Generate Cohesive Zone Model with Python

Jianming Zeng
Network Team of Rock Damage and Energy Geostorage Simulation
College of Computing- Georgia Tech
jzeng31@gatech.edu

Abstract

Creating cohesive zone model(CZM) from an finite element model(FEM) can be extremely difficult by hand. The difficulty of making a CZM increases with the number of the mesh elements in FEM. Continue on previous study and implementation, I develop an algorithm specifically for creating multi-crack CZM. This algorithm takes in FEM and produce an CZM with cohesive zone elements along the fracture pattern. This algorithm is consistent with any shape multi-crack model. Though there are some minor adjustments to the code, the overall result is satisfactory.

1. Background

Cohesive zone model is an model in fracture mechanics in which fracture takes place across an extended crack tip, cohesive zone. The naive way to generate a cohesive zone model is to create cohesive elements in between elements in a FEM. The better approach is to write a script in a languages that generates automatically. In this study, I chose python to do the text manipulation. It has shown great promise in efficiency if implemented correctly. Current design and implementation maintains a linear time in any of the algorithms that generate CZM from FEM. As of now, the python script generates any homogenous, inclusion, and multi-crack CZM from FEM, shown in figure 1.

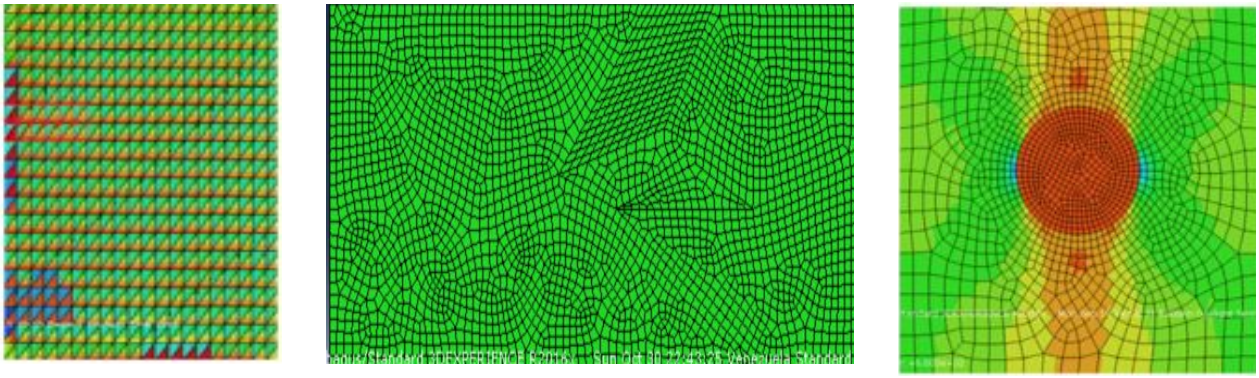


Figure 1
From left to right homogeneou, multi-crack, and inclusion CZM

Unlike the homogenous model or the inclusion model, multi-crack model doesn't necessarily have neat meshes. At every point where fracture is divided, the shape of these element and their neighbours are unpredictable. Many factors can alter the number of cohesive elements needed at the split point as shown in

figure 2 left. Therefore, there are many scenarios to consider before inserting cohesive element. The new algorithm mainly consists three steps. The first step is to read through the data file and establish a tree data structure to store useful information. This tree represents the geometry of the fracture in the form of data. The second step is to generate cohesive elements except at the split point. And the last step is to join all the single cracks from step 2 to form its original shape. The final result of the algorithm is shown on figure 2 right.

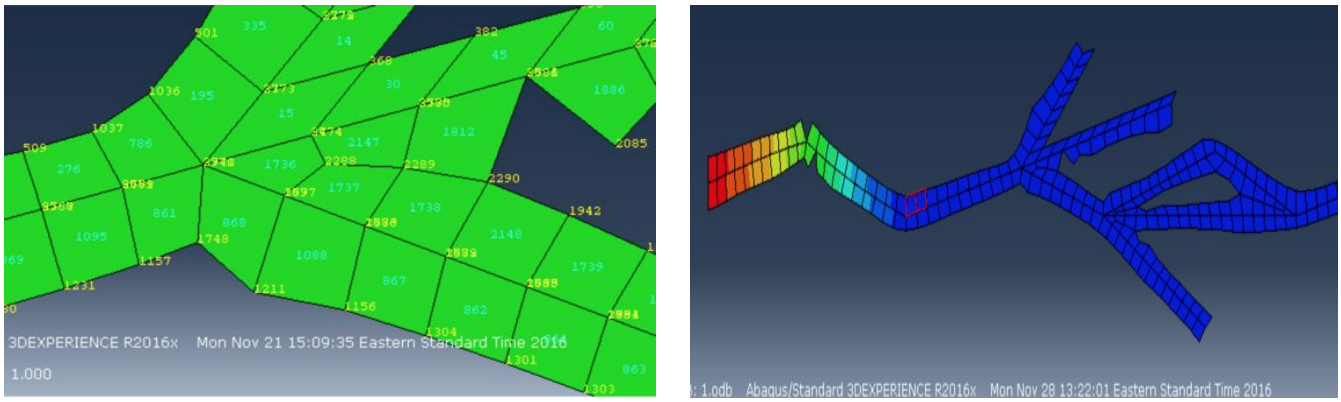


Figure 2
A split point at multi-crack FEM and result of the new algorithm

2. Objective

In this paper, I will first introduce all the theories and data structure used in this new developed algorithms and some easy examples solved in previous study and implementation. These examples are actual correct results and they come from previous implementation. Then I will explain in detail the difficulty I encountered during the actual implementation of the multi-crack CZM algorithm, and also a detail tracing on critical step.

