ME714 GENERATIVE DESIGN FOR DIGITAL MANUFACTURING

Mini Project I

Generated Layers for Fused Filament Fabrication

Ege Uğur Aguş
2096063

21 November 2021

1 Problem Definition

In this first mini project of the course, students will generate layers for the fabrication of parts that are implicitly modeled using a boundary curve and a set of lines. Students are supposed to write a Python script to visualize 3D models. The inputs of this script are listed as follows.

- Vertices of the boundary polygon with *n edges* are provided in a *.txt* file in CCW direction.
- Start and end points of the line segments are given in a separate *.txt* file along with their velocities *(mm/mm)* in the vertical axis. Signs *(+ or -)* of the velocities determine the direction of motion of these lines along the z axis. For instance, when a positive velocity is defined for a line, then it should move to the right (w.r.t an arrow between the start and end point of the line segment) in the upcoming layers.
- Layer thickness is in *mm*.
- Number of layers.

Once all these inputs are given to the script, 3D model should be visualized with the following restrictions.

- Only Numpy and Matplotlib libraries are allowed in addition to native Python commands. Other third party libraries are not allowed.
- Regardless of whether the line segment is inside the polygon or not, when the line segment is extended to infinity, the parts of the line inside the polygon should be seen in the layers to be drawn.

2 Code Review

Script of the project is shared in that section part by part with explanations of the parts.

- First import libraries that we need. We are going to use Matplotlib for visualization of the layers and Numpy for list operations.

```
import matplotlib.pyplot as plt
import matplotlib.colors as color
import numpy as np
```

- Then we need to define required functions while building layers. At first we need to read the input files. Thus, define the function *"file_to_list"* as follows,

```python
def file_to_list(path):
    '''
    Parameters
    ----------
    path : str
        Path of the input file.
    Returns
    -------
    A Numpy array that contains float values of the source file.
    '''
    file = open(path, "r") # Open the file
    text = file.read() # Read the file
    file.close() # Close the file

    text = text.replace("\n", ",") # Transform the string by
    text = text.replace("\t", ",") # replacing new lines, tabs
    text = text.replace(" ", ",")  # and spaces with comma
    # Create a list of floats and return
    return np.fromstring(text, dtype=float, sep=',')
```

- We are going to need a function that find intersection point of two lines each defined by given two points. We can use the formula giving at the *"https://mathworld.wolfram.com/Line-LineIntersection.html"* to calculate the x & y values of the intersection point. Thus, let us define *"intersection_point"* function as follows,

```python
def intersection_point(line1, line2):
    '''
    Parameters
    ----------
    line1 : tuple
        Contains the coordinates of 2 two points from a line
    line2 : tuple
        Contains the coordinates of 2 two points from a line
    Returns
    -------
    Coordinates of intersection point of the line1 and line2
    by using the formula given at
    "https://mathworld.wolfram.com/Line-LineIntersection.html"
    '''

    x_diff = (line1[0][0] - line1[1][0], line2[0][0] - line2[1][0])
    y_diff = (line1[0][1] - line1[1][1], line2[0][1] - line2[1][1])

    def det(x, y):
        """
        Returns the determinant of 2x2 matrix
        """
        return x[0] * y[1] - x[1] * y[0]

```

```
    denominator = det(x_diff, y_diff)
    if denominator == 0:              # Check if the lines are parallel
        return False,False            # Return False,False if so

    c = (det(*line1), det(*line2))    # Apply the formula
    x = det(c, x_diff) / denominator # at referenced link
    y = det(c, y_diff) / denominator
    return x, y
```

- We also need a function that checks if a point is on a line segment. A point P is on a line segment QR, if $|QR| = |QP| + |PR|$ Thus, let us define *"intersection_at_edge"* function as follows,

```
def intersection_at_edge(intersection,corner1,corner2):
    '''
    Parameters
    ----------
    intersection : tuple
        Contains the coordinates of a point
    corner1 : tuple
        Contains the coordinates of a point
    corner2 : tuple
        Contains the coordinates of a point
    Returns
    -------
    True if intersection point is on the line segment of
    corner1 to corner2, otherwise False
    '''
    x1,y1 = corner1          #Define x and y
    x2,y2 = corner2          #Values of the points
    x3,y3 = intersection
    # If x and y values of the intersection point
    # are not integers return False
    if type(x3) == bool or type(y3) == bool:
        return False

    corner_to_corner = np.sqrt((x2-x1)**2+(y2-y1)**2)
    corner1_to_intersection = np.sqrt((x3-x1)**2 + (y3-y1)**2)
    corner2_to_intersection = np.sqrt((x2-x3)**2+(y2-y3)**2)
    # Small errors might be observed even if |QR| = |QP|+|PR|
    # due to numerical methods. Thus define a threshold.
    th = 5e-10 # Define a threshold

    if th >= abs (corner_to_corner - corner1_to_intersection \
              - corner2_to_intersection):
        return True
    else:
        return False
```

