Patrick Walker
11 May 2020

Polygon Gluing: a Coding Project
(Many Foldings in a Manifold)

Every 2 dimensional manifold has a Polygon Schema—it can be shown as a polygon with some specific glueing instruction. However, when presented with a polygon schema, it can be a long process for determining the manifold to which it represents. The normal method for determining the manifold to which a polygon schema represents consists of 3 parts: determine the Euler characteristic, determine orientability, and determine the genus, which are the three qualities that form a complete invariant of 2 dimensional manifolds. In other words these three qualities define which objects are homeomorphic to each other. For use in quickly classifying highly complex Polygon Schema, we have developed a computer program that determines the homogeneity class of the presented polygon schema.

The issue in translating the algorithms used for determining Euler Characteristic and Orientability is that they are highly symbolic with no physical measurement being taken that is easily translatable to code. Thus a method of interpreting the actions that would normally be taken by a human in finding these values had to be developed.

For the purpose of finding the Euler Characteristic, three values have to be found, the number of vertices on the manifold, the number of edges, and the number of faces. Because our object involves a single polygon, we already know that the number of faces is exactly 1. However, finding the number of vertices and edges is more difficult. After all, some vertices on the polygon may actually be the same vertex on the 2 dimensional manifold. Normally, to determine the actual number of vertices on a polygon schema you use a circle tracing algorithm, in which you pass across glued edges to create a circle which only contains one vertex. Any vertex on the polygon included in this circle, is thus actually the same vertex on the 2d manifold. The issue in translating this to code is that it is highly symbolic and difficult to replicate such a drawing method for the computer. As such, a shortcut is needed that can represent this methodology without actually replicating it.

Furthermore, determination of orientability on these polygon schema is related to the existence of a mobius strip on the polygon. So discovering a way to use our notation to quickly determine if a mobius strip exists was necessary, rather than physically isolating said mobius strip to prove it exists, or in its absence, prove it doesn't exist. However, determining such a method was far less challenging that determining how many vertices actually existed on the glued manifold

The next part of this report includes pseudocode for each function of the program, where each function has a short summary of its use, followed by a lengthy description of its functionality. The use of functions in building this code, instead of constructing one long script allows for greater recyclability of the program in other projects and computer systems. After the function description section, we show a few examples of the results. Then, at the end of the report, we included an appendix with images of the code (with integrated comments where we felt it was necessary to explain specifics of code).

Def generate_polygon_vertices():

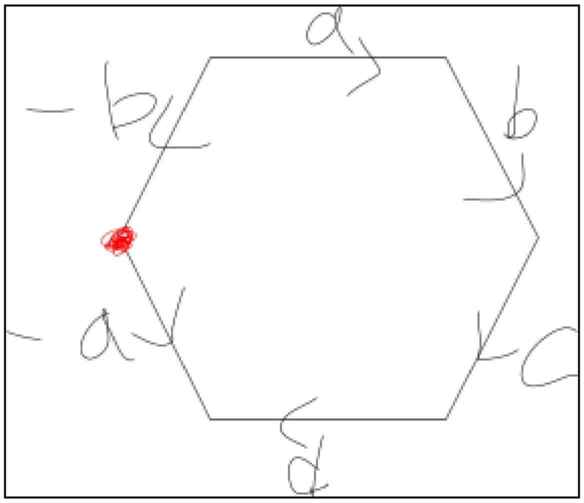
- *Determine a relationships dictionary to aid in imitating circle tracing in later functions*
- *Determine the identities of edges present in the polygon for later referencing*
- *Determine a list of all edges with gluing instructions present in polygon for later referencing*

Our program starts out by executing the "generate_polygon_vertices" method, which first asks the user to input a letter and sign pattern that surrounds a polygon in clockwise orientation; each letter corresponds to a side and the sign represents the direction of the arrow (clockwise or counterclockwise). For example, the glueing problem we were given on homework 8 would have the glueing notation of the form a,b,c,d,d,-c,-a,-b (note that there are no spaces between each comma and letter). In continuation of the function, the "edges" variable is an array that contains the comma delimited items of the "glueing" input (a manipulation of the input data to an easily readable format for later use).

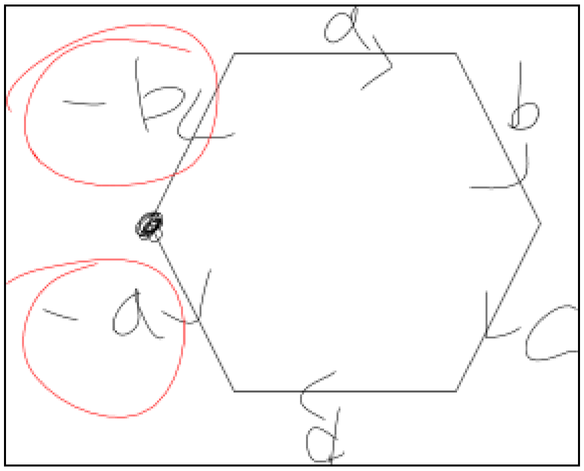
For how the program actually performs these actions, we first input the glueing instructions into our program. Then, for each edge that was input by the user, the program ignores the sign in front of each edge, and collects each edge identities it comes across, and adds it to “check_edges” (a list of edge identities). If it comes across an edge with a previously used identity (is glued to a previously checked edge), then it marks it as the second one of its type.

To understand what the function does next, we have to understand how we determine the actual number of vertices in the surface once it has been glued:

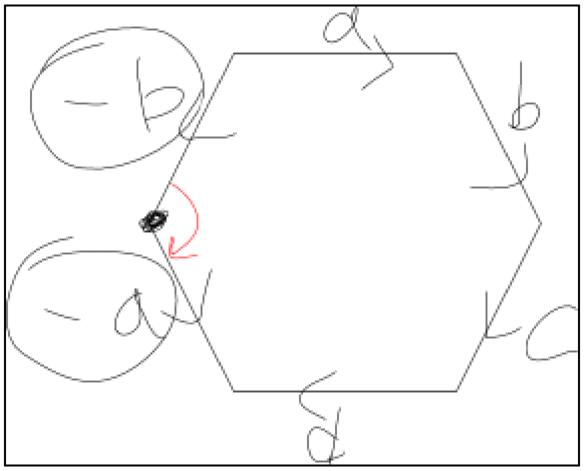
1. Our code first takes a vertex



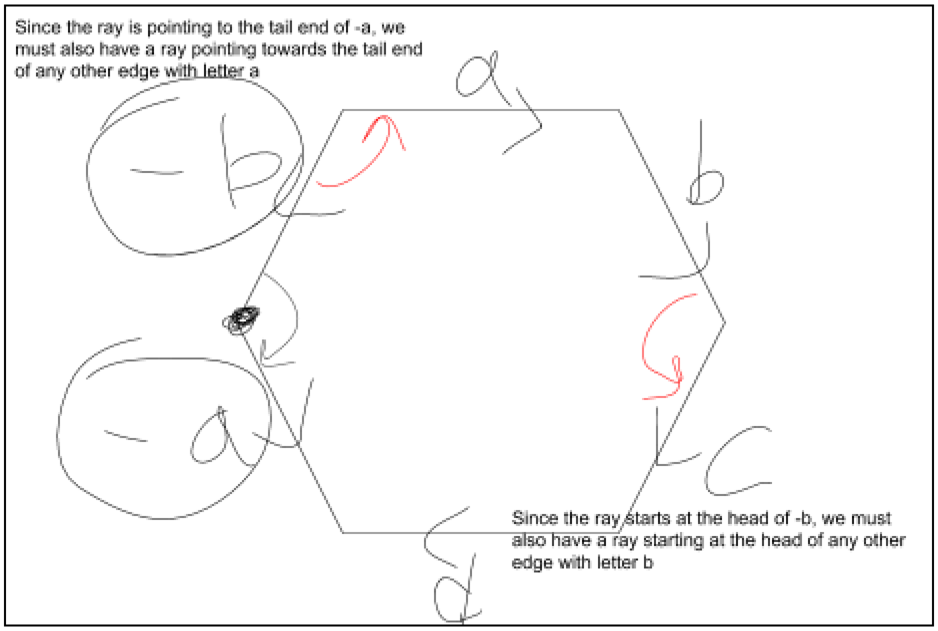
2. From there, we look at the edges adjacent to the vertex



