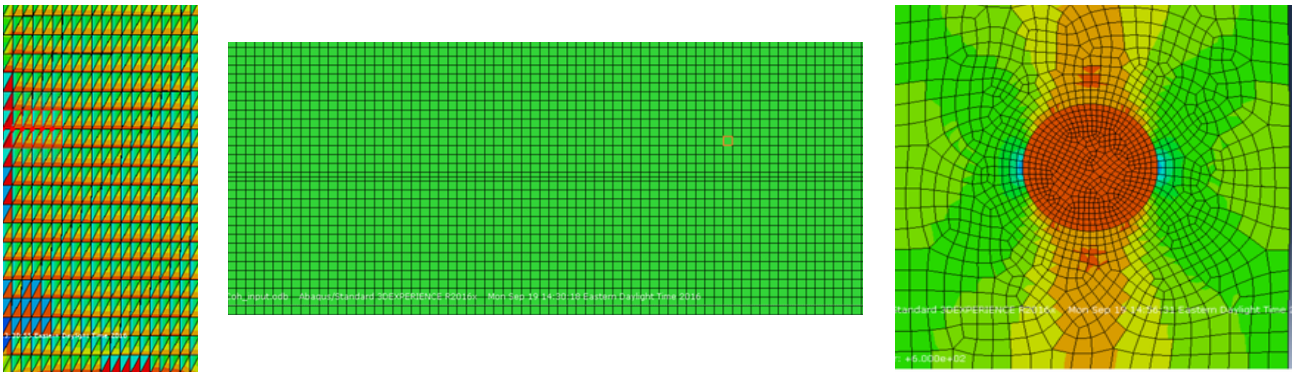
Network Team of Rock Damage Modeling and Energy Geostorage Simulation
Midterm Report: Benchmark Joint Elements, CZM and XFEM

Jianming Zeng

Introduction:

Previous study and implementation has shown great promise on how effectively python could help generating CZM(cohesive zone model). It's a very efficient way to generate relatively large size CZM. With current unique design and implementation, the program is able to generate CZM in linear time as before and has some level of intelligence that handles similar models. As of today, the python program could handle various simple FEM(Finite Element Model) models and turned them into CZM. These simple models include basic, single-crack, and single-inclusion models(from left to right).



Basic model: cohesive elements in between all elements
Single crack: cohesive elements along one specific path
Single inclusion: cohesive elements around some geometry

Though there are some success in creating CZM, the future study is still challenging. The main reason is also caused by the unique design and implementation(Discuss later).

Objective:

The objective of this semester is to continue the previous study on creating a smart program handles as many models as possible. At the same time, the program should be efficient and user-friendly. To summarize the long term and short term goals:

The ultimate goal of this study is that:

1. The program could handle as many models as possible in linear time (Efficiency).
2. The program could be easily adjusted (Reusability).
3. Package the code.
4. Create a manual/guideline .

The short term goals are:

1. Handle multi-crack model correctly.

2. Change data structure (Object-oriented).
3. Describe elements behavior(wrt abaqus) on shared edges.

Theory:

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

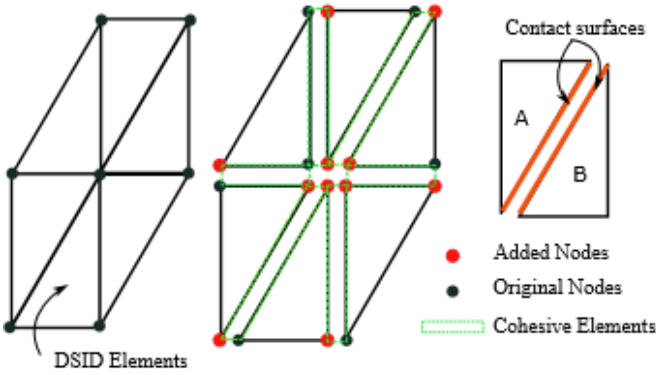
Name	Complexity class	Running time ($T(n)$)	Examples of running times	Example algorithms
constant time		$O(1)$	10	Determining if an integer (represented in binary) is even or odd
inverse Ackermann time		$O(\alpha(n))$		Amortized time per operation using a disjoint set
iterated logarithmic time		$O(\log^* n)$		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

implementation process.

In the above chart, it describes most of the 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.