- Now we can define the *Main* function that will perform the operation specified in the problem definition as follows,

```python
def Main(path_bl, path_lines, layer_thickness, number_of_layers):
    '''
    Parameters
    ----------
    path_bl : str
        Path or name of the file that contains the vertices of the boundary polygon
    path_lines : str
        Path or name of the file that contains the information of the line segments
    layer_thickness : int
        Layer thickness value in mm
    number_of_layers: int
        Number of layers value
    Returns
    -------
    None. Displays the object constructed layer by layer.
    '''
    current_layer = 0 # Define a variable to deposit the current layer

    ax = plt.axes(projection='3d') # Define axes for visualization
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')

    bl_coordinates = file_to_list(path_bl) # Read the vertices of the boundary polygon
    if len(bl_coordinates) == 0:     # Check given input file is not empty
        return print("Vertices of the boundary polygon are not defined.") # Return

    bl_corners = np.array_split(bl_coordinates, len(bl_coordinates)/2)  # List with vertex coors.
    if len(bl_corners) < 3 :     # Check if given number of vertices is larger than 2
        return print("Given number of vertices is not enough.") # Return
    bl_corners.append(bl_corners[0])     # Add first vertex as last index for iteration purposes
    bl_coordinates = np.append(bl_coordinates,[bl_coordinates[0],bl_coordinates[1]])
    extreme = bl_coordinates.max()*10    # Find the value that higher than all coors. of vertices

    line_coordinates = file_to_list(path_lines) # Read the line segment information
    if len(line_coordinates) == 0: # Check there are no line segments
        for i in range(number_of_layers):
            ax.plot3D(bl_coordinates[::2],bl_coordinates[1::2],current_layer,c='black')
            current_layer += layer_thickness
        return # If there are no line segments print only boundary polygon and return

    line_coordinates = np.array_split(line_coordinates, len(line_coordinates)/5) # List with line information

    for k in range(len(line_coordinates)): # For every distinct color needed
        spacing = 1/(len(line_coordinates)) # Find required spacing at hue
        line_color = np.array([[0+(spacing)*k,1,1]])  # Define a distinct value for every line in HSV
        line_color = color.hsv_to_rgb(line_color) # Convert color to RGB from HSV
        line_coordinates[k] = np.append(line_coordinates[k],line_color) # Add information to the line
```

```python
    for i in range(number_of_layers): # Start iteration for every single layer
        ax.plot3D(bl_coordinates[::2],bl_coordinates[1::2],current_layer,c='black') # Draw BL polygon
                                                                                     # fix BL color to black
        for line in line_coordinates: # Start iteration for every single line segment
            intersection_points =[] # Define lists to deposit needed data
            x_founded = []
            y_founded = []
            draw = line.copy()  # Copy line so manipulations do not change the original data
            line_color, draw = draw[-3:], draw[:-3] # Separate color information

            for i in range(1,len(bl_corners)): # Find the intersection points of the line that contains the
            # line segment and lines that contain the edges of the polygon
                line1 = ((bl_corners[i-1][0],bl_corners[i-1][1]),(bl_corners[i][0],bl_corners[i][1]))
                line2 = ((draw[0],draw[1]),(draw[2],draw[3]))
                x,y = intersection_point(line1,line2)
                intersection_points.append((x,y))

            intersection_points.append(intersection_points[0])
            for i in range(1,len(intersection_points)): # Check if the intersection point is on an edge
