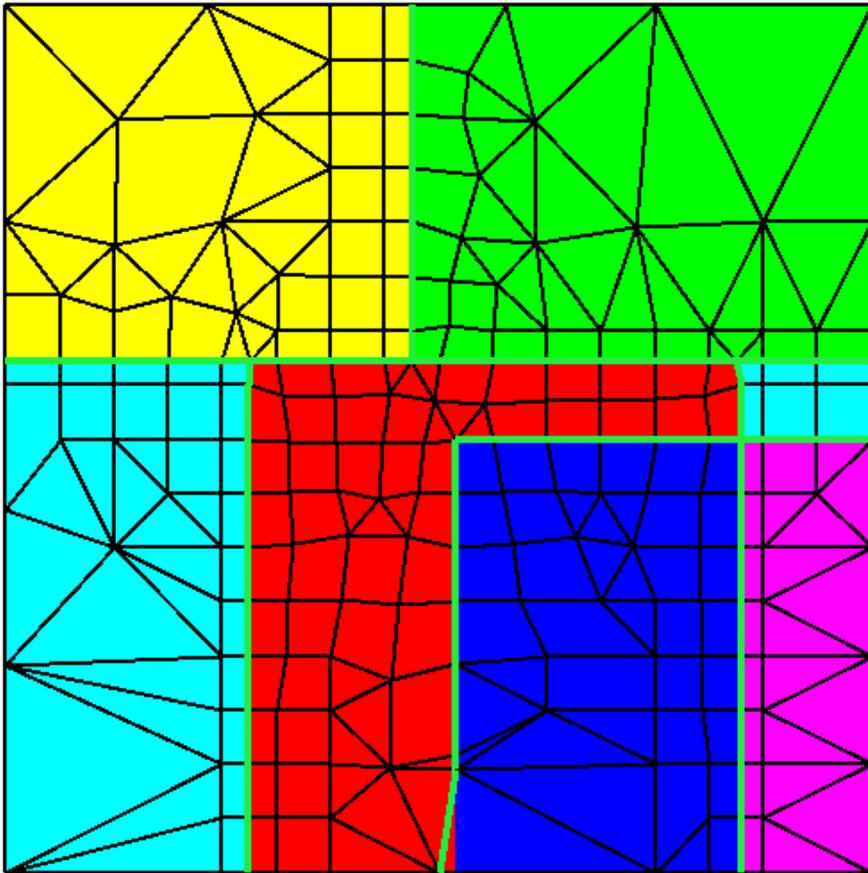


Notes

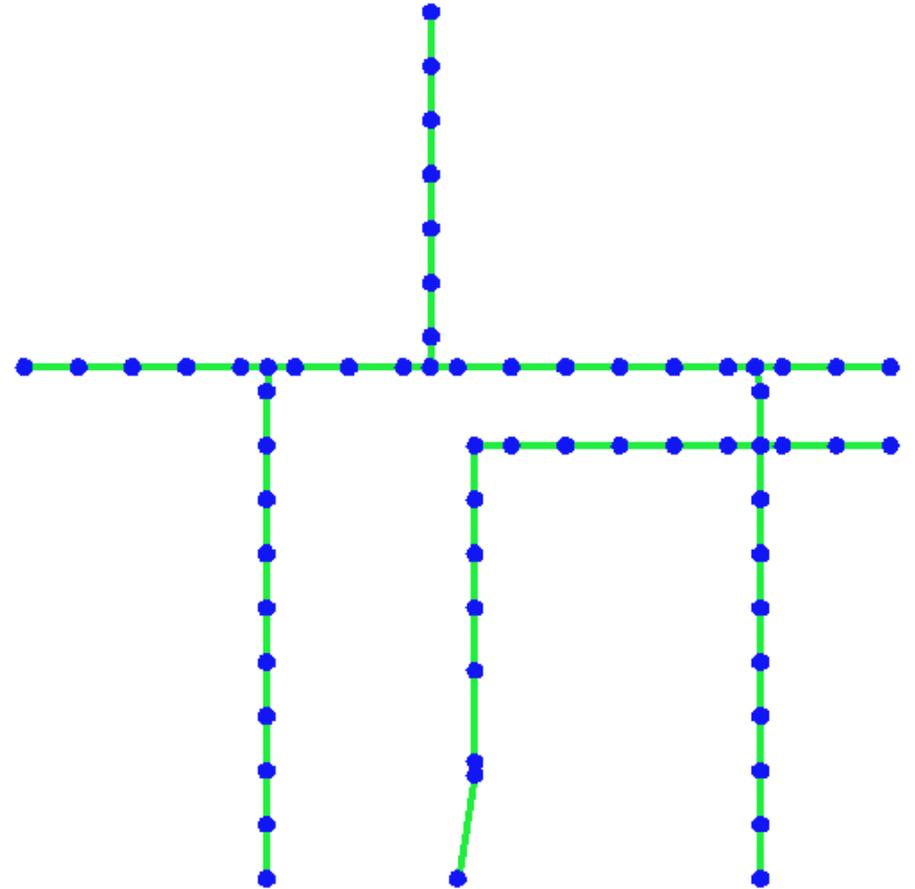
Dividing the mesh along interfaces

Interface graph

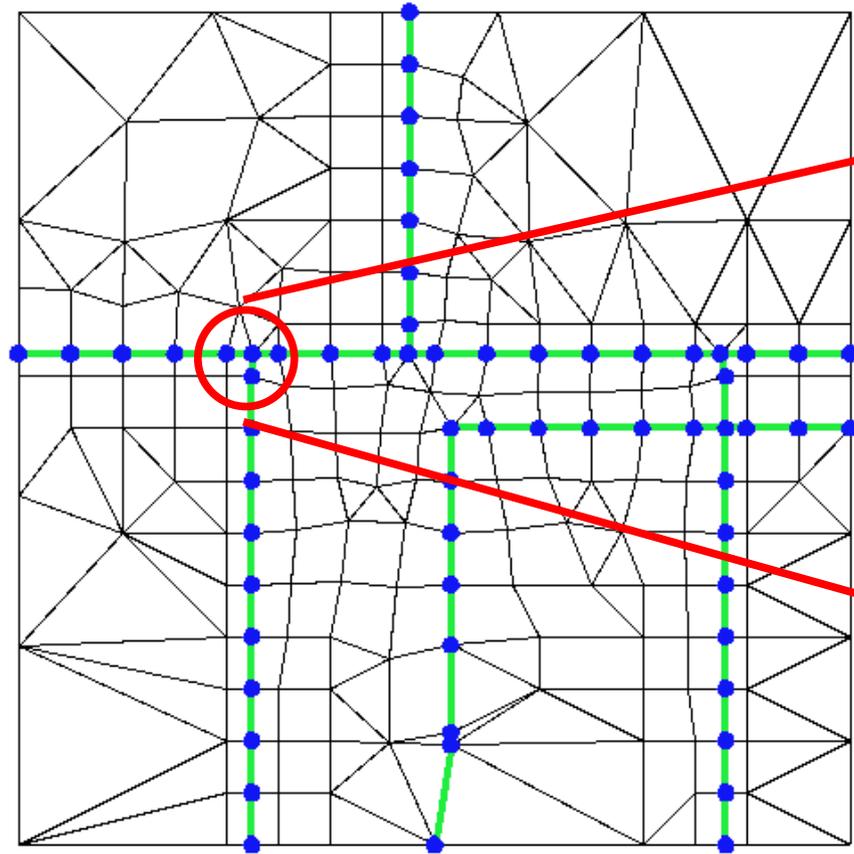