To summarize the procedure in this new algorithm:

1. Establish a tree structure to store geometry information about the fracture
2. Repeatable break the tree into smaller trunks (Divide and Conquer Algorithm)
3. Join smaller trunks of tree back together and form the original shape

3. Theory

3.1 Time and Space Complexity

Correctness and efficiency are the two key factors to be considered when implementing, and efficiency should be considered before correctness. Why? For example, there are many ways to calculate the distance between two points. But as we all know the shortest distance between them is always the Euclidean one. There is always a upper time limit on how fast a program could be. And therefore, one should always aims for the best complexity during the implementation process.

Figure 3 describes most of the time complexity in computation. Letter “n” stands for the size of the data or input. Depends on the type of calculation process and input size, calculation time could have a significant difference. Though the input size is not under control, calculation process is something we could make a difference. In a file I/O problem, the best time complexity one could achieve is linear. The reason is because retrieving information from an input file and write information to a file line by line takes as many time as the number of lines inside a file. In other words, if there are 10 lines information within a file we want,

the best we could do in this case is to read line by line, or 10 lines, so that we get information from every line. And therefore, if we do nothing, just reading information from file and put it back to the file, it takes O(n) + O(n), or O(n). That is the upper limit performance we could achieve in this problem, and that is the optimization we should aim for when start implementing.

Name	Complexity class	Running time ($T(n)$)	Examples of running times	Example algorithms
constant time		$O(1)$	10	Determining if an integer (represented in binary) is even or odd
inverse Ackermann time		$O(\alpha(n))$		Amortized time per operation using a disjoint set
iterated logarithmic time		$O(\log^* n)$		Distributed coloring of cycles
log-logarithmic		$O(\log \log n)$		Amortized time per operation using a bounded priority queue[2]
logarithmic time	DLOGTIME	$O(\log n)$	$\log n$, $\log(n^2)$	Binary search
polylogarithmic time		$\text{poly}(\log n)$	$(\log n)^2$	
fractional power		$O(n^c)$ where $0 < c < 1$	$n^{1/2}$, $n^{2/3}$	Searching in a kd-tree
linear time		$O(n)$	n	Finding the smallest or largest item in an unsorted array
"n log star n" time		$O(n \log^* n)$		Seidel's polygon triangulation algorithm.
linearithmic time		$O(n \log n)$	$n \log n$, $\log n!$	Fastest possible comparison sort
quadratic time		$O(n^2)$	n^2	Bubble sort; Insertion sort; Direct convolution
cubic time		$O(n^3)$	n^3	Naive multiplication of two $n \times n$ matrices. Calculating partial correlation.
polynomial time	P	$2^{O(\log n)} = \text{poly}(n)$	n, $n \log n$, n^{10}	Karmarkar's algorithm for linear programming; AKS primality test
quasi-polynomial time	QP	$2^{\text{poly}(\log n)}$	$n^{\log \log n}$, $n^{\log n}$	Best-known $O(\log^2 n)$-approximation algorithm for the directed Steiner tree problem.
sub-exponential time (first definition)	SUBEXP	$O(2^{n^\varepsilon})$ for all $\varepsilon > 0$	$O(2^{\log n^{\log \log n}})$	Assuming complexity theoretic conjectures, BPP is contained in SUBEXP.[3]
sub-exponential time (second definition)		$2^{o(n)}$	$2^{n^{1/3}}$	Best-known algorithm for integer factorization and graph isomorphism
exponential time (with linear exponent)	E	$2^{O(n)}$	1.1^n, 10^n	Solving the traveling salesman problem using dynamic programming
exponential time	EXPTIME	$2^{\text{poly}(n)}$	2^n, 2^{n^2}	Solving matrix chain multiplication via brute-force search
factorial time		$O(n!)$	$n!$	Solving the traveling salesman problem via brute-force search
double exponential time	2-EXPTIME	$2^{2^{\text{poly}(n)}}$	2^{2^n}	Deciding the truth of a given statement in Presburger arithmetic

Figure 3
Most seen time complexity

3.2 Graph Theory

Another useful theory in this task is graph theory. Graph theory is a study of mathematical structures used to model pairwise relations between objects. A graph in this context is made up of *vertices*, *nodes*, or *points* which are connected by *edges*, *arcs*, or *lines*. A graph may be *undirected*, meaning that there is no distinction between the two vertices associated with each edge, or its edges may be *directed* from one vertex to another. When constructing a FEM, abaqus stores the model information in an inp file. This inp file contains two major parts. The geometry information such as parts, nodes, and elements. And the physical property part controls any other parameters. The cohesive element insertion is mostly geometrical alteration and abaqus inp file doesn't define any pairwise element relationship. Therefore graph theory is the best suit for this task when storing data.

