**DFNgen 2.0 Documentation**

**Fracture Cluster Management**

**Introduction**

This document covers DFNgen 2.0’s cluster group management system and the isolated fracture removal process.

Fracture clusters are used in DFNgen for isolated fracture removal after the DFN has been generated and before DFNgen generates its output files. An isolated fracture is a fracture that does not intersect any other fractures and will not contribute to flow. Fracture clusters are also considered isolated when the cluster does not connect the users defined domain boundary faces.

NOTE: Isolated fracture removal only removes fractures with no intersections when the input option *ignoreBoundaryFaces* is set to 1.

Facture cluster data is kept and updated with each new polygon/fracture added to a DFN.

**Algorithm Overview**

In the DFNgen source code, relevant functions are:

1. *intersectionChecking*(), found in computationalGeometry.cpp

2. *assignGroup*(), found in clusterGroups.cpp

3. *updateGroups*(), found in clusterGroups.cpp

4. *getCluster*(), found in clusterGroups.cpp

As a new polygon is being tested for intersections and for feature sizes less than *h* (these checks happen one intersection at a time), three lists are maintained:

* 1. Intersected polygons list (variable *tempIntersectList* in *intersectionChecking*()). This list contains indices/pointers to all the polygons which the new polygon has intersected in the order that they occur.
  2. Intersections list (variable *tempIntPts* in *intersectionChecking()).* This list contains all new intersections (*IntPoints* structures) created by the new polygon in the order that they occur.
  3. Encountered cluster groups list (variable *encounteredGroups in intersectionChecking*()). This list contains all other cluster group numbers which the new polygon has intersected with after the new polygon already has been assigned a group number.

E.g. If from the first intersection, the new polygon is assigned to group 5, and the next intersection is with a fracture in group 2, ‘2’ is the first group saved to the encountered groups.

When a polygon bridges more than one group, there will be several different cluster groups to update.

If for any reason the fracture is rejected (FRAM rejects it while checking an intersection for features of size less than *h*), these lists are deleted and the fracture is either re-translated to a new position, or a new fracture is generated. If the fracture is accepted, the data in these lists are used to update the permanent fracture cluster data.

**Code overview:**

1. Go through previously accepted polygons and test for intersections with the new polygon being added to the DFN.

**Once an intersection is found (by function *intersectionChecking()*) and has passed the FRAM tests, several things happen:**

1. The intersection structure for the newest intersection is appended to the temp intersection array *tempIntPts.*
2. The index of the fracture the new polygon intersects with is appended to the intersected polygons list *tempIntersectList.*

1. The index to the new intersection structure’s place in the permanent *intPts* array, if the new polygon is accepted, is calculated and appended to the new polygons list *intersectionIndex*. That is, the index that is saved is the index the intersection will have once moved to the permanent array if it is not rejected.
2. Any triple intersection points are saved to a temporary list of structure *tempData*. This structure contains the triple intersection point, and the index to the place in the permanent *triplePoints* list of where it will go if the polygon is not rejected (similar to step 4).
3. New Polygon Gets a Cluster Group Number (*groupNum* in the *Poly* struct).
   1. If it is the first intersection found, the new polygon inherits the cluster group number of the intersecting polygon.
   2. If the new polygon has already been given a cluster group number from intersecting another fracture), the intersecting polygon’s cluster group number is added to the encountered cluster groups list *encounteredGroups*. This will be used to update the fractures and cluster groups (merging the two groups together) IF the new polygon does not end up being rejected (it still has more polygons to check for intersections with).

**Numbers 2 to 5 repeat until all fractures have been checked for intersections with the new polygon. If the polygon has not been rejected during the process:**

1. If no intersections were found after searching through previously accepted polygons, the new polygon is given a new cluster group number using the *assignGroup*() function (details below).
2. The new polygon is moved to the permanent *acceptedPoly* list.
3. If there were new intersections, they are now appended to the permanent *intPts* list.
4. All intersected polygons will have their *intersectionIndex* list updated with the indices of the new intersections. We do this by adding the index of each new intersection to its corresponding polygon in the same order which they were found. The list for polygons we encountered is in the variable *tempIntersectList*.

E.g. if the permanent *intPts* intersection list already has 10 (indexes 0 - 9) intersections from previous fractures and we just added 3 more fractures and intersections, and each fracture can only intersect with the new polygon once, the indexes to the new intersections once they are moved to the permant *intPts* list will be indexes 10, 11, and 12 (indexes start at 0). So, we append to the first polygon listed in the *tempIntersectList* index 10, the second polygon in the list index 11, and the third index 12.

1. If there are new triple intersection points, they are now appended to the permanent *triplePoints* list. The temporary triple intersection points are held in a list of *TrieplePtTempData* structures. This structure contains the triple intersection point, and the index for each of the intersections it belongs to (three total). One of the intersections will be a new intersection just created by the new polygon, and the other two will be a triple intersection point on previously accepted intersections.