Example: Microstructure with interfaces defined along skeleton internal boundaries.



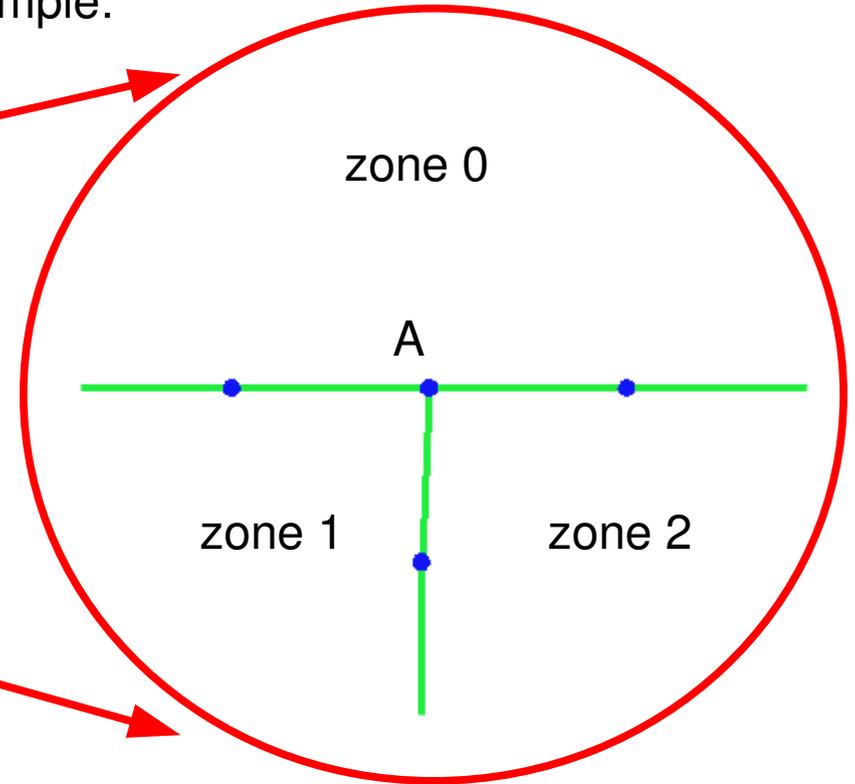
Graph showing interface segments and nodes