3.3 Divide and Conquer (D&C)

In computer science, divide and conquer is an algorithm design paradigm based on multi-branched recursion. A divide and conquer algorithm works by recursively breaking down a problem into two or more subproblems of the same or related type, until these become simple enough to be solved directly. The solutions to the subproblems are then combined to give a solution to the original problem. With the same idea, the multi-crack model is recursively being splitted into single crack model and then put back together to its original form.

3.4 Object Oriented Programming (OOP)

OOP and data structure offer many options when completing the task. For the better structure, OOP should be used in implementing the addition process of cohesive zone elements. This setup gives a more precise implementation of the graph theory. Node object class has coordinate as local variables. And Element object includes the node's ID that define it. Element Set, Node Set and Cohesive Zone Set objects are optional because it doesn't change the geometry with or with putting nodes or elements into set. So theses objects could be inserted into a list or array for the same purpose. Figure 4 is an example of how list or array performs in term of different operations. Same graphs could be found online for set and dictionary(hashmap). In order to keep time complexity and space complexity as small as possible, one should choose data structure wisely. For example, list or array is one of the best options for storing information in order and set is best for data comparison. After all, list, set and dictionary could build up the best solution for storing data in general.

Operation	Average Case	Amortized Worst Case
Copy	O(n)	O(n)
Append[1]	O(1)	O(1)
Insert	O(n)	O(n)
Get Item	O(1)	O(1)
Set Item	O(1)	O(1)
Delete Item	O(n)	O(n)
Iteration	O(n)	O(n)
Get Slice	O(k)	O(k)
Del Slice	O(n)	O(n)
Set Slice	O(k+n)	O(k+n)
Extend[1]	O(k)	O(k)
Sort	O(n log n)	O(n log n)
Multiply	O(nk)	O(nk)
x in s	O(n)	
min(s), max(s)	O(n)	
Get Length	O(1)	O(1)

3.5 Examples

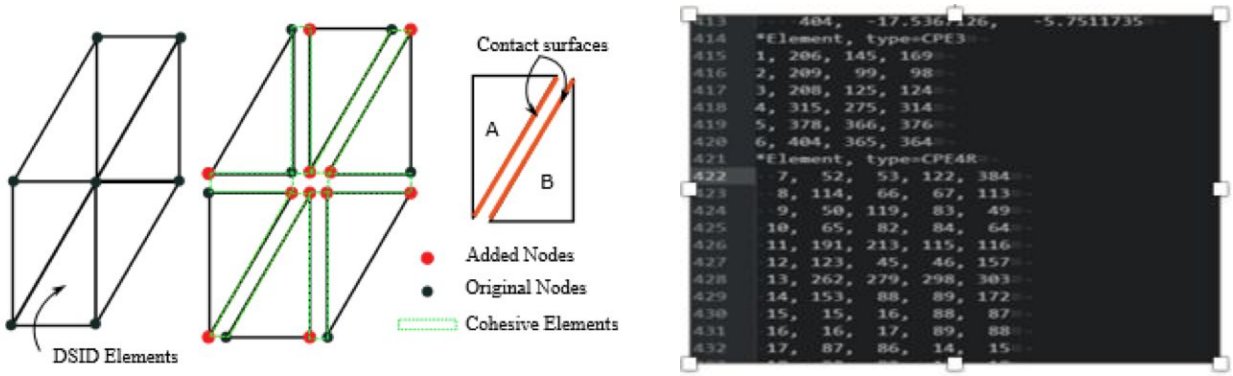


Figure 4
Cohesive element and its data representation

There is a big difference between what we know about FEM and how it is represented in an inp file. In figure 4 left shows a basic structure of how cohesive zone element is formed. This process relies on pieces of information. First, we need to know the location to insert cohesive zone element. This location is always on the contact face between two elements, or on the shared edge. Second, every time a new cohesive zone element is created, there will be at least one new additional node associated with the new cohesive zone element. Last and most important, after updated correctly we also need to run a check on all the elements to make sure that stay in contact. Shown in figure 5, we can see if any of the node doesn't follow the correct element even only one pair of nodes are flipped, the model will result in