Another useful theory in this task is graph theory. 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. Considering the follow graph:



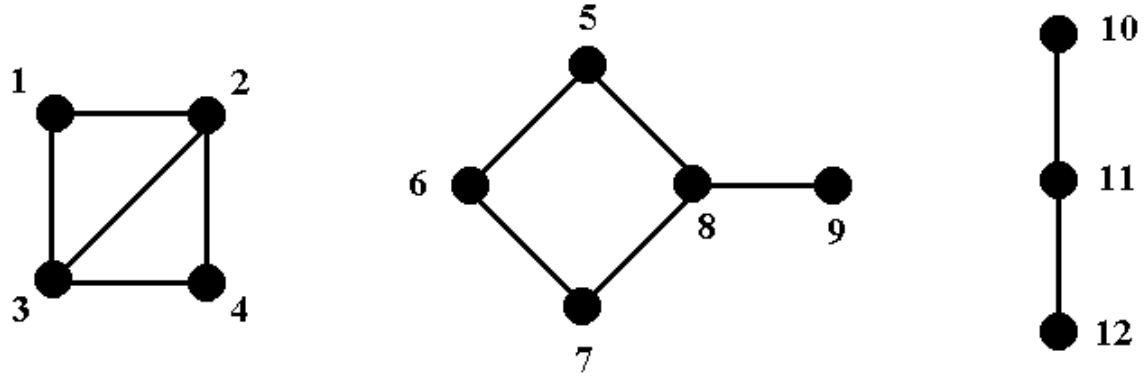
In abaqus, node is given by its coordinate

```
413      404,  -17.5367126,   -5.7511735
414  *Element, type=CPE3
415  1, 206, 145, 169
416  2, 209,  99,  98
417  3, 208, 125, 124
418  4, 315, 275, 314
419  5, 378, 366, 376
420  6, 404, 365, 364
421  *Element, type=CPE4R
422   7,  52,  53, 122, 384
423   8, 114,  66,  67, 113
424   9,  50, 119,  83,  49
425  10,  65,  82,  84,  64
426  11, 191, 213, 115, 116
427  12, 123,  45,  46, 157
428  13, 262, 279, 298, 303
429  14, 153,  88,  89, 172
430  15,  15,  16,  88,  87
431  16,  16,  17,  89,  88
432  17,  87,  86,  14,  15
```

and element is made of the nodes ID that define it(example to the above right). This tells very little information where to put cohesive zone elements, and hence searching for the right place to put cohesive zone can be very expensive. A naive approach is described in the pseudo:

```
For element in elements:
    For node1 in element:
        For node2 in element and node2 != node1:
            Edge = (node1, node2)
            If edge has neighbour:
                Add cohesive elements
```

While this is only a simple version, we can tell is a very bad implementation. For loop is very costly in term of time optimization. This naive approach constantly visits the same information over and over again. With small data size the complexity may still look considerably good. But time complexity grows exponentially in this model. One significant improvement is described by graph theory. Like the following sketch, we could establish some sort of relationship between



nodes and elements. This implementation could decrease the number of revisit time to achieve time optimization. In other words, with additional information, we could use a lot less for loop. A possible pseudo as followed:

```
For edge in graph:
    If elements[edge] contains two edges:
        Add cohesive elements:
```

The actual implementation is more complicated but this set-up only requires one for loop and therefore O(n).

Object Oriented Programming(OOP) and Data Structure:

OOP and data structure offer many options when completing the task. For the sake of 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. The graph to the left 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.

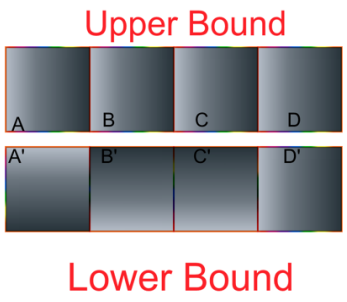
Operation	Average Case	Amortized Worst Case
Copy	O(n)	O(n)
Append[1]	O(1)	O(1)
Insert	O(n)	O(n)
Get Item	O(1)	O(1)
Set Item	O(1)	O(1)
Delete Item	O(n)	O(n)
Iteration	O(n)	O(n)
Get Slice	O(k)	O(k)
Del Slice	O(n)	O(n)
Set Slice	O(k+n)	O(k+n)
Extend[1]	O(k)	O(k)
Sort	O(n log n)	O(n log n)
Multiply	O(nk)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