            # of the polygon
                if intersection_at_edge(intersection_points[i-1],bl_corners[i-1],bl_corners[i]):
                    x_founded.append(intersection_points[i-1][0])
                    y_founded.append(intersection_points[i-1][1])

            if len(x_founded) > 2:  # If the number of intersection points is larger than 2 arrange
            # the values
                x_founded , y_founded = list(map(list, zip(*list(dict.fromkeys(list(zip(x_founded,\
                y_founded)))))))
                arrange_values = list(zip(x_founded,y_founded))
                arrange_values.sort(key = lambda x:x[0])
                x_founded , y_founded = list(map(list,zip(*arrange_values)))

            if len(x_founded) == 2: # If the number of intersection points
                if any(([x_founded[0],y_founded[0]] == x).all() for x in bl_corners) and \
                any(([x_founded[1],y_founded[1]] == x).all() for x in bl_corners):
                    pass
                else: # Draw line segment if start or end points of the line segment is not
                # belong to any edges
                    ax.plot3D(x_founded,y_founded,current_layer,c=line_color)

            elif len(x_founded) > 2:
                x_founded.append(x_founded[0])
                y_founded.append(y_founded[0])
                for counter1 in range(1,len(x_founded)-1):  # Find mid points of the line segments between
                # intersection points and obtain a horizontal line by using extreme value
                    multiple_segment_points = []
                    multiple_x_points = []
                    multiple_y_points = []
```

```
98                      mid_point_coor = ((x_founded[counter1-1]+x_founded[counter1])/2, \
99                      (y_founded[counter1-1]+y_founded[counter1])/2)
100                     line3 = ((mid_point_coor[0],mid_point_coor[1]),(extreme,mid_point_coor[1]))
101 
102                     for j in range(1,len(bl_corners)): # Find intersection points of the obtained line
103                     # and edges of the polygon
104                         line4 = ((bl_corners[j-1][0],bl_corners[j-1][1]), \
105                         (bl_corners[j][0],bl_corners[j][1]))
106                         x,y = intersection_point(line3,line4)
107                         multiple_segment_points.append((x,y))
108 
109                     multiple_segment_points.append([multiple_segment_points[0]])
110                     for m in range(1,len(multiple_segment_points)): # Check if the intersection points
111                     # are on the edge of the polygon
112                         if intersection_at_edge(multiple_segment_points[m-1], \
113                         bl_corners[m-1],bl_corners[m]):
114                             multiple_x_points.append(multiple_segment_points[m-1][0])
115                             multiple_y_points.append(multiple_segment_points[m-1][1])
116 
117                     points_at_right = [x for x in multiple_x_points if x > mid_point_coor[0]]
118                     if len(points_at_right) % 2 == 1: # Mid point is inside the polygon
119                         ax.plot3D([x_founded[counter1-1],x_founded[counter1]], \
120                         [y_founded[counter1-1],y_founded[counter1]],current_layer,c=line_color)
121             # Find the displacement of the star and end points of the line segment
122             teta  = np.arctan2(line[3]-line[1],line[2]-line[0])
123             y_displacement = np.cos(teta)*(line[4]*layer_thickness)
124             x_displacement = np.sin(teta)*(line[4]*layer_thickness)
125             for i in range(2): # Shift the line segment
126                 line[i*2] += x_displacement
127                 line[i*2+1] -= y_displacement
128 
129         current_layer += layer_thickness # Update current layer's Z value
```