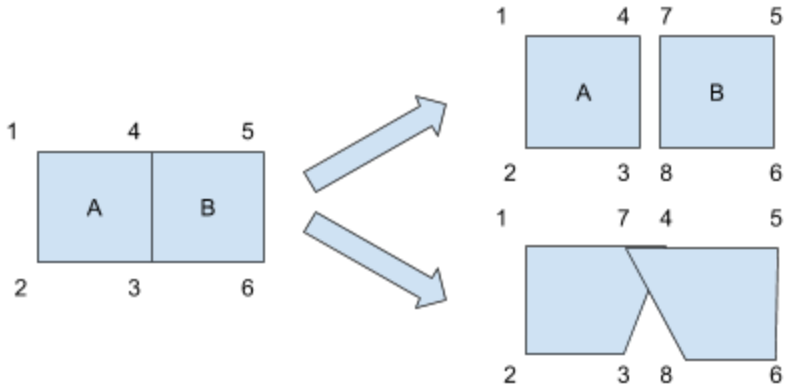


Figure 5
Correct update and fasle update

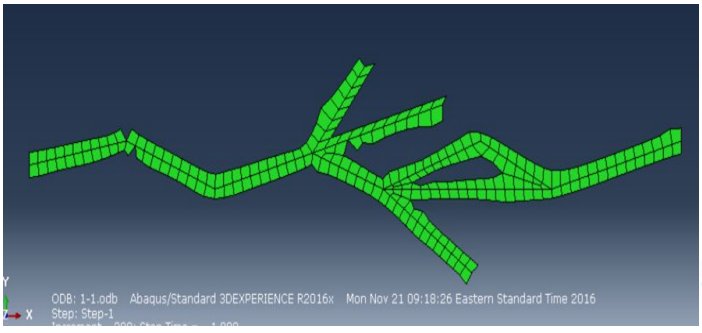
element overlap and intersection, for which will cause an the entire model to fail during execution.

4. Algorithm Explanation

For multi-crack model, the algorithm mainly consist three major parts. Unlike any models seen before, multi-crack model has a lot more uncertainty due to its unpredictable mesh shape. Therefore we need some more extra relationship information.

4.1 Tree

Just by looking at the two graphs of figure 6, we can see they have a lot in common. The reason of using ADT tree is beyond their physical similarity. In an ADT tree, every node in tree has information about its predecessors and successors, and how deep it is in the tree as shown in figure 6.



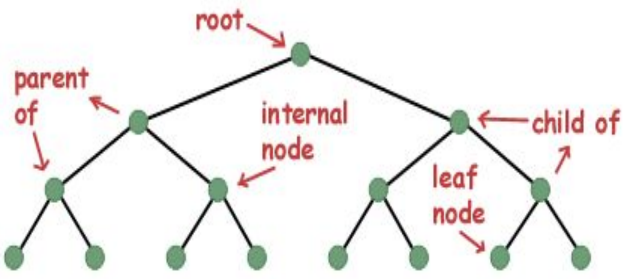


Figure 6
Geometry along fractures and ADT tree

Firstly, unlike the type of cohesive element shown in figure 4, elements along a crack are generally categorized into two sides. As stated before, a cohesive zone element requires at least one update node. On one side, all elements stay the same. And on the other side, elements are update partially, as shown in figure 7. The upper half of the graph represents the side that stays the same. So nothing is needed to be done. All the elements from the lower half need new copies of node to replace the original ones. Having elements categorized in first hand and update them based on their membership will result in less work. Tree is helpful in the case, because tree not only stores data but also it maintains an order between nodes. When following the path down the tree, we also have a reading order such that we could use it for categorizing which set the elements belong to. And in my implementation, the set to the left of the reading discretion is the set needed to be updated.

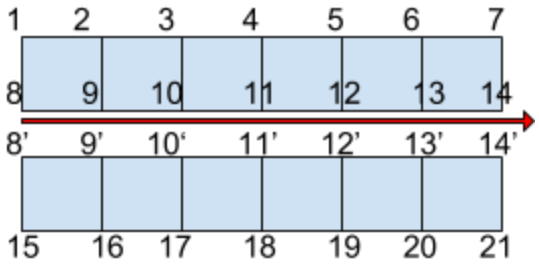


Figure 7
Single Crack CZM

Secondly, whenever fracture splits, it is reflected on the tree as well. In usual case, any node has one and only one child. But if a node has more than one child, that tells us the current node we are on is a split point. With this piece of information we know when to treat the current node with extra care.