The new triple intersection point is added to the permanent *triplePoints* array, and then its index in that permanent array is appended to the intersection structure variable *triplePointsIdx* for the intersection that it belongs to.

1. The last thing that is done is a call to the function *updateGroups*() (details below).

***Function assignGroup*() overview, defined in clusterGroups.cpp**

The function *assignGroup*() is used to assign a new polygon to a new cluster group. This function is for polygons that do not intersect with any other polygons; otherwise a cluster group will be inherited from the intersected polygon.

Arguments to this function:

1. *Poly* structure reference. A reference to the new polygon being assigned a new group.
2. *Stats* structure reference. The program statistics object (variable name *pstats* throughout the code). The *Stats* structure contains two structures within it that contain all the cluster group information. These structures are *FractureGroups* and *GroupData* (details below).
3. Index (integer) of the new polygons place in the permanent polygon list *acceptedPoly.*

**Code Overview**

**(See sections on *GroupData* and *FractureGroups* structures for their details)**

1. The new polygon is assigned the next available group number. This comes from the *Stats* variable *nextGroupNum.*
2. A *GroupData* structure is created.
3. Inside the *GroupData* structure, there is a boolean array of six elements*.* This array, *faces,* contains connectivity information for the cluster. There is an element for each of the six faces, or walls, of the domain. False meaning it is not touching that face, true meaning it is touching the face (see *GroupData* section for more details). Likewise, there is another *faces* array in the polygon *Poly* structure.

The polygon’s *faces* array and the *GroupData’s faces* array are bitwise ORed together so that anywhere there is a true in the polygons *faces* array, there will be a true in the *GroupData’s faces* array. After many polygons go through this process for a single cluster group, by looking at the *GroupData’s faces* array we are able to see which domain faces the cluster connects.

1. Next, the variable *size* inside of the structure *GroupData* is incremented. This contains the number of fractures contained in the fracture cluster group.
2. The *GroupData* structure is now saved to a permanent location within the *Stats* structure.
3. A *FractureGroups* structure is now created.
4. The new *FractureGroups* structure is assigned the same group number from step 1 using the same *nextGroupNum* variable.
5. *nextGroupNum* is incremented.
6. Inside the *FractrueGroups* structure is the list (*polyList)* of polygons belonging to the group. The index for the location in the permanent polygon list, *acceptedPoly,* for the new polygon is added to this list.
7. The *FractureGroup* structure is then saved to a permanent location within the *Stats* structure.

***Function updateGroups*() overview, defined in clusterGroups.cpp**

The function *updateGroups*() is used to update the fracture cluster group information for new polygons that have intersected other polygons. When updating the cluster group information, there are two cases:

1. The new polygon only intersected with polygons of a single group.
2. The new polygon intersected and connected more than one group. The groups now need to be merged together into a single group.

Arguments to this function:

1. *Poly* structure reference. A reference to the new polygon being added to fracture cluster groups.
2. Permanent list of accepted polygons already in the DFN (variable *acceptedPoly).*
3. List of cluster groups which the new polygon has intersected with, if more than one group (see example in part c on page 1).
4. *Stats* structure reference. The program statistics object (variable name *pstats* throughout the code). The *Stats* structure contains two structures within it that contain all the cluster group information. These structures are *FractureGroups* and *GroupData* (details below).
5. Index (integer) of the new polygons place in the permanent polygon list *acceptedPoly.*

**Case A**

1. The new polygons *faces* data is ORed into its corresponding *GroupData* structure.

The *GroupData* array, (in variable *pstats)* is always aligned with cluster group numbers. Group numbers start at 1, the indexes to the array start at 0. E.g. to access the *GroupData* structure for cluster group 12, it is the variable *pstats.groupData[12 – 1].*

1. The corresponding *GroupData* structure’s variable *size* is incremented (number of polygons in the group).
2. Next, the corresponding *FractureGroup* structure must be found. This has to be done by searching through the array (*pstats.fractGroup)* and comparing the new polygons *groupNum* and the group number in the *FractureGroup* structure.

See below for an explanation as to why we have to search for the group number, and why the *GroupData* and *FractureGroup* structures are not combined a single structure.

1. Once the correct *FractureGroup* structure is found, the index to the new polygon in the permanent polygon list *acceptedPoly* is appended to the list *polyList* in the *FractureGroups* structure.

**Case B**

1. The new polygon’s corresponding *FractureGroup* structure is searched and found. The poly is added to the *FractureGroup* structure (see 3 and 4 in Case A).
2. The new polygon’s *faces* data is ORed into the new polygons corresponding *GroupData* structure (see 1 in Case A).
3. The new polygon’s corresponding *GroupData* structure has it’s *size* incremented (see 2 in case A).