Defining “zones” around interface nodes



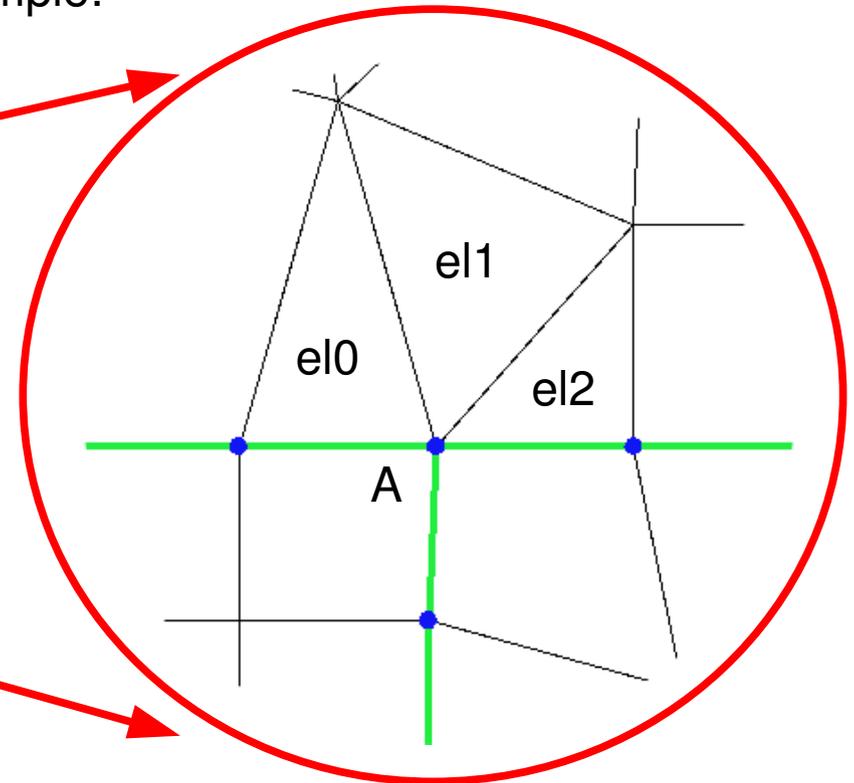
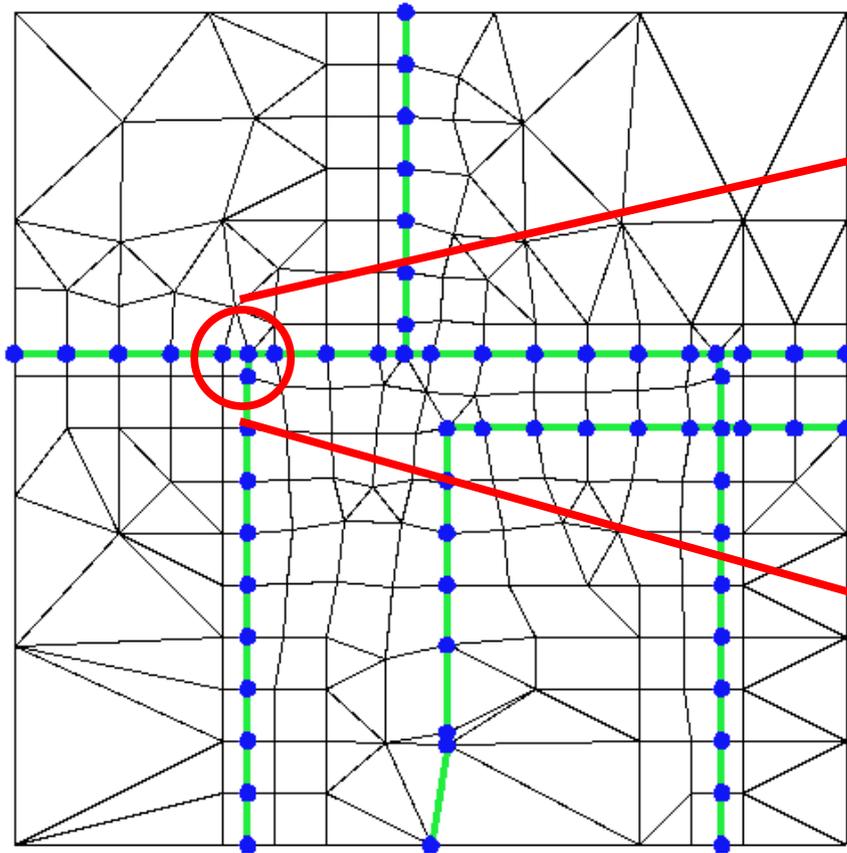
Example:



There are three “zones” around node A. Zones are bordered by interface segments that meet at a node. (Zone-numbering is arbitrary but consistent.)