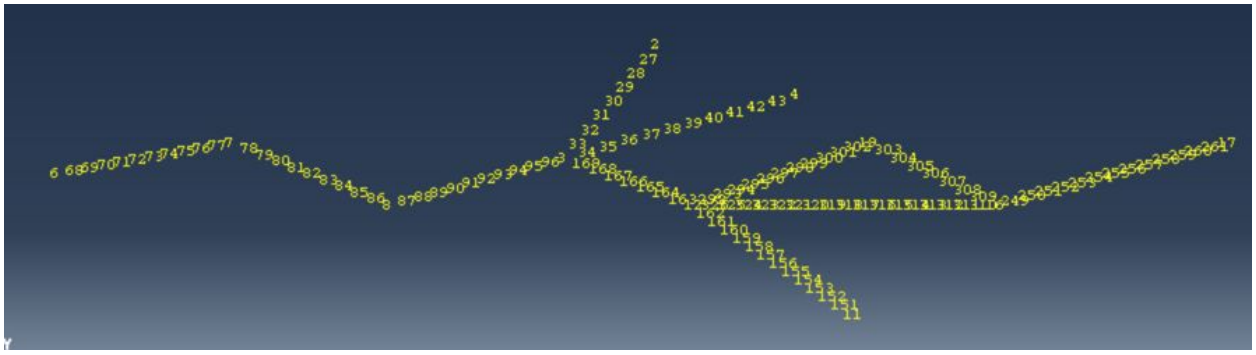
graphNode.py

```
class graphNode:
    ID = None
    neighbours = None
    number = 0

    def __init__(self, ID):
        self.ID = int(ID)
        self.neighbours = []
        self.number = 0

    def addNeighbour(self, node):
        self.neighbours += [node]
        self.number += 1
```

Graph node data structure used for constructing tree
(continued on next page)



Tree(with nodes only)
Figure 8

4.2 Divide

Up until split point, update is relatively easy as only one side of the elements needed to updated and the reading direction tells us exactly which side it is. Things become very complicated at the split point. There are a lot of situation needed to be considered. And therefore, in this section the algorithm leaves out the complicated part and continues on working the easy ones. There are two reasons behind this design. First, multi-crack model is a union of a series of smaller single crack models. The update procedure follows that in single crack model except at the split point. Second, this design rules out some of the uncertainties at the split point at the same time. Later when we come back to the split point we have fewer scenarios to consider.

The algorithm takes the tree from previous step and loop through every layers. There are only two cases we worry:

1. The current node has one child. Continue grow the current single crack.
2. The current node has more than one child. Record current node and skip the current one and continue on the next one, starting a new single crack.

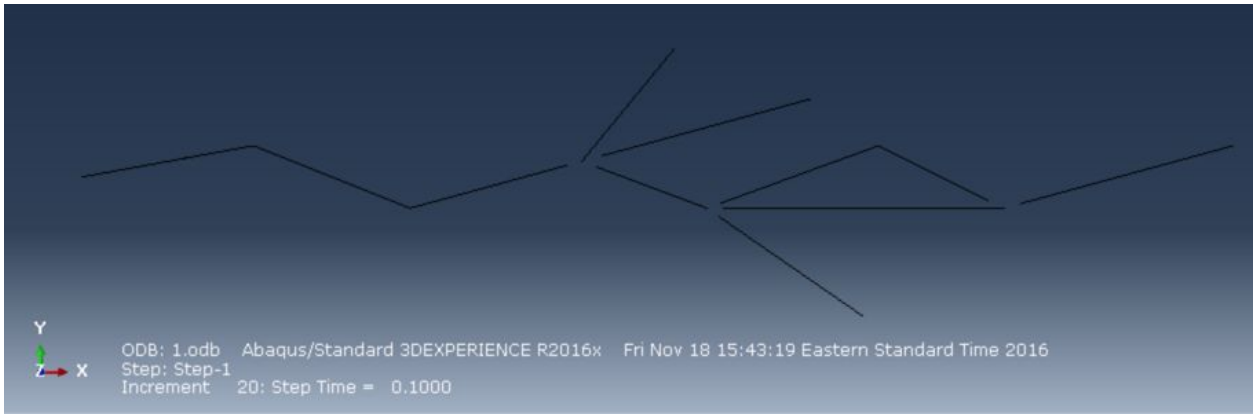


Figure 9
Many single cracks(with cohesive zone elements)

4.3 Merge

At this point, the algorithm has successfully generates most of the cohesive zone elements needed and the task left is to join these single cracks back to its original form. To do so, the algorithm starts with all elements around the recorded node from previous step. There are generally steps:

1. Update peripheral nodes according to its neighbours
2. If element is on the boundary, update the other node according to its neighbour
3. Or if element is surrounded by others, give it a new copy of node

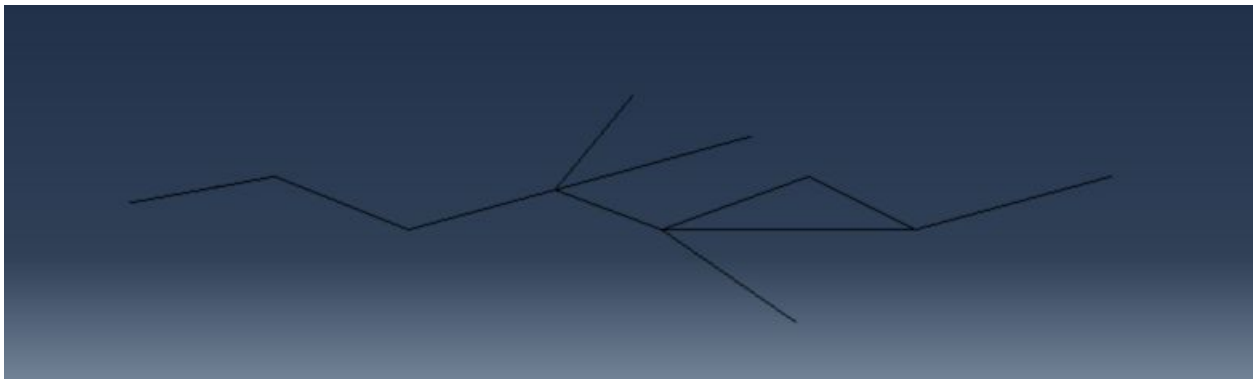


Figure 10
Restored to original form (all cohesive zone elements present)