3. Then, we draw a ray from one point on one of the edges to one point on the other edge

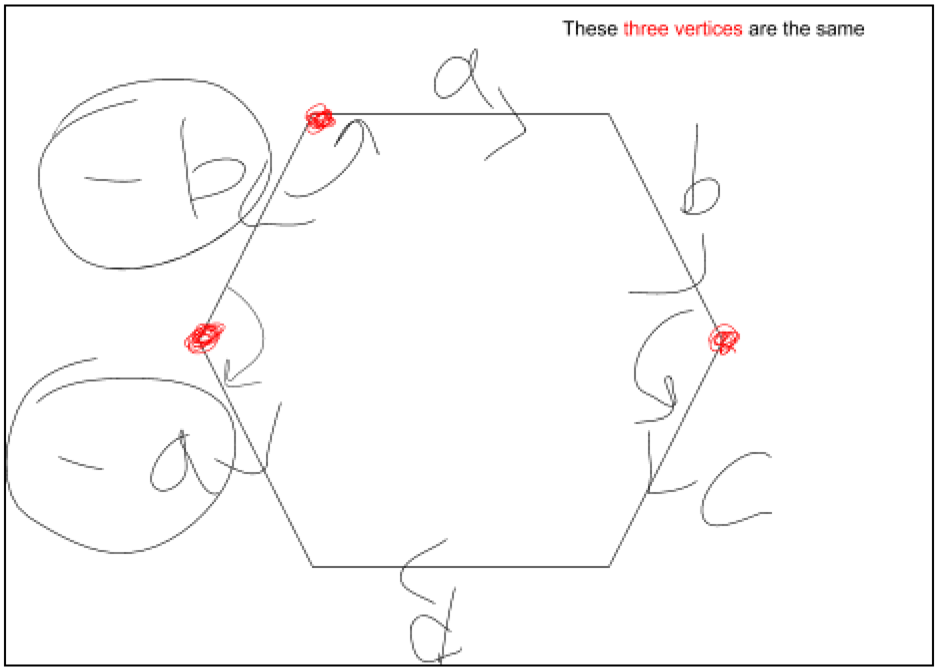


4. We then will then draw corresponding rays on each corresponding edge

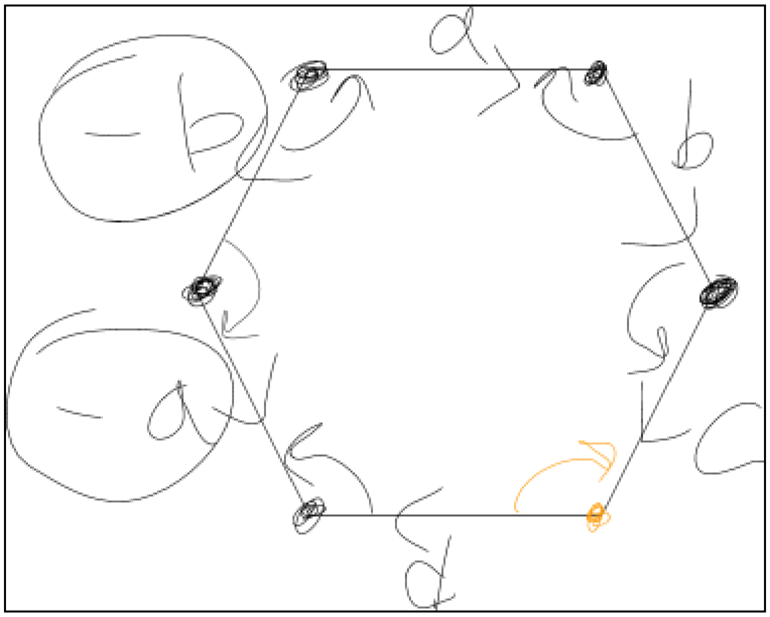

Since the ray is pointing to the tail end of -a, we must also have a ray pointing towards the tail end of any other edge with letter a

Since the ray starts at the head of -b, we must also have a ray starting at the head of any other edge with letter b

5. The vertex corresponding to each ray is then made to be the same



6. Finally we repeat these steps multiple times until we know which vertices are the same.



So, we need a way to imitate this circle tracing. If you consider the arcs above, you notice that each edge on the polygon has two arcs associated with it, each arc has one point on each edge the arc touches, thus each edge has two ‘points’ we are interested in—one point on the ‘upper’ end of the edge and one point on the ‘lower’ end of the edge in relation to the gluing direction given. Furthermore, by considering the arcs, we see that each of these points corresponds to another point respectively, found on another edge. Thus each arc can be considered a relationship of two points on two different edges. And we can thus create a dictionary in our program to store these relationships. In terms of what this means, by calling a point in this dictionary, it returns to us the point that an arc would be traced to. This dictionary will be aptly called, ‘relationships’. We will later use this dictionary in our circle tracing method for determining the number of vertices.

Def determine_boundary_edges(check_edges, edges):

- *Determine which edges are boundaries. Important for determining vertices in 2d manifold, determining euler characteristic, and classifying manifold*

This function cycles through identities in check_edges, and determines if there are 1 or 2 sides with that identity. If there is one side, the edge does not have a partner to glue to, thus it is a boundary. If there are 2 sides associated with that identity, the edge does have a partner to glue to, thus it is not a boundary. The function returns the list of boundary edge identities for later use (in “determine_vertices”, “determine_number_of_edges”, “determine_orientability”, “determine_genus” and the final steps).

Def determine_vertices(relationships, boundaries):

- *Determine the number of vertices on the 2d manifold for later use in calculating the Euler characteristic*

The parameter “relationships” is the relationships dictionary generated by the first function, generate_polygon_verteces. This relationship dictionary is important for the circle tracing methodology used to determine which polygon vertices are the same in the 2d glued manifold. The boundaries parameter ensures we know where our boundary conditions are, as they have special considerations for the circle tracing method for determining the actual number of vertices on the 2d glued manifold.

With these two parameters, we can begin the setup for imitating the circle tracing method. The first thing we have to do is collect all of the arbitrary points that we have defined before, so that we have an easy and manipulatable reference to all the points we need to consider. This is done by using the .keys() function on our relationships dictionary, which will give us a list of all of these points.

In accordance with how we described imitating the circle tracing method in our description of the generate_polygon_verteces function, we begin at a random point and begin tracing around a vertex, following the proper gluing pattern until we complete our circle which must contain only one vertex on the 2d manifold, thus each circle found indicates the existence of exactly one vertex. We imitate this in our function by choosing one of the arbitrary points in our list of points and saving it as a starting point, while deleting it from the list of points to represent having used that point, and also save the point to a temporary list. Then, we begin cycling through the following steps until we reach our starting point again.

1. Use the relationships dictionary to find where to trace the arc as previously defined
2. Remove that point from the list of points, and append it to the temporary list
3. Follow the gluing and pass over to the correlating point on the glued edge. I.E A2 lower passes to A lower, B Upper passes to B2 upper.
4. If this point is the startpoint, you're done. If not, remove this point from the list, and save the point to a temporary list, and continue looping

Once the loop is complete, the temporary list defines a circle we have traced, so we save it to another list that contains all of our defined circles. Furthermore, if there are any remaining points in our list of points, we clear our temporary list, and repeat the above to generate another circle. Circles are continuously generated until all points have been used. Now, we can simply calculate the number of circles we have generated, which is equal to the number of vertices in the glued manifold.

However, there is a condition that breaks this method. If a manifold has a boundary condition, what occurs if your circle tracing reaches a boundary before reaching its starting point? Because the boundary condition does not have a glued edge to pass the circle tracing to, the circle tracing loop we defined above stops functioning. However, there is a trick to avoid this. Consider what occurs if you start at a boundary condition. That circle tracing does not end at its starting point, because you can’t reach a different edge that glues back to that point. However, it is guaranteed to end at another boundary condition. Furthermore, each boundary condition has two ‘points’ we are concerned about, thus we always have an even number of these boundary ‘points’. Given that each circle tracing, starting at a boundary point must thus end at another boundary point, each contains exactly 2 boundary ‘points’, then we can create a loop system to imitate the circle tracing that safely deals with all boundary conditions.

First, we will start with a random point in our boundary conditions, then we remove this point from both our list of boundary conditions, our general list of points, and also save it to a temporary list much like we did before, then we start a loop:

1. Use the relationships dictionary to find where to trace the arc to as previously defined
2. Remove that point from the list of points, and append it to the temporary list
3. If the point is a boundary condition, we also remove the point from our list of boundary conditions, and then end the loop
4. If the point is not a boundary condition, we pass over to the correlating point on the glued edge in the same manner as described before, and remove that point from the list of edge points while saving it to our temporary list.

Once the loop is complete, we have defined a circle, and removed two boundary conditions we need to worry about, we reset our temporary list and repeat until all boundary conditions are dealt with.

Thus, the proper method for imitating the circle tracing method for determining the number of vertices is to follow the method for dealing with boundary conditions, before transitioning to the general purpose loop. Then we count the number of circles, which are equal to the number of vertices.