- Let's investigate the *Main* function.
  - **Line** 17: We defined a variable to deposit the current layer
  - **Lines** 19-22: We defined Three-Dimensional Coordinate Systems and label the axes for the visualization purposes.
  - **Lines** 24-26: Read the input file that contains the vertices of the boundary polygon. If the input file is empty next operations going to *raise an error*. Thus, check if the input file is empty or not. If it is empty print a response and exit the function.
  - **Lines** 28-30: Arrange the list so that each index contains the information of a single vertex. Check the length of the arranged list and check its length. Since its length is the number of the edges, it must be greater than 2, since it will not form a polygon otherwise. If the length of the list is less than 3 print a response and exit the function.
  - **Lines** 31: Add first item of the first vertex information to the end of the list for iteration purposes at next lines.
  - **Lines** 32: Add first two item of the list to the end of the list, since drawing the boundary polygon we also need to draw the edge from n^{th} to 1^{st} vertex.

- **Lines** 33: Find an extreme value higher than every coordinate value of each vertex.
- **Lines** 35-40: Read the input file that contains line segments' information. If there is no given line segment print the boundary layer and exit the function. Arrange the list so that each index contains the information of a single vertex. Arrange the list so that each index contains the information of a single line segment.
- **Lines** 44-48: Line segments should be colored distinctly while observing the final figure for visual purposes. Picking random values from RGB color space will not work since (R,G,B) and (R,G,B-5) are not the same color for the Python, however they are almost same color for human eye. Thus, we need to work in the HSV color space since by distinct colors can be obtained by only varying Hue value when Saturation and Value values are fixed to maximum value. First, we need to find how many distinct colors that we need for line segments. Then, we should divide the interval of the Hue value to obtain distinct color for each line segment. Lastly, we need to add the color values to arrays to be able to draw line segments by their specific color at each layer.
- **Line** 50: Start the for loop for every layer.
- **Line** 51: Draw the boundary polygon. Since line segments cannot be colored black due to their color selection process, black is chosen as the color of the border polygon.
- **Lines** 53-58: Start the for loop for every line segment defined in the input files. Define some variables to deposit needed data. Copy the line to avoid not attempted manipulations. Divide color information of the line from the information array.
- **Lines** 60-65: Assume that line segment and edges of the boundary polygon are lines and find their intersection points by using *intersection_point* function. Add every intersection point to the *intersection_points* array.
- **Lines** 67-72: Add first element of the *intersection_points* to the end of the list for iteration purposes. Then check if any of the intersection points are on the edge of the polygon. Note that each intersection point is controlled only with the edge used when finding the intersection point. If anyone of the intersection points is on the edge of the polygon, then add its x and y values to the lists of *x_founded* and *y_founded*.
- **Lines** 76-77: If more than 2 intersection points are happen to be on the edges of the polygon. **Replica values** should be checked, since if one of the intersection points is also a vertex of the boundary polygon it is going to be founded at two edges and added two times to the *x_founded* and *y_founded* arrays. Thus, by using *zip(),list(), map() and dict.fromkeys() methods* eliminate the replica points.
- **Lines** 78-80: If more than 2 intersection points are happen to be on the edges of the polygon, **intersection points should be sorted according to their x values**, since algorithm used in this solution works by investigating consecutive line segments when more than 2 intersection points are found. Line segment between the first and last intersection points **does not have to be investigated**, if intersection points are sorted. Since that line segment would be the union of the consecutive line segments and for sure a part of it is at the outside of the boundary polygon.
- **Lines** 82-86: If only 2 intersection points are happen to be on the edges of the polygon and at least one of them is different from the vertices of the boundary polygon, draw the line segment.
- **Lines** 90-92: Start iteration for every line segment between founded intersection points.
- **Lines** 93-100: Find the mid point of the line segment between founded intersection points. Create a horizontal line that goes far more than the boundary polygon in the x-axis with the usage of the extreme value.
- **Lines** 102-115: Assume that created horizontal line and edges of the boundary polygon are lines and find their intersection points by using *intersection_point* function. Add every intersection point to the *multiple_segment_points* array. Then, check if any of the intersection points are on the edge of the polygon. Note that each intersection point is controlled only with the edge used when finding the intersection point. If anyone of the intersection points is on the edge of the polygon, then add its x and y values to the lists of *multiple_x_points* and *multiple_y_points*.
- **Line** 117: Find the intersection points that lies between midpoint of the line and the extreme.
- **Lines** 118-120: If the total number of intersection points are odd, then it means that midpoint is in the boundary polygon, so line segment also should be in the boundary polygon and vice versa. Draw the line segment if it is inside the boundary polygon.
- **Lines** 122-124: Find the total x and y displacement of the line's start and end points.
- **Lines** 125-127: Apply the displacement to the star and end points of the line for next layer.
- **Line** 129: Find next layer's z value in 3D coordinate axes.

3 Results

One can test the outputs of this project by importing the Python script and calling the *Main()* function with proper inputs. In this section some outputs of the script are shown.