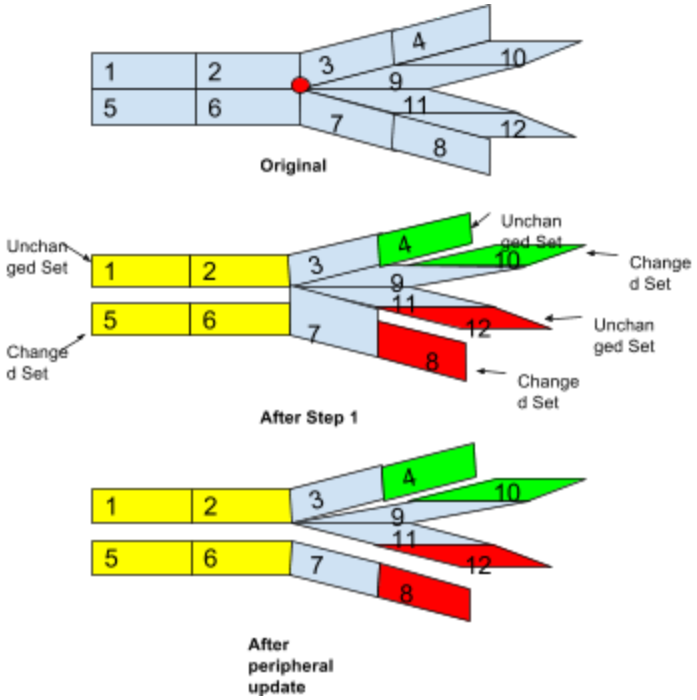


Figure 11
Demonstration

4.4 demonstration on 1-3 split

This section demonstrates the behavior of the algorithm in a 1-3 split model. There are 12 elements in the first diagram of figure 11 and split happens in between elements 2, 3, 6, 7, 9, and 11, shown by the red dot. Shortly after the algorithm divides the crack into smaller ones, the result looks as shown in the middle of figure 11. Different color represents different single cracks and they are updated correctly so there is a white space in between which denotes cohesive zone. Because of the different membership of changed or unchanged set, element 7 and 9 are twisted because they still attach to the original copies of node. Since we know every single crack(shown in color) is updated correctly, the algorithm relies on that information and update node according to the neighbour of current element. Shortly after updating the peripheral nodes based on their neighbours, we see in the bottom diagram of figure 11, element 9 and 10, element 6 and 7 and element 7 and 8 are attached to each other correctly. Finally, the algorithm determines whether the current element is on the boundary(element 3) or surrounded by others(element 9 and 11). Element on the boundary is updated firsts based on its other neighbour, then elements within boundary gets new copies of nodes.

