Final Project for Computational Topology

Given a simplicial complex, compute the boundary matrix

By: Jacob Stern and Tamar Yastrab

I. Abstract

Our project interprets a simplicial complex in order to create the various boundary matrices that describe each dimension of a simplicial complex. For a *k*-simplicial complex, the *p*th boundary matrix is the matrix whose rows are the simplicies with dimension (*p*-1) and whose columns are the simplicies with dimension *p*. An index is marked with a 1 if the (*p*-1) row is a face on the *p* column. The index is 0 otherwise. We accomplish this by taking simplicies as input from the user and recursively find the faces of each simplex one dimension lower, then the faces of those faces, and so on until we reach individual vertices and eventually the empty vertex. We repeat this process until we have processed a complete simplicial complex. To represent this structure, our code models the complex as a graph, where the nodes correspond to the simplicies and keep track of their dimensions. The dimensions enable us to calculate the *p* boundary matrices between each dimension accurately. Additionally, we compute the total boundary matrix of the entire simplicial complex by using the list of all simplicies as the rows and columns to populate the matrix.

II. Introduction

In rigorous mathematics a topology might classically be defined in the language of set theory. However, to gain a more intuitive grasp one can consider topology as a generalization of geometry. Features like length, size, and sides in geometry have deeper analogous properties in topology, where properties are preserved even if the original shape is stretched and bent. So long as no additional holes are added to the figure, topological figures can be molded, and for this reason it has been dubbed “rubber sheet geometry”.

The simplest way to think of a simplex is to think of the generalization of the triangle. In two dimensions the simplest polygon is the triangle. Triangles make up all polygons such that all of two-space itself can be triangulated. Much of understanding 2-manifolds is about understanding triangles, and thus the goal of simplexes is to generalize this idea for all dimensions.

III. Content

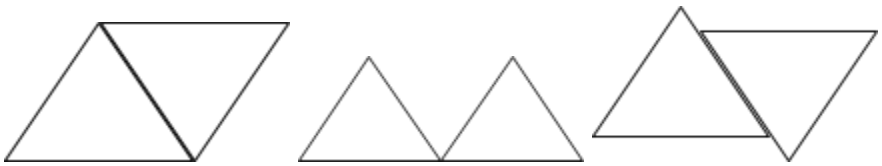
The formal definition of a *k*-dimensional simplex is a convex hull of *K*+1 vertices. In other words, if there is a set of points {a,b,c ....} such that for any element of that set *p*, all nonzero vectors in the set {a-p,b-p,c-p,...} are linearly independent then the set {a,b,c ....} , is the

set of vertices of the simplex. The simplex itself is defined as a linear combination of the vertices where the coefficients are all positive and add to one.

It is clear that a *k*-dimensional simplex is a shape in dimension *k*, because the K vectors are linearly independent in K dimensions. It is also clear that three points in R2 would form two linearly independent vectors, making them not collinear and thus form a triangle. Furthermore, any 2 points form both a line segment, and a 1-simplex. So a 1-simplex is a line segment. Therefore it is confirmed that the intuition behind the generalized triangle is met by the definition of the simplex.

Dimension	Simplex
-1	The empty set
0	A point
1	A line
2	A triangle
3	A tetrahedron

A simplicial complex can be thought of as a structure made up of simplices. The formal definition of a simplicial complex is a set of simplices such that the intersection of any two simplices is another simplex in the complex. This condition ensures that the simplices meet "where they should." For example it makes sense if two triangles share either an edge or a corner, however "missaliagned" triangles are not allowed in a simplicial complex.



Another important building block of topology is the manifold. A N-manifold is a topology that is locally homeomorphic to Rn . In other words, a manifold looks like RN when zoomed in on. A 2-manifold looks like a flat plane when one zooms infinitely close to the manifold much like a person standing on the Earth believes the ground to be flat even though it's really roughly spherical. While not all manifolds can nesserialy be given an associated simplicial complex, it is a fundamental result of topology, that all smooth (that is, infinitely differentiable) N-manifolds can be show to make up an N-complex.

The main reasons for studying simplicial complexes are computational in nature. For one thing it is much easier for a computer to work with the straight lines, and graph nature of a simplex, then it is able to work with abstract topologies. There are also certain computations that

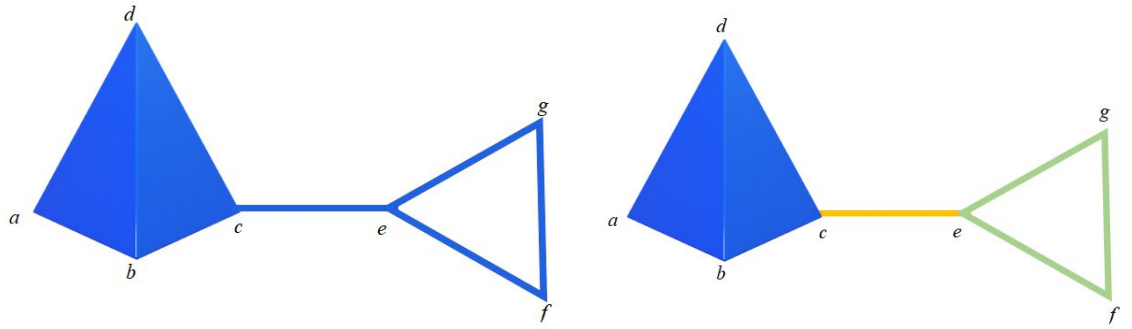
are easier to work out via, simplicail complexes.One example of such a problem is the computation of homology groups via boundary matrices.