- Basic Case
  - Vertices: [0 15] [15 15] [15 0] [0 0]
  - Line Segments: [0 7 0 8 2]
  - Layer Thickness: 0.1 mm
  - Number of Layers: 50

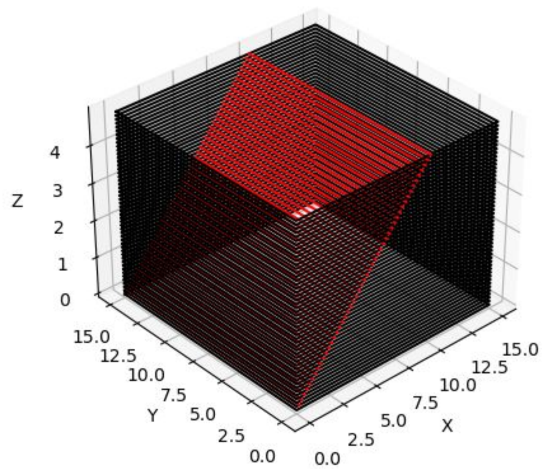

Figure 1: Basic Case Output

  - Note that at the first layer line segment is on the edge thus it should not be drew. Check that by making number of layers 3.

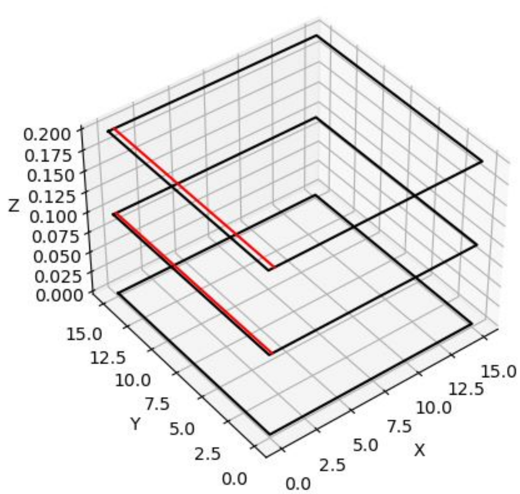

Figure 2: Basic Case Output With 3 Layers

- Basic Case with Increased Layer Thickness
  - Vertices: [0 15] [15 15] [15 0] [0 0]
  - Line Segments: [0 7 0 8 5]
  - Layer Thickness: 0.3 mm
  - Number of Layers: 50

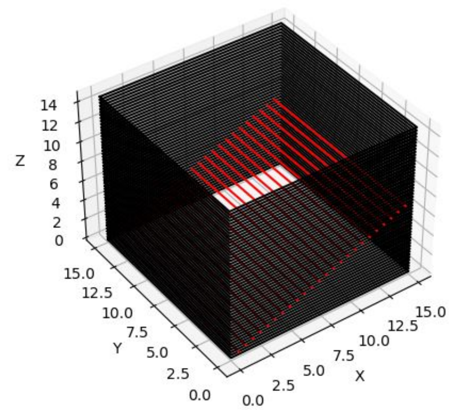

Figure 3: Basic Case with Increased Layer Thickness

  - Since layer thickness is increased total length at z-axis is increased and since line velocity is the same with the *Basic Case*, line segment left the boundary layer faster.

- Basic Case with Two Line Segments
  - Vertices: [0 15], [15 15], [15 0], [0 0]
  - Line Segments: [0 7 0 8 5], [10 5 2 7 -1]
  - Layer Thickness: 0.1 mm
  - Number of Layers: 50

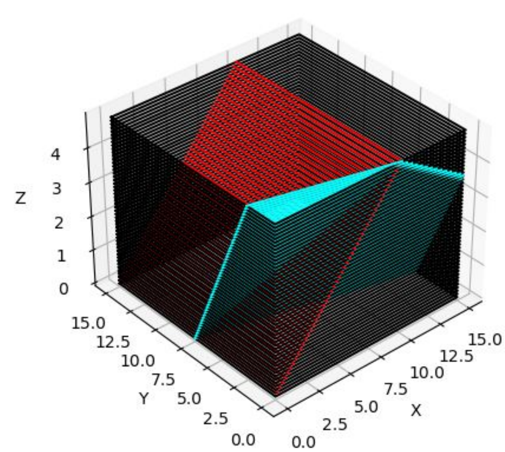

Figure 4: Basic Case with 2 Lines

- Complex Case with Two Line Segments
  - Vertices: [1 1], [5 5], [10 1], [10 10], [1 10]
  - Line Segments: [0 7 0 8 5], [10 5 2 7 -1]
  - Layer Thickness: 0.1 mm
  - Number of Layers: 50