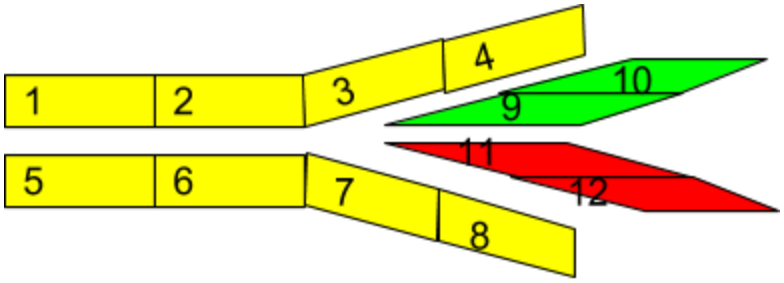


Figure 12
Final product

5. Result

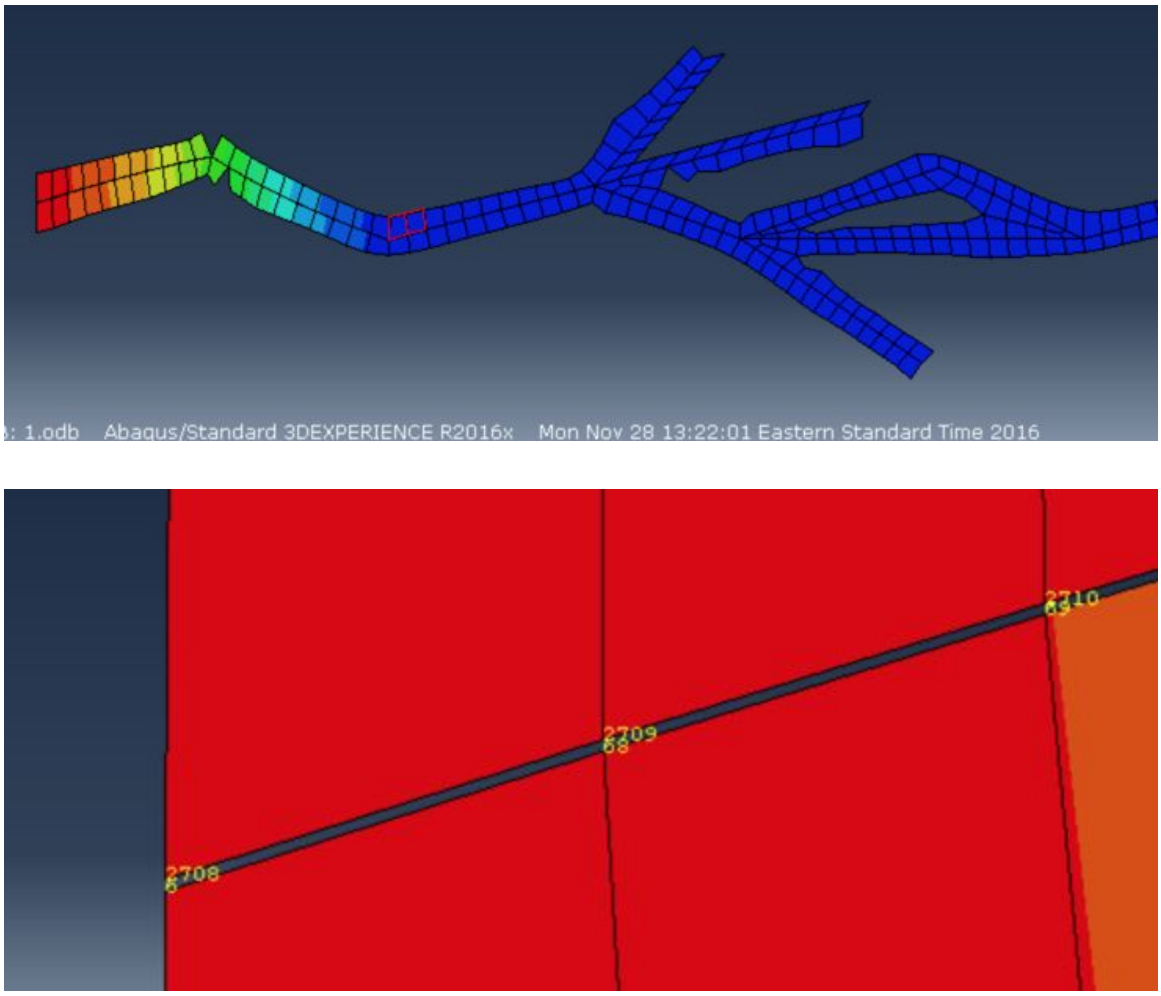


Figure 13
Overview and detail view

Cohesive zone elements are all correctly inserted and neighbour elements are also correctly attached, as shown in graph 13. Clearly, the elements on the upper half of the detail view diagram belong to the changed set. A new copy of node 6, which is 2708, is assigned to the upper element. However, there are still some minor adjustments to make on parameters. there are high pressure on the tip of the crack but not enough for it the break. Either we need greater force load on the model, or likely wrong input parameters such that they prevent the fracture propagation. Either way the final result of the benchmark with go to next semester report.

6. Conclusion

This algorithm has once more reveals the great success of using python to build CZM based on FEM. Once a script is finished, it's fast and efficient compared to manual. Until now, the algorithm takes less than a second to calculate any model with less than a 10000 node size or a 10000 element size. Combining previous work, the program now handles any CPE4 homogenous, inclusion, and multi-crack model.

Divide and conquer is a new technique introduced for multi-crack model specifically. This technique helps recursively break a difficult multi-crack model into smaller, easier-to-solve single crack models first, and merge them back together. Along this process many scenarios are ruled out and what left over are much easier cases. Like graph theory and OOP, computer science has shown so much power in help shaping this algorithm efficient and fast.