Def determine_number_of_edges(edges, boundaries):

- *Determine the number of edges once the polygon has been glued properly for use in determining euler characteristic*

This function takes as parameters "edges" and "boundaries" and calculates the number of edges on the 2 dimensional manifold. The number of edges is calculated using the following formula: ((the number of edges on the polygon - the number of boundaries)/2) + the number of boundaries. The number of edges is returned and is used when finding the Euler characteristic (in "euler_characteristic").

Def euler_characteristic(vertices, number_of_edges):

- *Determine Euler characteristic*

This function finds the Euler characteristic of our surface, where the input is "vertices" and "number_of_edges". The Euler characteristic is the number of vertices - the number of edges + 1. The Euler characteristic is returned and is used when determining the genus of the surface (in "determine_genus") and in the final steps of our program (the output).

Def determine_orientability(edged, check_edges, boundaries):

- *Check if a mobius strip exists in manifold to determine if the manifold is orientable*

This function takes as parameters "edges", "check_edges" and "boundaries" and determines the orientability of our surface. It first sets the orientability to true and then loops through the edge identities, checking whether each identity is not a boundary, and if it is in the edges list (the positive identity is present), then we want to check the second side (with a - sign) of that identity. If it is also positive, a möbius strip exists as part of the surface, and is thus not orientable. However, if the edge identity is not present in the edges list, then the negative edge is present. Then we want to check the second side, and if it is also negative, a möbius strip exists and the surface is not orientable. However, if we cycle through all edge identities without finding a mobius strip possibility, then the surface is orientable. This function returns an orientable boolean (true/false) value, and is used when determining the genus of our surface (in "detemine_genus") and in the final steps as part of the output.

<u>Def determine_compacted_boundaries(edges,boundaries):</u>

- *Deal with conditions in which a polygon gluing gives two or more boundary edges in sequence*

One issue with polygon gluing instructions, is that the use may decide to input a gluing instruction that contains more than one boundary edge in sequence. In actuality, these are not different boundary edges, and are in fact one boundary condition, split into sections. This can cause problems for determining the genus, and when printing the number of boundaries given. Thus this function checks each boundary, and if it finds two in sequence, it subtracts one from the number of boundaries, representing the two boundary edges given actually being the same boundary condition. However, there is a specific problem that arises with this algorithm. If all edges are boundary conditions - the 2d manifold is a filled in circle, then this algorithm would actually produce a result of no boundaries. However, this error only occurs in this specific case, so we can check to see if this case actually occurs, and if it does, set the number of boundary conditions to 1, as we already know that a filled-in circle manifold has exactly one boundary condition. This corrected boundaries number is used for determining genus and for the final printing of classification

<u>Def determine_genus(Euler, orientable, compacted_boundaries):</u>

- *From number of vertices, number of edges, number of boundaries and orientability, determine the genus of the surface*

This function determines the genus of our surface. If the surface is orientable, then the genus is calculated using the following formula: 0 - ((the Euler characteristic + the number of boundaries - 2)/2). If the surface is nonorientable, then the genus is calculated using the following formula: 0 - (the Euler characteristic + the number of boundaries - 2). This formula comes from a rearranged version of the formula for calculating the genus of orientable surfaces with boundary components, which reads $\chi = 2 - 2g - b$, where χ is the Euler characteristic, g is the genus and b is the number of boundaries. The genus number is an integral part of our program, and is presented to the user in the final steps.

Final steps:

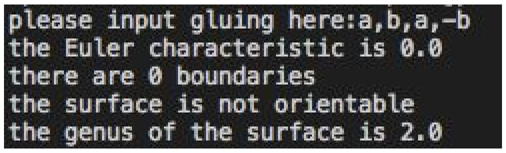
The last number of steps that our program does is print the Euler characteristic of the input surface, the number of boundaries of the surface and whether or not the surface is orientable. Finally the program also prints the genus of the surface.

The next steps to perfecting this program is to have information like the genus, orientability and Euler characteristic used to find the specific resulting surface, i.e. a Klein bottle, a torus, double torus, etc. - to a certain level. An example of searching for a surface would be: if the surface is orientable, then our program looks at a list of surfaces that are orientable with any number Euler characteristic and any genus. Then, within this subset, the program searches for surfaces of a certain genus, and then a certain Euler characteristic. A Klein bottle is nonorientable, genus 2, Euler characteristic 0.

A further use of the functions contained in this program may be to develop a computer program capable of determining all possible manifolds that can be generated from a given polygon, i.e., a pentagon, hexagon,dodecagon, etc.

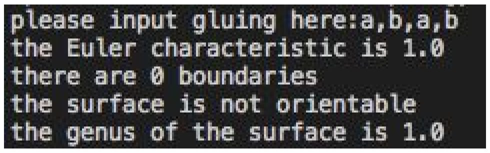
Examples:

Klein bottle



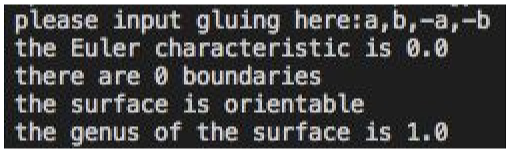
```
please input gluing here:a,b,a,-b
the Euler characteristic is 0.0
there are 0 boundaries
the surface is not orientable
the genus of the surface is 2.0
```

Projective Plane



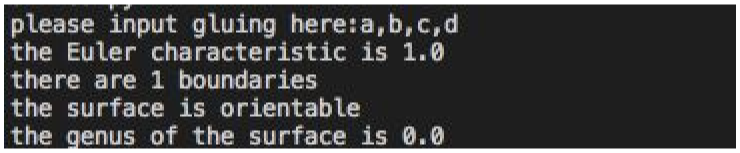
```
please input gluing here:a,b,a,b
the Euler characteristic is 1.0
there are 0 boundaries
the surface is not orientable
the genus of the surface is 1.0
```

1 Genus Torus



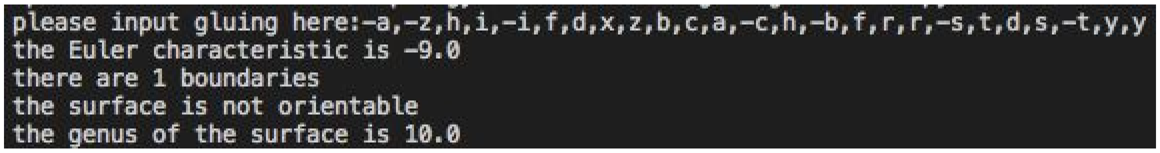
```
please input gluing here:a,b,-a,-b
the Euler characteristic is 0.0
there are 0 boundaries
the surface is orientable
the genus of the surface is 1.0
```

Filled-in circle



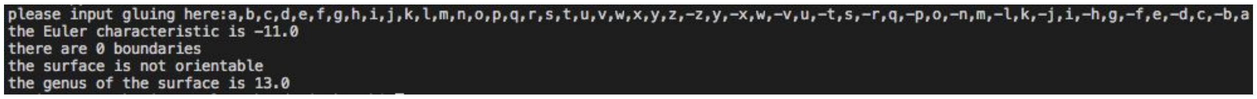
```
please input gluing here:a,b,c,d
the Euler characteristic is 1.0
there are 1 boundaries
the surface is orientable
the genus of the surface is 0.0
```

Non-orientable 10 genus surface with one boundary



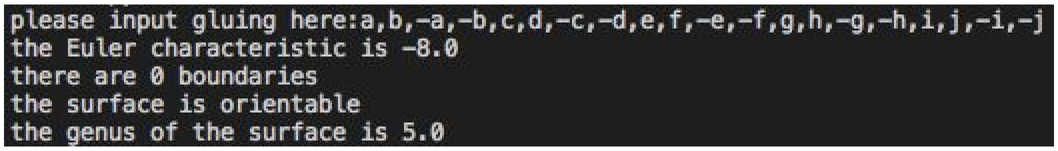
```
please input gluing here:-a,-z,h,i,-i,f,d,x,z,b,c,a,-c,h,-b,f,r,r,-s,t,d,s,-t,y,y
the Euler characteristic is -9.0
there are 1 boundaries
the surface is not orientable
the genus of the surface is 10.0
```