**Merge Cluster Groups**

1. For all groups in the *encounteredGroups* list (see part c under Algorithm Overview at the beginning of this document), the *GroupData’s* *size* variable, is added to and the *GroupData* structure corresponding to the new polygons group number.
2. The *GroupData’s* *faces* array for each of the groups in *encounteredGroups* is ORed together with the *GroupData* structure corresponding to the new polygons group.
3. While doing steps 4 and 5, the *GroupData’s valid* variable for each group in *encounteredGroups* is set to false. This means that that *GroupData’s* data is no longer valid and it should be disregarded (see next section of this document for more details).
4. Search for the corresponding *FractureGroup* for the group numbers listed in *encounteredGroups.*
5. For each of the corresponding *FractureGroups* for the group numbers listed in *encounteredGroups,* change the *groupNum* variable in *FractureGroups* to the new polygon’s group number.
6. Inside the *FractureGroups* structure, go through all the polygons listed there and change their *groupNum* group number variables to match the new polygon’s group number.

**Why not combine GroupData and FractureGroups into one structure?**

**Structure Definitions:**

NOTE: Both structures use a constructor to initialize their variables (see code in structures.cpp).

struct **GroupData** {

unsigned int size;

bool valid;

bool faces[6];

/\* Domain boundary sides/faces that this cluster connects to..

Index Key:

[0]: -x face, [1]: +x face

[2]: -y face, [3]: +y face

[4]: -z face, [5]: +z face \*/

};

struct **FractureGroups** {

unsigned long long int groupNum;

std::vector<unsigned int> polyList;

};

The reason we do not combine the *GroupData* and *FractureGroups* into a single structure is for performance reasons.

If the two structures were combined, a problem arises when two different fracture groups merge together. The structures could no longer be aligned with the group numbers in an array because the group numbers will be changing whenever groups merge together. This would cause constant searching every time you needed to access any of the data. We still need to search when dealing with the *FractureGroups* array, but save some performance costs by being able to access everything in the *GroupData* arrayfor any group number without any searching.

If you tried to force the alignment by having empty structures where groups were merged to another group, it would require constantly deleting and reallocating the arrays, and copying polygons to the new group every time groups merged to make everything fit as it should. This would be a huge performance hit and probably the worst solution.

The solution implemented was to keep the two structures separate. When clusters merge together, we simply have to set the old cluster’s *GroupData* *valid* bit false (no search required), add its *size* and OR the *faces* to the *GroupData* structure that it is being merged into. We then need to find (search required) the group number that is about to go away in the *FractureGroups* list and change it to the new group number, and change the polygons in that group to the same group number. Nothing is ever re-allocated.

NOTE: When the group number changes in *FractureGroups* after clusters merge together, there will be two *FractureGroups* with the same group number but with different polygons listed. To get all the polygons from a single group, the two lists (or more if clusters continued to merge) need to be concatenated.

***Function getCluster*() overview, defined in clusterGroups.cpp**

The *getCluster*() function is responsible for returning a list indexes to the polygons which match the user’s connectivity option.

Arguments to this function:

1. The program statistics *Stats* object (named *pstats* throughout the code).

There are three user options that deal with fracture connectivity:

1. *boundaryFaces*
   1. This option provides a way to select which faces or walls of the domain the user wants the fractures to connect with. It is an array of 6 elements. A zero means not to enforce a connection, a 1 means fractures must have a connection to that face.
      1. Array elements match to each boundary wall as follows:

[0]: -x face, [1]: +x face

[2]: -y face, [3]: +y face

[4]: -z face, [5]: +z face

1. *ignoreBoundaryFaces*
   1. This option ignores the *boundaryFaces* connectivity option completely and causes *getCluster*() to return a list of all polygons containing at least one intersection.
2. *keepOnlyLargestCluster*
   1. This option keeps causes *getCluster*() to return the largest cluster using the above two options as well. If *ignoreBoundaryFaces* is being used, *getCluster*() will return the largest cluster of fractures in the DFN, even if they do not connect to any of the domain walls. If the *boundaryFaces* option is being used, *getCluster()* will return the largest cluster which connects the user’s required domain walls.

**Code Overview**

**Part 1: Find cluster groups that match the user’s connectivity option**

1. If the user is using the *boundaryFaces* option, search through the *GroupData* and compare the *GroupData’s faces* array to the users *boundaryFaces* array. If the groups *faces* connectivity array connects the required user defined domain walls, add that group number to a list (*matchingGroups* in the code*)*.
2. If the user is using the *ignoreBoundaryFaces* option, go through the *GroupData* array and add all the valid groups to the *matchingGroups* array.
3. If the user is using the *keepOnlyLargestCluster* option, go through the *matchingGroups* array and compare each group’s *GroupData.size* variable. Keep group with the largest size.
4. Search for each group in the *FractureGroups* array and concatenate their polygon lists in a list to be returned by the function.