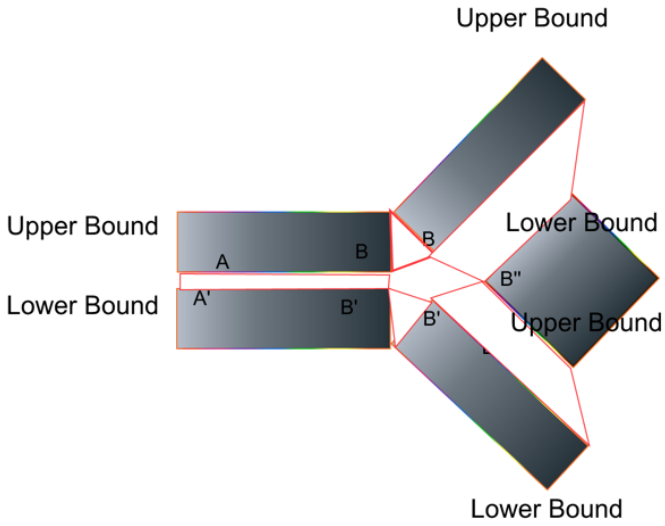
Challenge:

```
>>>
File Name: example
Output Name:example output
CZE set:the target node or element set
CZM material name:the type of material
1. CZM-Homogenous
2. CZM-Single Inclusion
3. CZM-Single Crack
Choose a function, input the integer:
```

The program could handle three major simple CPE4 models and any variation of them correctly and efficiently, as listed above. And the program has shown great success in practicing the theories and data structures. But it's becoming more difficult to extend its functionality to new models. Homogenous program is hard to achieve.

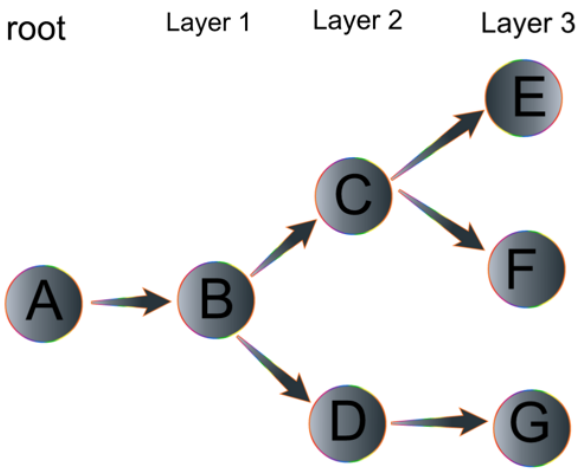
One problem I am facing is the reactive coding approach. Every time I am provided a new model, I made some changes to the program in order to fit the need. There are both advantages and disadvantages with this approach. The good thing is that is't guaranteed to handle everything we've seen so far correctly. The downside is that the program will always break when unintended geometry is encountered. And now the downside has becoming a huge problem.





In single crack CZM, I've separated the boundary(crack pattern) into upper and lower so that i only make one new copy for every vertices(left). However, in multi crack CZM, there's disjunction like B(right) where fracture splited into 2 sub fractures. If using the same method as in single crack, the program isn't able to correctly identify the proper behavior of the middle elements. This is a result of reactive programing and there isn't a good solution with current implementation.

An alteration is to use a tree structure, where nodes on the fracture are tree nodes. Every node has information about its parent and it children. This setup looks like the graph below.



The advantage of storing data in a tree is that for every node, it knows where it came from, so we could construct cze, and knows how many children it has, the number of sub cracks. This implementation avoids defining upper or lower bound. Most importantly, we know how many sub cracks there are so that we update vertices correctly in real time. So far this is only in theory. I haven't tested the correctness of this set up.

Pseudo:

```
For level in tree:
    For node in level:
        Parent = node.parent
        Edge = (Parent,  node):
        Add cohesive zone elements
        Update original elements
        Check for duplication at crack tip
```

Research Plan:

There are two urgent adjustments to the program. First, previously I gave up OOP in hope of a better performance when dealing with new models' information. It didn't turn out very well as every time I am given a new model I would have to make a few adjustments eventually. And OOP is the key factor in implementing the tree structure. After changing back to OOP, I will implement the tree to represent the crack. In theory, it is a very representation but the actual performance could be slightly different depends on the whether there is constraint I missed.

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.