“Zones” around interface nodes

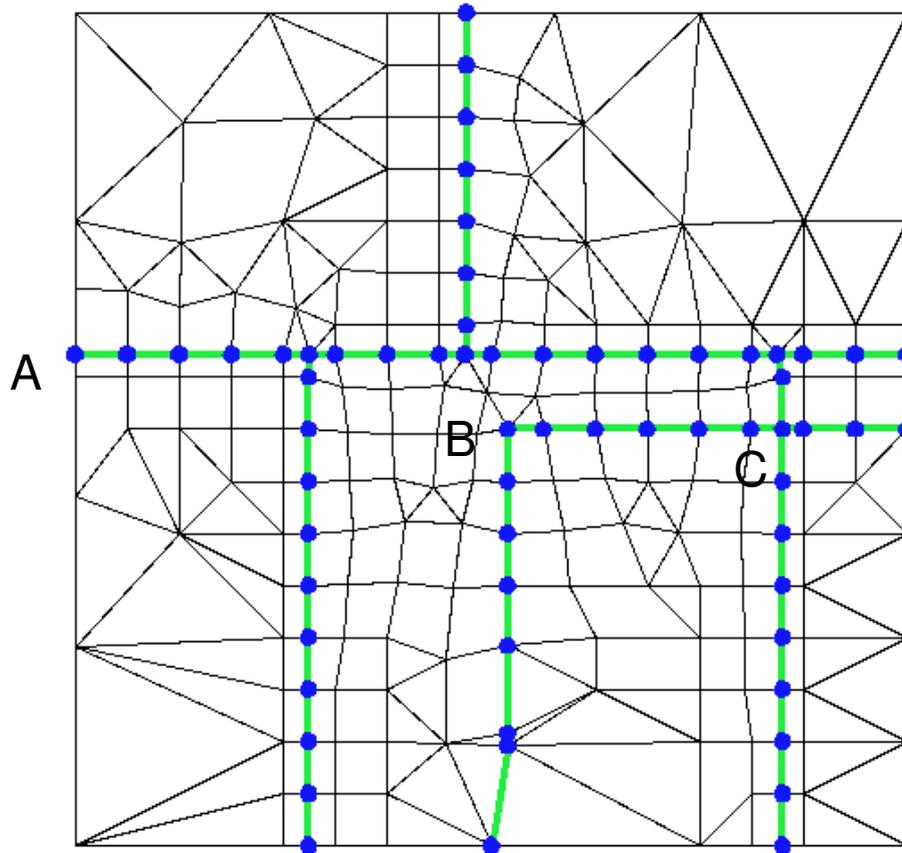
Example:



When a mesh is created, 3 mesh nodes are created at A, corresponding to the 3 zones around node A. The 'fan' of elements (el0,el1,el2) located at zone 0 are assigned one of these mesh nodes. Elements located in different zones get different mesh nodes. This is one way to do “split nodes” when an arbitrary number of interface segments meet at a node.

More Zones

There are two zones around node A. This node is incident on an exterior boundary



There are four zones around node C.
Four interface segments meet at node C.

There are two zones around the corner node B.

More details

- The interface graph (or network) is stored as a dictionary. Example: *interface_seg_dict = { key= (canonical order of interface segment nodes): value=(interface segment)}*. The dictionary is constructed from segments of skeleton boundaries and internal boundary segments determined at mesh construction time.
- For a more compact storage of the interface graph, one could also just store the keys in a list and use the skeleton's segment dictionary to access the actual segments, but access may be slower. May also be worthwhile to explore the complement of the interface graph.

Useful functions

- *getInterfaceZoneNumberAtElem(self, skelnode, skelelem, seg_dict)* (in Skeleton class). The zone number of an element is determined exactly using the chain of relationships between skeleton elements, segments and nodes. Computational geometry (e.g. using angles) is not used.
- *countInterfaceZonesAtNode(self, skelnode, seg_dict)* (in Skeleton class)
- *getOppositeSegment(self, node1, seg1, skeleton)* (in SkeletonElement class). This function gets the segment opposite seg1 and that is incident on node1. seg1 and the segment we are looking for form a 'V' on the boundary of the element.