Boundary matrices are representations of the relationships of the various simplicies in a complex. The *p*-th boundary matrix represents the p-simplices as columns and the (*p*-1) simplicies as rows. The entries are zero except where the simplex represented by the row is a boundary of the simplex represented by the column. The computation of homologies and other topological invariants can be derived from only the boundary matrices of a simplex.

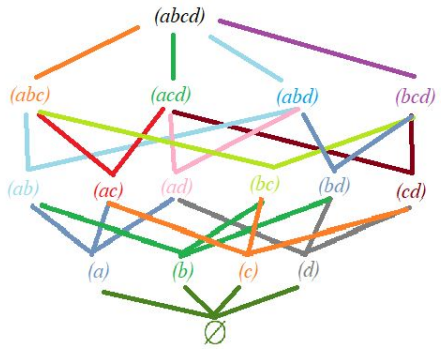
The objective of our code was to efficiently store simplicial complexes by creating a graph structure whose nodes were all of the simplices in the larger simplicial complex. Our approach was to accept large simplices and recursively store their faces of lower dimensions until we reach dimension -1, in the case of the empty simplex. Once we have a fully populated graph representing a simplicial complex, we create the individual *p* boundary matrices and the large boundary matrix which has all of the vertices as the rows and columns which is essential to find the homology groups and Betti numbers of a simplex K.

IV. Code

Our project seeks to take a simplicial complex as input from the user and output the boundary matrices for each level of dimensions, as well as one large boundary matrix where are the simplicies within the larger complex are represented. To do this, we ask the user to enter each non-connected simplex separately, so that for this simplicial complex:



abcd, *ce*, and *fdg* are imputed. The way we want to represent each simplex is by subdividing recursively into its faces until we arrive at dimension 0. For example, if a user entered the tetrahedron *abcd*, we would want to analyze *abcd*, the four triangle faces *abc*, *abd*, *acd*, and *bcd*, the subsequent six lines *ab*, *ac*, *ad*, *bc*, *bd*, and *cd*, the smallest dimension 1 simplicies, the four points *a*,*b*,*c*, and *d*. We developed a graph structure, where the nodes of the graph are these simplicies, and edges are shared by simplicies and their faces, such that:



We begin by creating a Simplex class that will serve as the nodes in the nodes of the graph.

```
class Simplex(object):
     def __init__(self, name):
          self.name = list(name)        # typecase the name so it split into the vertices
          self.dim = len(name)-1        # the dimension is the number of vertices minus one
          self.neighbors = []
          self.wasVisited = False

     def addNeighbor(self, s):
          if s not in self.neighbors:
               self.neighbors.append(s)
```

Each simplex will be initialized with a name, dimension, an empty list of neighbors, and a boolean value representing if that node has been visited. The name will be a list of vertices, so if the user enters triangle *abc*, the name will be [a,b,c]. The dimension is calculated by counting the number of vertices and subtracting one, and this will be important for determining which vertices share and edge. The 'was visited' boolean will not be used in the simplicial complex code, but we thought it was valuable to include if we want to add more functionality to the code for depth and breadth searches throughout the simplicial complex. The add neighbor method, when called, will add to a node's neighbor list without repetition.

```
class SimplicialComplex(object):
     def __init__(self):
          self.simplicies = dict()
          self.highest = 0
```

We will now discuss the simplicial complex class. The class is initialized with only two attributes. There is a dictionary that will store all of the nodes, or simplicies, of the graph, with the keys set to the name of the simplex and the corresponding value as the simplex object discussed above. Highest will record the greatest dimension of any simplex on the graph, and as more objects are added, this will be updated.

When the user inputs simplicies, they will be added with the addEdge() method. Each time a simplex is added, the getS() method will be called.

```
def getS(self, name):
     # if this is a new node, add it to the grapph
     if name not in self.simplicies: self.simplicies[name] = Simplex(name)

     #keep track of the highest dimension seen so far in the graph
     if self.simplicies[name].dim > self.highest: self.highest = self.simplicies[name].dim
     return self.simplicies[name]
```

This method checks if the graph already contains the simplex, and if not, it adds the name to the simplicies dictionary along with the newly initialized simplex object. The method also checks the dimension of the new node, and updates the graph attribute 'highest' if it discovers a higher dimension simplex. This will be important for determining from which dimension to start calculating boundary matrices later on.

```
def addEdge(self, name):
     self.getS("")
     x = [''.join(l) for i in range(len(name)) for l in combinations(name, i+1)] # all combinations of the simplex
     for i in x:
          self.getS(i)
     for i in x:
          for j in x:
               if 0 not in [c in j for c in i] \
                  and self.simplicies[i].dim == self.simplicies[j].dim-1:
                    self.simplicies[i].addNeighbor(self.simplicies[j])
                    self.simplicies[j].addNeighbor(self.simplicies[i])

               if self.simplicies[i].dim == 0: self.simplicies[i].addNeighbor(self.getS(""))
```