7. Future Research Plan

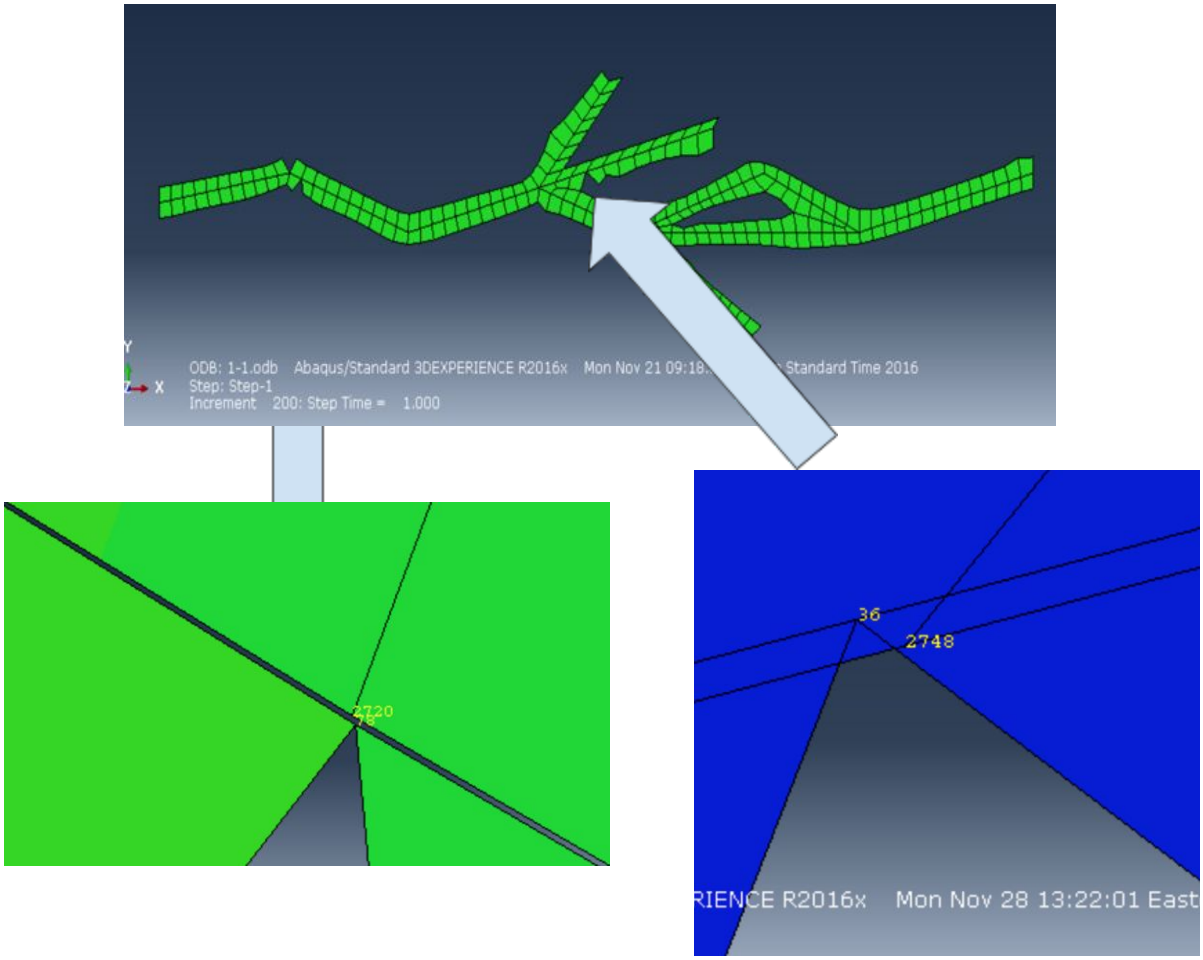


Figure 14
Fraud on current script

The first thing to do next semester is to fix a fraud in the script. As shown in figure 14, there are two place where a node is shared by five elements rather than four in most places along crack. This is due to a random mesh shape when building FEM. If lucky, when the missing piece and the unchanged set are on the same side, as shown in bottom left of figure 14, the CZM still work perfectly. However, there is a possibility that they are not and the result is bottom right of figure 14. The missing element has the original copy of node while its neighbours have the new copies. And therefore we see an overlap and intersection between these elements.

Next, I am moving onto 3D CZM as the team is more interested in a more realistic model. It's imaginable 3D is much harder compared to 2D. In any 2D model, an edge is always shared by two elements. That is not the case in 3D and hence there will be more scenarios to discuss. Though it's challenging, it's also very fun to practice theories and data structures.

Finally, I will package all the source code online for future collaboration. Up to now there are over two thousand lines of code and hence a lot of commenting to do. Hopefully by the end of next semester I have all the documentation ready and a user guide or manual.

8. References

[1] Jin, W., Kim, K. & Wang, P. (2015). Hydraulic-Mechanical Analysis of Damaged Shale Around Horizontal Borehole (02).

[2] Complexity, Time. "Page." TimeComplexity. Python, 15 June 2015. Web. 21 Apr. 2016.

[3] "Slime Mold Grows Network Just Like Tokyo Rail System." Wired.com. Conde Nast Digital, n.d. Web. 21 Apr. 2016.

[4] "Mapping Tokyo's Train System in Slime Mold." CityLab. N.p., n.d. Web. 21 Apr. 2016.

[5] "Time Complexity." Wikipedia. Wikimedia Foundation, n.d. Web. 21 Apr. 2016.

[6] "Abaqus XFEM Capability." Abaqus XFEM Modelling of Concrete Crack. N.p., n.d. Web. 21 Apr. 2016.

[7] "Divide and Conquer Algorithms." *Wikipedia*. Wikimedia Foundation, n.d. Web. 28 Nov. 2016.