Non-orientable 13 genus surface with zero boundary



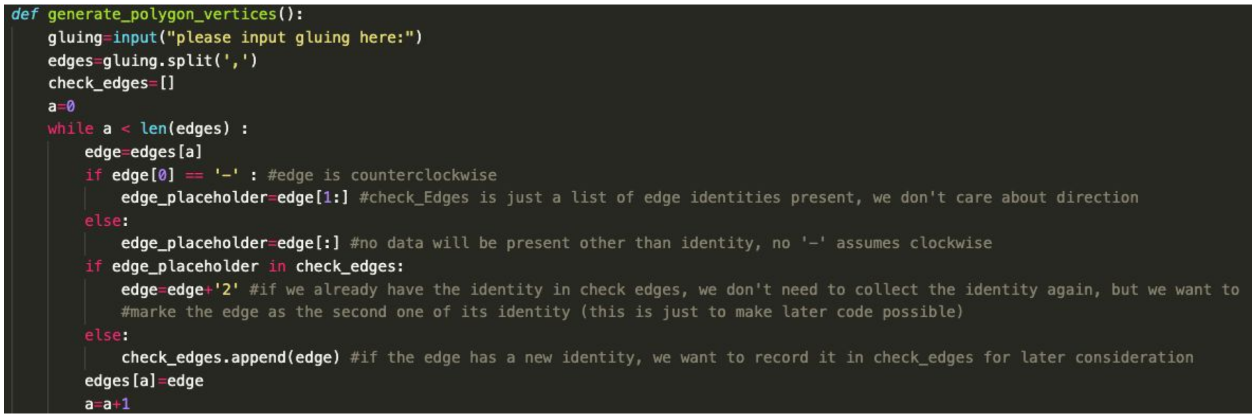
```
please input gluing here:a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z,-z,y,-x,w,-v,u,-t,s,-r,q,-p,o,-n,m,-l,k,-j,i,-h,g,-f,e,-d,c,-b,a
the Euler characteristic is -11.0
there are 0 boundaries
the surface is not orientable
the genus of the surface is 13.0
```

Orientable 5 torus



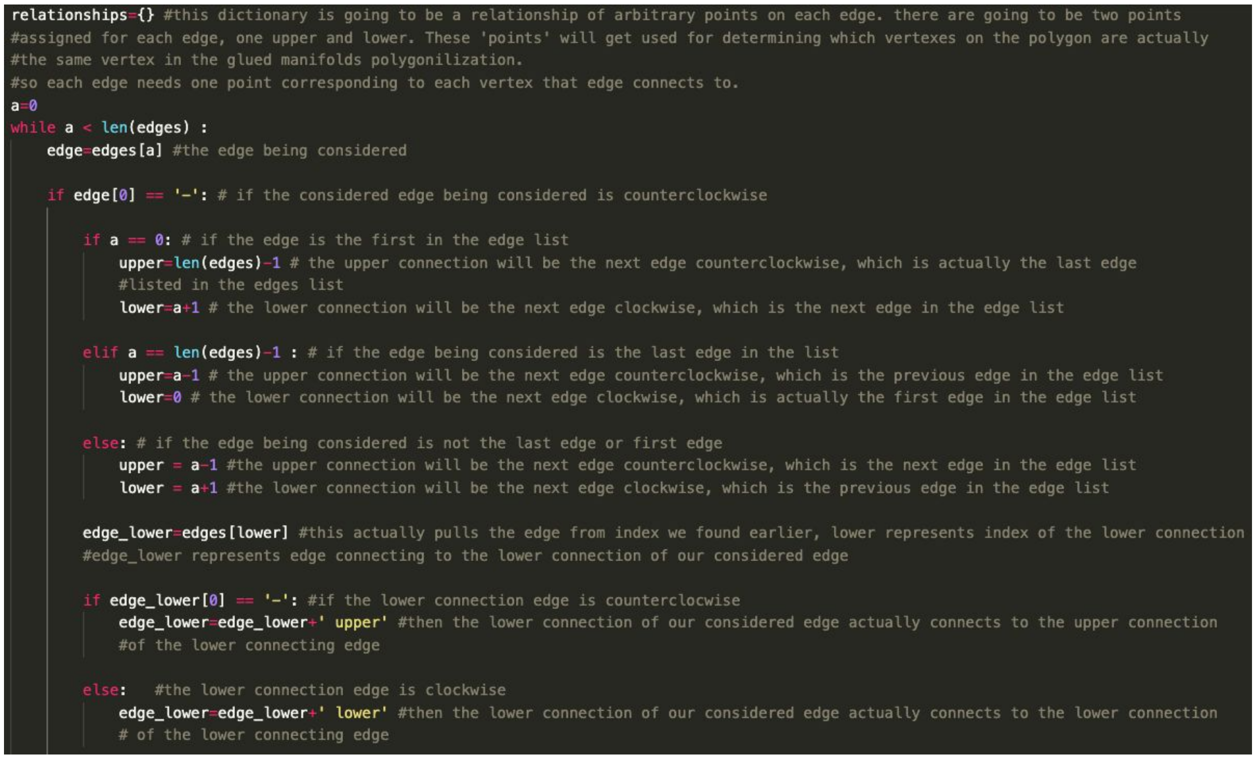
```
please input gluing here:a,b,-a,-b,c,d,-c,-d,e,f,-e,-f,g,h,-g,-h,i,j,-i,-j
the Euler characteristic is -8.0
there are 0 boundaries
the surface is orientable
the genus of the surface is 5.0
```

Appendix:



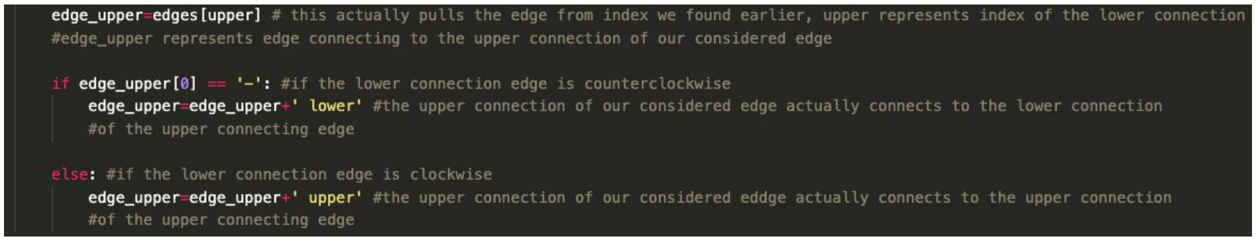
```python
def generate_polygon_vertices():
    gluing=input("please input gluing here:")
    edges=gluing.split(',')
    check_edges=[]
    a=0
    while a < len(edges) :
        edge=edges[a]
        if edge[0] == '-' : #edge is counterclockwise
            edge_placeholder=edge[1:] #check_Edges is just a list of edge identities present, we don't care about direction
        else:
            edge_placeholder=edge[:] #no data will be present other than identity, no '-' assumes clockwise
        if edge_placeholder in check_edges:
            edge=edge+'2' #if we already have the identity in check edges, we don't need to collect the identity again, but we want to
            #marke the edge as the second one of its identity (this is just to make later code possible)
        else:
            check_edges.append(edge) #if the edge has a new identity, we want to record it in check_edges for later consideration
        edges[a]=edge
        a=a+1
```

Image 1: generate_polygon_vertices (code #1)



