BUILD TOPOLOGY:

This was a fairly trivial part of the assignment, I simply allocated space in the topology object for each necessary component of the mesh, then followed the basic pseudocode outline as provided in the assignment documentation when it came to creating and adding halfedges and their elements.

PRIMITIVE CONSTRUCTION:

Another fairly simple task. I found some online sources which detailed algorithms for efficiently traversing a halfegde data structure. Using those and my own understanding I completed the required functions.
(https://www.flipcode.com/archives/The_Half-Edge_Data_Structure.shtml)

NON-MANIFOLDNESS:

This task consisted mostly of just applying the previous two tasks, but I ran into an issue where it appeared that nonmanifold vertices were not being detected correctly. To solve the problem I used view_with_topology to check which faces were being counted as adjacent for each vertex. This led me to correct and improve my previous task since I could visually see that there was an error with how I was finding adjacent faces for a vertex.

ADDITIONAL MESH FUNCTIONS:

Also trivial. I had already implemented get_3d_pos for the previous assignment, and vector() was a simple application of that function and some numpy matrix math. faceNormals() was slightly more difficult but I found a source online which laid out the process in great detail and allowed me to complete the function.
(https://www.danenglesson.com/images/portfolio/MoA/halfedge.pdf)

LAPLACIAN SMOOTHING:

Much more difficult than the previous assignments because it revealed several mistakes in my previous implementations of certain functions. Using view_with_topology and my previous sources, I corrected these errors. I also found a source online that directed me towards using an “empty” / dummy coordinate vector to compute the average position of all neighbors which helped significantly. (https://nosferalatu.com/LaplacianMeshSmoothing.html)

EDGE COLLAPSE:

A very difficult task. I worked through the basic functions using the single edge collapse test object. This helped me to establish a process. First, find the vertices of the edge being collapsed (shown in green) and choose one of them to be deleted, and one to be kept. This choice was arbitrary, and so I also choose to delete the “first” vertex (i.e. the one at index 0) and keep the second point. Next, we find the faces that are incident to the edge being deleted (shown in orange), and mark them for deletion. Now we can choose which two other edges/halfedge pairs

are going to be deleted. We do this by selecting the halfedge for each face being deleted whose vertex is the vertex being deleted (shown in red). Now we have our deletion targets and can move onto repair.

Repairing vertices is a matter of collecting all the halfedges that originate from the vertex slated for deletion. We can then simply reassign all those halfedges to point to the vertex not being deleted, that is the new vertex. Repairing faces is slightly more complex. First we find the faces that need to be repaired, which is every face that is losing a halfedge, and is not being eventually deleted (shown in purple). For each such face, a new halfedge needs to be found that will replace the one being lost. We can find the appropriate replacement halfedge in the following way: find a halfedge that is being edited (i.e. was associated with one of the collapsed edge's vertices) and that also shares a vertex, either base or tip with the face (shown in yellow). We can then connect the halfedge that matches these characteristics to the face, accounting for directionality.