate on the more delicate issues of modelling.

Moreover, we are working on extending our codebase to allow for extremely complex problems, with thousands of inclusions, voids and fractures propagating in viscoelastic media. XFEM should also allow us to integrate external effects, such as chemical (Alkali-Silica Reaction) and thermal effects easily.

11. References

Areias Pedro M., Belytschko T., " Analysis of three-Dimensional Crack initiation and propagation using the extended Finite Element Method", *International Journal for Numerical Methods in Engineering*, vol. 0, n 63, p. 760-788, 3, 2005.

Babuška I., Melenk I., " Partition of unity method", *International Journal for Numerical Methods in Engineering*, vol. 40, n 4, p. 727-758, 1997.

Belytschko T., Black T., " Elastic Crack Growth in Finite Elements With Minimal Remeshing", *International Journal for Numerical Methods in Engineering*, vol. 45, n 5, p. 601-620, 1999.

Bordas S., Extended Finite Element and level set methods with applications to growth of cracks and biofilms, PhD thesis, Northwestern University, December, 2003.

Bordas S., Conley J. G., Moran B., Gray J., Nichols E., " Design paradigm for castings integrating non-destructive evaluation, casting and damage tolerance simulations", *Engineering with Computers*, n.d.a. submitted.

Bordas S., Duddu R., Moran B., Chopp D. L., " Extended Finite Element Method for biofilm growth", *International Journal for Numerical Methods in Engineering*, 2005a. submitted.

Bordas S., Legay A., " Enriched finite element short course: class notes", *The extended finite element method, a new approach to numerical analysis in mechanics: course notes*, Organized by S. Bordas and A. Legay through the EPFL school of continuing education, Lausanne, Switzerland, 12, 2005b.

Bordas S., Moran B., " Extended finite element and level set method for damage tolerance assessment of complex structures: an object-oriented approach", *Engineering Fracture Mechanics*, n.d.b. in press.

Bordas S., Nguyen V. P., Dunant C., Nguyen-Dang H., Guidoum A., " An extended finite element library", *International Journal for Numerical Methods in Engineering*, n.d.c. submitted.

Bordas S., Phong L., Duflot M., " A-posteriori error estimation for enriched finite element methods", *International Journal for Numerical Methods in Engineering*, n.d.d. in progress.

Budyn E., Multiple Crack Growth by the eXtended Finite Element Method, PhD thesis, Northwestern University, June, 2004.

Chessa J., Smolinski P., Belytschko T., " The Extended Finite Element Method (X-FEM) for Solidification Problems", *International Journal for Numerical Methods in Engineering*, vol. 53, p. 1957-1977, 2002.

Chiyokura H., Kimura F., " Design Of Solids With Free-Form Surfaces", *Computer Graphics*, vol. 3, n 3, p. 289-298, 7, 1983.

Devillers O., Meiser S., Teillaud M., " Fully dynamic Delaunay triangulation in logarithmic expected time per operation", *Comput. Geom. Theory Appl.*, vol. 2, n 2, p. 55-80, 1992.

Frederickson G. N., " Optimal Algorithms for Tree Partitioning", p. 169-177, 1991.

Laborde P., Pommier J., Renard Y., Salaun M., " High order extended finite element method for cracked domains", *International Journal for Numerical Methods in Engineering*, vol. 190, n 47, p. 6183-6200, 2004.

Larive C., *Moyens et Méthodes d'Essai*, études et recherches des laboratoires des ponts et chaussées edn, 1998.

Legrain G., Moës N., Verron E., " Stress analysis around the crack tips in finite strain problems using the extended finite element method", *International Journal for Numerical Methods in Engineering*, vol. 63, p. 290-314, 2005.

Mackerle J., " Object-oriented techniques in FEM and BEM, a bibliography (1996-1999)", *Finite Elements in Analysis and Design*, vol. 36, p. 189-196, 2000.

Magnenat S., " Enki", On The Web `http://teem.epfl.ch`, 2000.

Magnenat S., Charrière L.-O., " Globulation 2", , On The Web `http://glob2.ysagoon.com`, 2000.

Melenk J. M., Babuška I., " The Partition of Unity Finite Element Method: Basic Theory and Applications", *Computer Methods in Applied Mechanics and Engineering*, vol. 139, p. 289-314, 1996.

Moës N., Dolbow J., Belytschko T., " A Finite Element Method for Crack Growth Without Remeshing", *International Journal for Numerical Methods in Engineering*, vol. 46, n 1, p. 131-150, 1999.

Moran B., Shih C. F., " Crack tip and associated domain integrals from momentum and energy balance", *Engineering Fracture Mechanics*, vol. 127, p. 615-642, 1987.

Nguyen V. P., *" An object oriented approach to the X-FEM with applications to fracture mechanics"*, Master's thesis, EMMC, 11, 2004.

Nikishkov G. P., Atluri S. N., " Calculation of fracture mechanics parameters for an arbitrary three-dimensional crack by the 'equivalent domain integral' method", *International Journ Numer Meth. Engng*, vol. 24, p. 851-867, 1987.

Poyet S., Etude de la dégradation des ouvrages en béton atteints par la réaction alcali-silice : Approche expérimentale et modélisation numérique multi-échelles des dégradations dans

un environnement hydro-chemo-mécanique variable, PhD thesis, Université de Marne-La-Vallée, 2003.

Remacle J.-F., Karamete B. K., Shephard M., " Algorithm oriented mesh database", *Ninth International Meshing Roundtable, Newport Beach, California*, p. 349-359, October, 2000.

Sethian J. A., *Level Set Methods & Fast Marching Methods: Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision, and Materials Science*, Cambridge University Press, Cambridge, UK, 1999.

Shih C. F., Moran B., Nakamura T., " Energy release rate along a three-dimensional crack front in a thermally stressed body", *International Journal of Fracture*, vol. 30, p. 79-102, 1986.

Strouboulis T., Copps K., Babuška I., " The generalized finite element method : An example of its implementation and illustration of its performance", *International Journal for Numerical Methods in Engineering*, vol. 47, n 8, p. 1401-1417, 2000.

Sukumar N., Prévost J.-H., " Modeling Quasi-Static Crack Growth with the Extended Finite Element Method. Part I: Computer Implementation", *International Journal of Solids and Structures*, 8, 2003.

Various, " Pov Ray", On The Web : `http://www.povray.org/`, n.d.

Ventura G., Moran B., Belytschko T., " Dislocations by partition of unity", *International Journal for Numerical Methods in Engineering*, vol. 0, n 62, p. 1463-1487, 1, 2005.

Zimmermann T., Pelerin Y. D., Bomme P., " Object oriented finite element programming: I. Governing principles.", *Computer Methods in Applied Mechanics and Engineering*, vol. 93, n 8, p. 291-303, 1992.