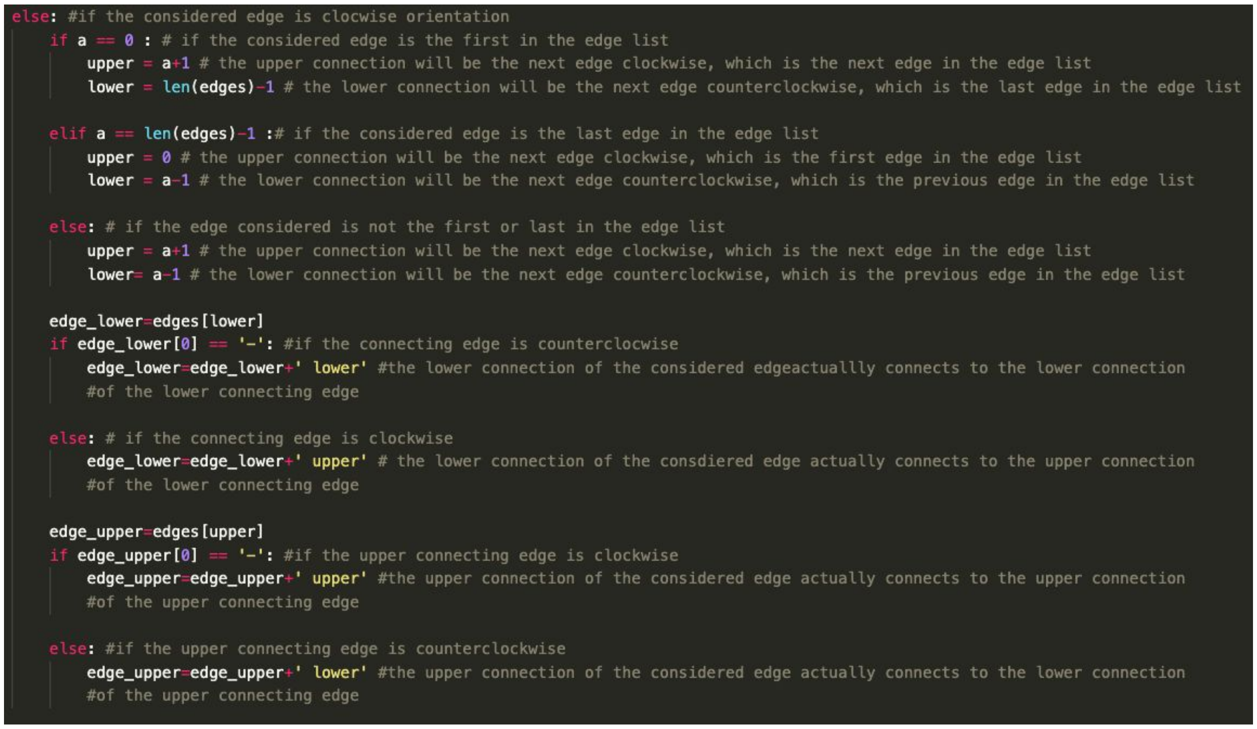
```python
relationships={} #this dictionary is going to be a relationship of arbitrary points on each edge. there are going to be two points
#assigned for each edge, one upper and lower. These 'points' will get used for determining which vertexes on the polygon are actually
#the same vertex in the glued manifolds polygonilization.
#so each edge needs one point corresponding to each vertex that edge connects to.
a=0
while a < len(edges) :
    edge=edges[a] #the edge being considered

    if edge[0] == '-': # if the considered edge being considered is counterclockwise

        if a == 0: # if the edge is the first in the edge list
            upper=len(edges)-1 # the upper connection will be the next edge counterclockwise, which is actually the last edge
            #listed in the edges list
            lower=a+1 # the lower connection will be the next edge clockwise, which is the next edge in the edge list

        elif a == len(edges)-1 : # if the edge being considered is the last edge in the list
            upper=a-1 # the upper connection will be the next edge counterclockwise, which is the previous edge in the edge list
            lower=0 # the lower connection will be the next edge clockwise, which is actually the first edge in the edge list

        else: # if the edge being considered is not the last edge or first edge
            upper = a-1 #the upper connection will be the next edge counterclockwise, which is the next edge in the edge list
            lower = a+1 #the lower connection will be the next edge clockwise, which is the previous edge in the edge list

        edge_lower=edges[lower] #this actually pulls the edge from index we found earlier, lower represents index of the lower connection
        #edge_lower represents edge connecting to the lower connection of our considered edge

        if edge_lower[0] == '-': #if the lower connection edge is counterclocwise
            edge_lower=edge_lower+' upper' #then the lower connection of our considered edge actually connects to the upper connection
            #of the lower connecting edge

        else:   #the lower connection edge is clockwise
            edge_lower=edge_lower+' lower' #then the lower connection of our considered edge actually connects to the lower connection
            # of the lower connecting edge
```

Image 2: generate_polygon_vertices (code #2)



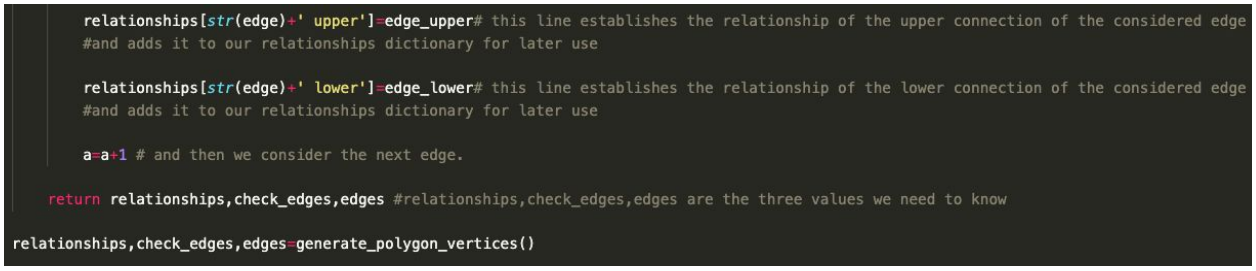
```
    edge_upper=edges[upper] # this actually pulls the edge from index we found earlier, upper represents index of the lower connection
    #edge_upper represents edge connecting to the upper connection of our considered edge

    if edge_upper[0] == '-': #if the lower connection edge is counterclockwise
        edge_upper=edge_upper+' lower' #the upper connection of our considered edge actually connects to the lower connection
        #of the upper connecting edge

    else: #if the lower connection edge is clockwise
        edge_upper=edge_upper+' upper' #the upper connection of our considered eddge actually connects to the upper connection
        #of the upper connecting edge
```

Image 3: generate_polygon_vertices (code #3)



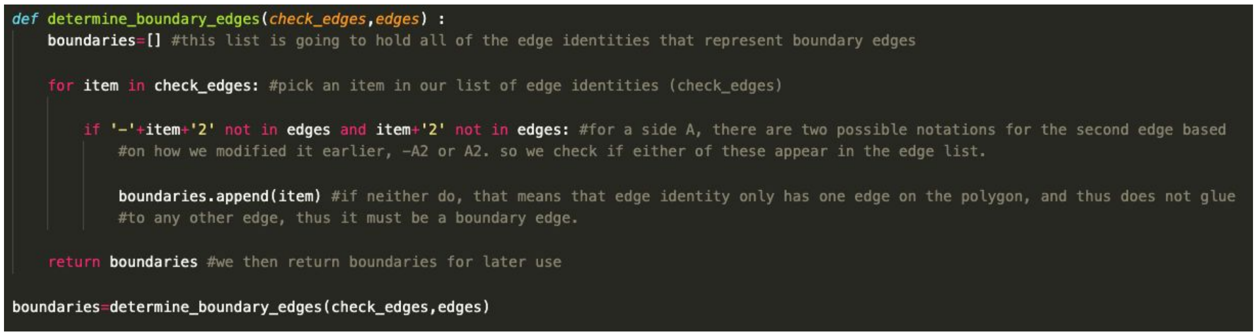
```
else: #if the considered edge is clocwise orientation
    if a == 0 : # if the considered edge is the first in the edge list
        upper = a+1 # the upper connection will be the next edge clockwise, which is the next edge in the edge list
        lower = len(edges)-1 # the lower connection will be the next edge counterclockwise, which is the last edge in the edge list

    elif a == len(edges)-1 :# if the considered edge is the last edge in the edge list
        upper = 0 # the upper connection will be the next edge clockwise, which is the first edge in the edge list
        lower = a-1 # the lower connection will be the next edge counterclockwise, which is the previous edge in the edge list

    else: # if the edge considered is not the first or last in the edge list
        upper = a+1 # the upper connection will be the next edge clockwise, which is the next edge in the edge list
        lower= a-1 # the lower connection will be the next edge counterclockwise, which is the previous edge in the edge list

    edge_lower=edges[lower]
    if edge_lower[0] == '-': #if the connecting edge is counterclocwise
        edge_lower=edge_lower+' lower' #the lower connection of the considered edgeactuallly connects to the lower connection
        #of the lower connecting edge

    else: # if the connecting edge is clockwise
        edge_lower=edge_lower+' upper' # the lower connection of the consdiered edge actually connects to the upper connection
        #of the lower connecting edge

    edge_upper=edges[upper]
    if edge_upper[0] == '-': #if the upper connecting edge is clockwise
        edge_upper=edge_upper+' upper' #the upper connection of the considered edge actually connects to the upper connection
        #of the upper connecting edge

    else: #if the upper connecting edge is counterclockwise
        edge_upper=edge_upper+' lower' #the upper connection of the considered edge actually connects to the lower connection
        #of the upper connecting edge
```

Image 4: generate_polygon_vertices (code #4)



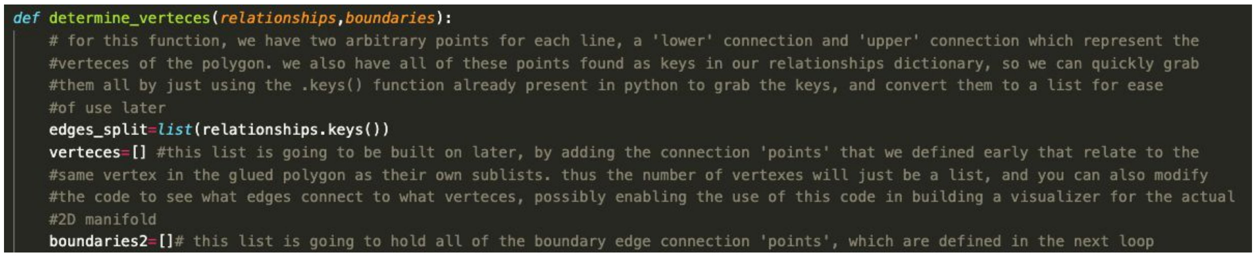
```
        relationships[str(edge)+' upper']=edge_upper# this line establishes the relationship of the upper connection of the considered edge
        #and adds it to our relationships dictionary for later use

        relationships[str(edge)+' lower']=edge_lower# this line establishes the relationship of the lower connection of the considered edge
        #and adds it to our relationships dictionary for later use

        a=a+1 # and then we consider the next edge.

    return relationships,check_edges,edges #relationships,check_edges,edges are the three values we need to know

relationships,check_edges,edges=generate_polygon_vertices()
```