AddEdge() begins by creating an empty simplex with dimension -1 which is necessary for determining the reduced homology of the complex. This ensures that even graphs with only one simplex will have this empty node. Next, the code takes the user's input and finds all the combinations of strings which can be made from the vertices. For example, if the user input *abcd*, x = ['a','b','c','d','ab','ac','ad','bc','bd,'cd','abc','abd','acd','bcd','abcd']. These are all of the faces of every level of the simplex. With these string combinations, we initialize a Node in the graph with each one. Once this is completed and their dimensions have been calculated and stored, we want to discover which simplicies share edges. We do this by looping through all of the simplexes and checking for 2 conditions:

1. The smaller simplex is a face of the larger one
2. The simplicies are within one dimension of each other

To find the smaller simplex, we checked if all of the vertices in the smaller node were present in the larger one in any order. This means that we wanted *ac* to be found in *acd* and *abc* as well, even though in the string vertex *b* separated *a* and *c*. However, this also allowed for a potential edge to be shared between point *a* and triangle *abc*, for example, and thus we check to ensure that the simplices are within one dimension of each other, so that *a* with dimension 0 and *abc* with dimension 2 will be excluded.

If these two conditions were satisfied, we added both simplicies to the neighbor list of the other. Additionally, we forced the empty node to be neighbors with every node with dimension 0.

After the user completes entering each of the simplicies into the graph, we are able to begin calculating the boundary matrices. The *p*th boundary matrix of the simplicial complex has *p*-simplicies as columns and (*p*-1)-simplicies as rows. An index[*i*][*j*] = 1 if the *p*-1 simplex is a face of the *p* simplex. The output of the program is *p* matrices, plus a larger boundary matrix which represents all of the simplicies in the entire simplicial complex as the rows and columns. This process is split into two functions: One which calculates the rows and columns for every level of the graph, and one which takes those bounds as input and populates the boundary matrix.

```
def matrix(self):
     for p in range(self.highest,-1,-1):
          row = []
          col = []
          for i in self.simplicies:
               if self.simplicies[i].dim == p-1:
                    row.append(i)
               if self.simplicies[i].dim == p:
                    col.append(i)
          print("Dimension ", p, "-", p-1, ":")
          print("Rows: ", row)
          print("Cols: ", col)
          self.matrixPopulation(row,col)
     print("Complete Boundary Matrix:")
     self.matrixPopulation(self.simplicies,self.simplicies)
```

Starting from the highest dimension of the graph, the function will create an empty 2-dimensional matrix until stopping at dimension -1. For every dimension, it creates two empty lists, corresponding to rows and columns. The code then loops through the nodes and adds nodes with dimension *p* to the rows and *p*-1 to the columns. Once it has added all of the nodes to their appropriate lists, the function calls matrixPopulation() and sends the rows and columns as arguments. It will also call matrixPopulation() sending in the entire list of vertices as the rows and columns, which will calculate the large boundary matrix.

```
def matrixPopulation(self, row, col):
     matrix = [[0 for i in range(len(col))] for j in range(len(row))]
     r = -1
     for i in row:
          r += 1
          c = -1
          for j in col:
               c +=1
               if self.simplicies[i] in self.simplicies[j].neighbors:
                    matrix[r][c] = 1
          print(matrix[r])
```

The function begins by initializing a 2-dimensional list with rows corresponding to the length of the row list and columns corresponding to the length of the column list. It then loops through the lists and sets entries to one when the *p*-1 simplex is a neighbor of the *p* simplex. This function is called ;self.dim' times, meaning it will be called for every level of the graph. The last call produces the large boundary matrix, which diagonals all equal to zero.

V. Conclusion

We have now seen that given a simplicial complex, we can compute the boundary matrix for every *p*-dimension. By representing simplicies as nodes in a graph, we are able to deconstruct a simplicial complex into a structure which stores every layer of simplicies, and loops through the vertices for each dimension to create boundary matrices. Building upon our last project, it was enlightening to advance the idea of triangulating topological manifolds, as simplicial complexes aim to generalize triangulatizations for higher dimensions. The run time of this project is $O(n^2)$ because it loops through the vertices to determine the row and column indices. An alternate way of storing this data is by creating matrix objects that could be initialized and added to upon each encounter with a new simplex, and while this may be more time efficient because vertices would be added in O(n) time, it would take up more space. Further optimizations may include expanding the scope of input to include coordinates, point field, or triangulations themselves.

Works Cited

[1] Edelsbrunner, Herbert, and J Harer. Computational topology : an introduction. Providence, R.I: American Mathematical Society, 2010. Print.

[2]] John Henry C. Whitehead, On C1-complexes, Ann. of Math. (2) 41 (1940), 809–824.