Image 5: generate_polygon_vertices (code #5)



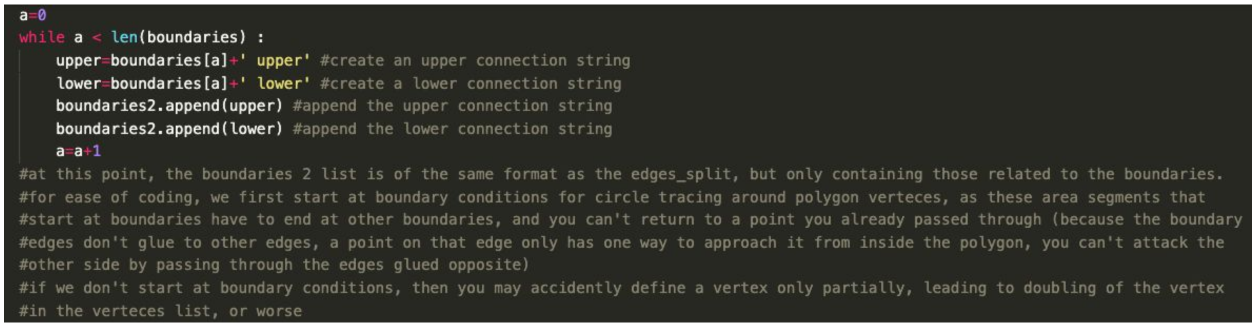
```python
def determine_boundary_edges(check_edges,edges) :
    boundaries=[] #this list is going to hold all of the edge identities that represent boundary edges

    for item in check_edges: #pick an item in our list of edge identities (check_edges)

        if '-'+item+'2' not in edges and item+'2' not in edges: #for a side A, there are two possible notations for the second edge based
            #on how we modified it earlier, -A2 or A2. so we check if either of these appear in the edge list.

            boundaries.append(item) #if neither do, that means that edge identity only has one edge on the polygon, and thus does not glue
            #to any other edge, thus it must be a boundary edge.

    return boundaries #we then return boundaries for later use

boundaries=determine_boundary_edges(check_edges,edges)
```

Image 6: determine_boundary_edges (code #1)



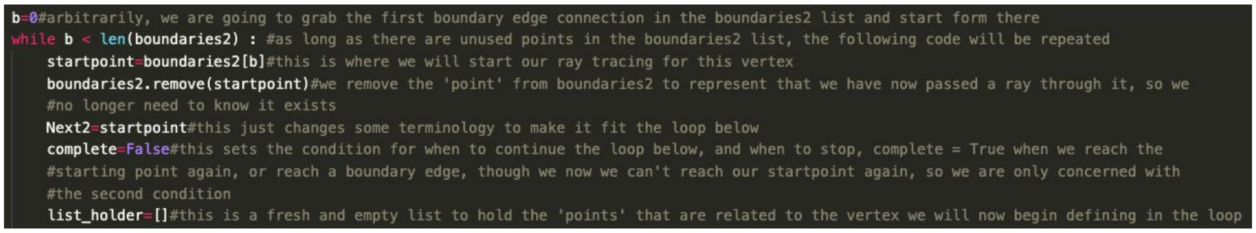
```python
def determine_verteces(relationships,boundaries):
    # for this function, we have two arbitrary points for each line, a 'lower' connection and 'upper' connection which represent the
    #verteces of the polygon. we also have all of these points found as keys in our relationships dictionary, so we can quickly grab
    #them all by just using the .keys() function already present in python to grab the keys, and convert them to a list for ease
    #of use later
    edges_split=list(relationships.keys())
    verteces=[] #this list is going to be built on later, by adding the connection 'points' that we defined early that relate to the
    #same vertex in the glued polygon as their own sublists. thus the number of vertexes will just be a list, and you can also modify
    #the code to see what edges connect to what verteces, possibly enabling the use of this code in building a visualizer for the actual
    #2D manifold
    boundaries2=[]# this list is going to hold all of the boundary edge connection 'points', which are defined in the next loop
```

Image 7: determine_vertices (code #1)



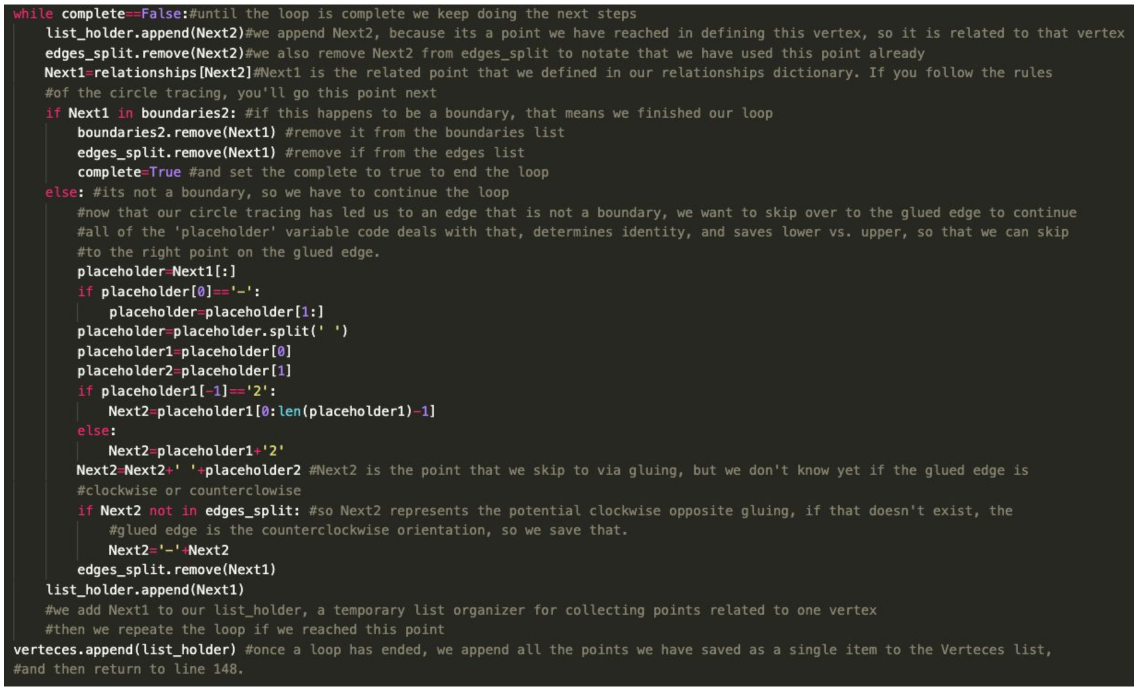
```python
a=0
while a < len(boundaries) :
    upper=boundaries[a]+' upper' #create an upper connection string
    lower=boundaries[a]+' lower' #create a lower connection string
    boundaries2.append(upper) #append the upper connection string
    boundaries2.append(lower) #append the lower connection string
    a=a+1
#at this point, the boundaries 2 list is of the same format as the edges_split, but only containing those related to the boundaries.
#for ease of coding, we first start at boundary conditions for circle tracing around polygon verteces, as these area segments that
#start at boundaries have to end at other boundaries, and you can't return to a point you already passed through (because the boundary
#edges don't glue to other edges, a point on that edge only has one way to approach it from inside the polygon, you can't attack the
#other side by passing through the edges glued opposite)
#if we don't start at boundary conditions, then you may accidently define a vertex only partially, leading to doubling of the vertex
#in the verteces list, or worse
```

Image 8: determine_vertices (code #2)



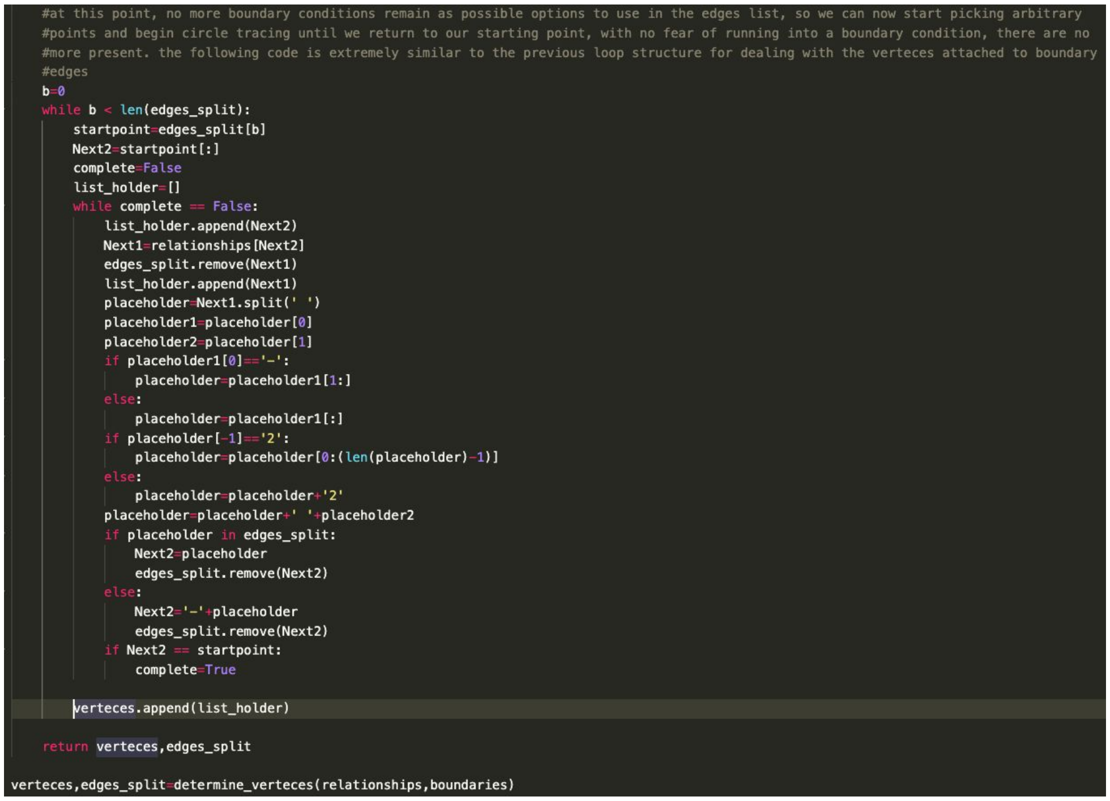
```python
b=0#arbitrarily, we are going to grab the first boundary edge connection in the boundaries2 list and start form there
while b < len(boundaries2) : #as long as there are unused points in the boundaries2 list, the following code will be repeated
    startpoint=boundaries2[b]#this is where we will start our ray tracing for this vertex
    boundaries2.remove(startpoint)#we remove the 'point' from boundaries2 to represent that we have now passed a ray through it, so we
    #no longer need to know it exists
    Next2=startpoint#this just changes some terminology to make it fit the loop below
    complete=False#this sets the condition for when to continue the loop below, and when to stop, complete = True when we reach the
    #starting point again, or reach a boundary edge, though we now we can't reach our startpoint again, so we are only concerned with
    #the second condition
    list_holder=[]#this is a fresh and empty list to hold the 'points' that are related to the vertex we will now begin defining in the loop
```

Image 9: determine_vertices (code #3)



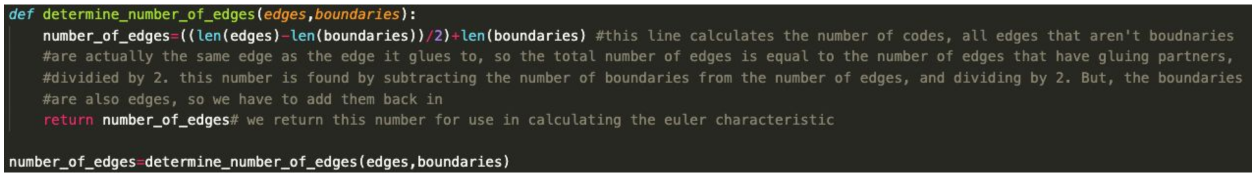
```
while complete==False:#until the loop is complete we keep doing the next steps
    list_holder.append(Next2)#we append Next2, because its a point we have reached in defining this vertex, so it is related to that vertex
    edges_split.remove(Next2)#we also remove Next2 from edges_split to notate that we have used this point already
    Next1=relationships[Next2]#Next1 is the related point that we defined in our relationships dictionary. If you follow the rules
    #of the circle tracing, you'll go this point next
    if Next1 in boundaries2: #if this happens to be a boundary, that means we finished our loop
        boundaries2.remove(Next1) #remove it from the boundaries list
        edges_split.remove(Next1) #remove if from the edges list
        complete=True #and set the complete to true to end the loop
    else: #its not a boundary, so we have to continue the loop
        #now that our circle tracing has led us to an edge that is not a boundary, we want to skip over to the glued edge to continue
        #all of the 'placeholder' variable code deals with that, determines identity, and saves lower vs. upper, so that we can skip
        #to the right point on the glued edge.
        placeholder=Next1[:]
        if placeholder[0]=='-':
            placeholder=placeholder[1:]
        placeholder=placeholder.split(' ')
        placeholder1=placeholder[0]
        placeholder2=placeholder[1]
        if placeholder1[-1]=='2':
            Next2=placeholder1[0:len(placeholder1)-1]
        else:
            Next2=placeholder1+'2'
        Next2=Next2+' '+placeholder2 #Next2 is the point that we skip to via gluing, but we don't know yet if the glued edge is
        #clockwise or counterclowise
        if Next2 not in edges_split: #so Next2 represents the potential clockwise opposite gluing, if that doesn't exist, the
            #glued edge is the counterclockwise orientation, so we save that.
            Next2='-'+Next2
        edges_split.remove(Next1)
    list_holder.append(Next1)
    #we add Next1 to our list_holder, a temporary list organizer for collecting points related to one vertex
    #then we repeate the loop if we reached this point
verteces.append(list_holder) #once a loop has ended, we append all the points we have saved as a single item to the Verteces list,
#and then return to line 148.
```

Image 10: determine_vertices (code #4)



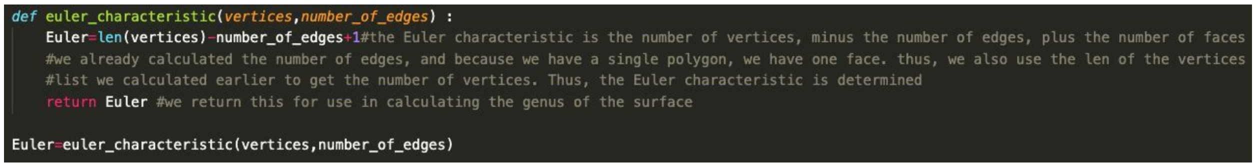
```
    #at this point, no more boundary conditions remain as possible options to use in the edges list, so we can now start picking arbitrary
    #points and begin circle tracing until we return to our starting point, with no fear of running into a boundary condition, there are no
    #more present. the following code is extremely similar to the previous loop structure for dealing with the verteces attached to boundary
    #edges
    b=0
    while b < len(edges_split):
        startpoint=edges_split[b]
        Next2=startpoint[:]
        complete=False
        list_holder=[]
        while complete == False:
            list_holder.append(Next2)
            Next1=relationships[Next2]
            edges_split.remove(Next1)
            list_holder.append(Next1)
            placeholder=Next1.split(' ')
            placeholder1=placeholder[0]
            placeholder2=placeholder[1]
            if placeholder1[0]=='-':
                placeholder=placeholder1[1:]
            else:
                placeholder=placeholder1[:]
            if placeholder[-1]=='2':
                placeholder=placeholder[0:(len(placeholder)-1)]
            else:
                placeholder=placeholder+'2'
            placeholder=placeholder+' '+placeholder2
            if placeholder in edges_split:
                Next2=placeholder
                edges_split.remove(Next2)
            else:
                Next2='-'+placeholder
                edges_split.remove(Next2)
            if Next2 == startpoint:
                complete=True

        verteces.append(list_holder)

    return verteces,edges_split

verteces,edges_split=determine_verteces(relationships,boundaries)
```

Image 11: determine_vertices (code #5)



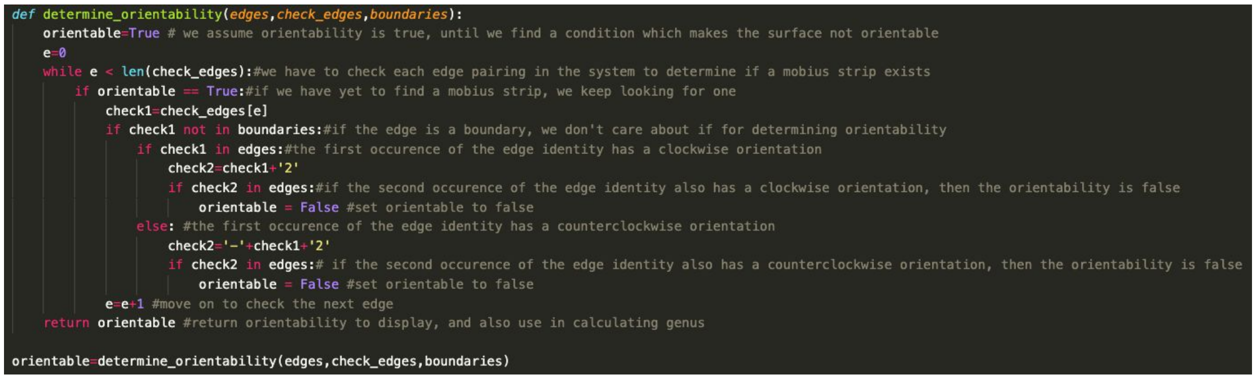
```python
def determine_number_of_edges(edges,boundaries):
    number_of_edges=((len(edges)-len(boundaries))/2)+len(boundaries) #this line calculates the number of codes, all edges that aren't boudnaries
    #are actually the same edge as the edge it glues to, so the total number of edges is equal to the number of edges that have gluing partners,
    #dividied by 2. this number is found by subtracting the number of boundaries from the number of edges, and dividing by 2. But, the boundaries
    #are also edges, so we have to add them back in
    return number_of_edges# we return this number for use in calculating the euler characteristic

number_of_edges=determine_number_of_edges(edges,boundaries)
```

Image 12: determine_number_of_edges (code #1)



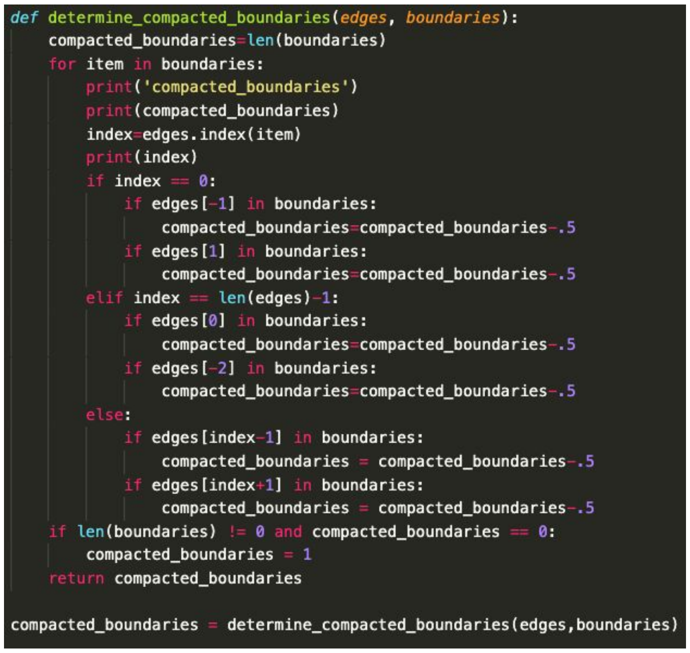
```python
def euler_characteristic(vertices,number_of_edges) :
    Euler=len(vertices)-number_of_edges+1#the Euler characteristic is the number of vertices, minus the number of edges, plus the number of faces
    #we already calculated the number of edges, and because we have a single polygon, we have one face. thus, we also use the len of the vertices
    #list we calculated earlier to get the number of vertices. Thus, the Euler characteristic is determined
    return Euler #we return this for use in calculating the genus of the surface

Euler=euler_characteristic(vertices,number_of_edges)
```

Image 13: euler_characteristic (code #1)



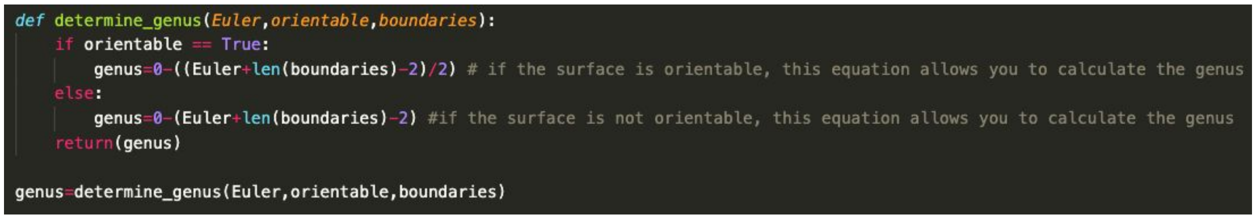
```python
def determine_orientability(edges,check_edges,boundaries):
    orientable=True # we assume orientability is true, until we find a condition which makes the surface not orientable
    e=0
    while e < len(check_edges):#we have to check each edge pairing in the system to determine if a mobius strip exists
        if orientable == True:#if we have yet to find a mobius strip, we keep looking for one
            check1=check_edges[e]
            if check1 not in boundaries:#if the edge is a boundary, we don't care about if for determining orientability
                if check1 in edges:#the first occurence of the edge identity has a clockwise orientation
                    check2=check1+'2'
                    if check2 in edges:#if the second occurence of the edge identity also has a clockwise orientation, then the orientability is false
                        orientable = False #set orientable to false
                else: #the first occurence of the edge identity has a counterclockwise orientation
                    check2='-'+check1+'2'
                    if check2 in edges:# if the second occurence of the edge identity also has a counterclockwise orientation, then the orientability is false
                        orientable = False #set orientable to false
            e=e+1 #move on to check the next edge
    return orientable #return orientability to display, and also use in calculating genus

orientable=determine_orientability(edges,check_edges,boundaries)
```

Image 14: determine_orientability (code #1)



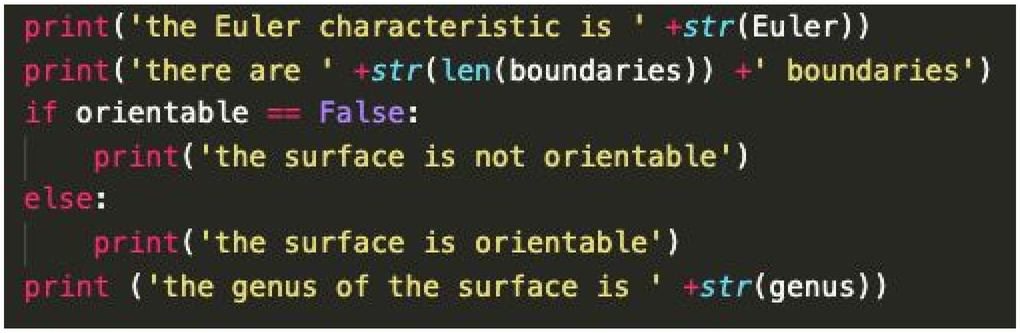
```python
def determine_compacted_boundaries(edges, boundaries):
    compacted_boundaries=len(boundaries)
    for item in boundaries:
        print('compacted_boundaries')
        print(compacted_boundaries)
        index=edges.index(item)
        print(index)
        if index == 0:
            if edges[-1] in boundaries:
                compacted_boundaries=compacted_boundaries-.5
            if edges[1] in boundaries:
                compacted_boundaries=compacted_boundaries-.5
        elif index == len(edges)-1:
            if edges[0] in boundaries:
                compacted_boundaries=compacted_boundaries-.5
            if edges[-2] in boundaries:
                compacted_boundaries=compacted_boundaries-.5
        else:
            if edges[index-1] in boundaries:
                compacted_boundaries = compacted_boundaries-.5
            if edges[index+1] in boundaries:
                compacted_boundaries = compacted_boundaries-.5
    if len(boundaries) != 0 and compacted_boundaries == 0:
        compacted_boundaries = 1
    return compacted_boundaries

compacted_boundaries = determine_compacted_boundaries(edges,boundaries)
```

Image 15: determine_compacted_boundaries (code #1)



```python
def determine_genus(Euler,orientable,boundaries):
    if orientable == True:
        genus=0-((Euler+len(boundaries)-2)/2) # if the surface is orientable, this equation allows you to calculate the genus
    else:
        genus=0-(Euler+len(boundaries)-2) #if the surface is not orientable, this equation allows you to calculate the genus
    return(genus)

genus=determine_genus(Euler,orientable,boundaries)
```

Image 16: determine_genus (code #1)



```python
print('the Euler characteristic is ' +str(Euler))
print('there are ' +str(len(boundaries)) +' boundaries')
if orientable == False:
    print('the surface is not orientable')
else:
    print('the surface is orientable')
print ('the genus of the surface is ' +str(genus))
```

Image 17: final steps (code #1)