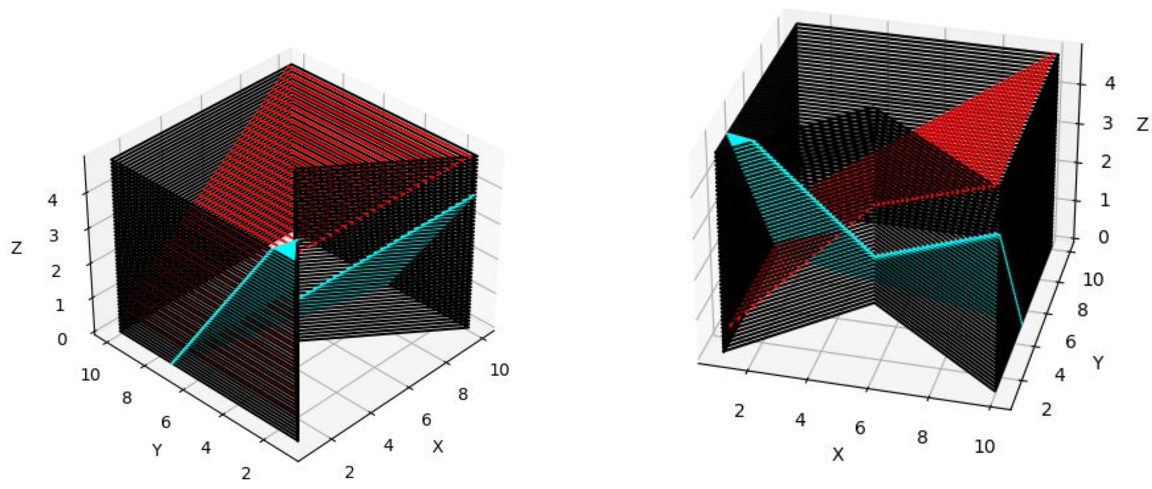

Figure 5: Complex Case with 2 Lines

- Complex Case One Edge & Line Segment are on the Same Line
  - Vertices: [0 0], [4 4], [4 5], [10 0], [10 10], [0 10]
  - Line Segments: [1 8 1 7 -1]
  - Layer Thickness: 1 mm
  - Number of Layers: 4

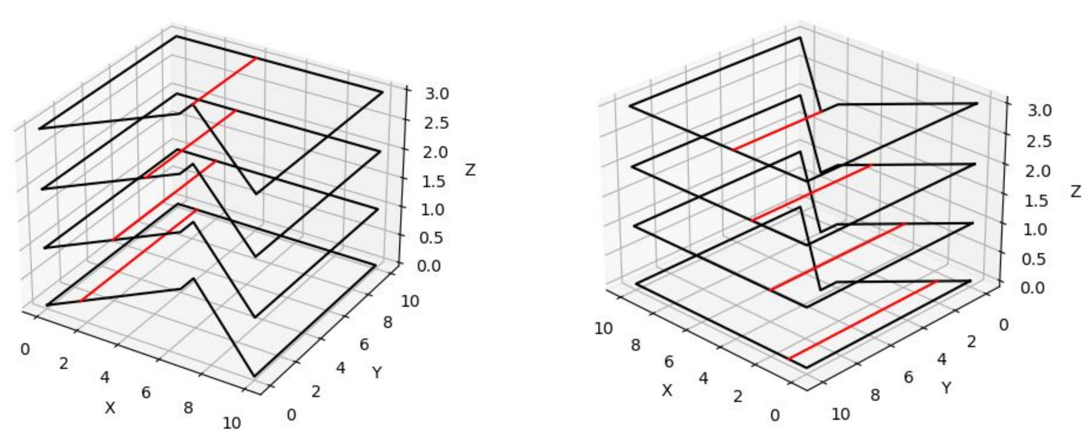

Figure 6: Complex Case One Edge & Line Segment are on the Same Line

    - At 4^{th} layer line segment and an edge are on the same line and script should not draw a line an the edge.

- Final Complex Case
    - Vertices: [0 0], [4 4], [4 5], [10 0],[25 3],[5 5],[25 7], [10 10], [5 25], [-5 10]
    - Line Segments: [0 7 0 8 5], [10 5 2 7 -1]
    - Layer Thickness: 0.1 mm
    - Number of Layers: 40

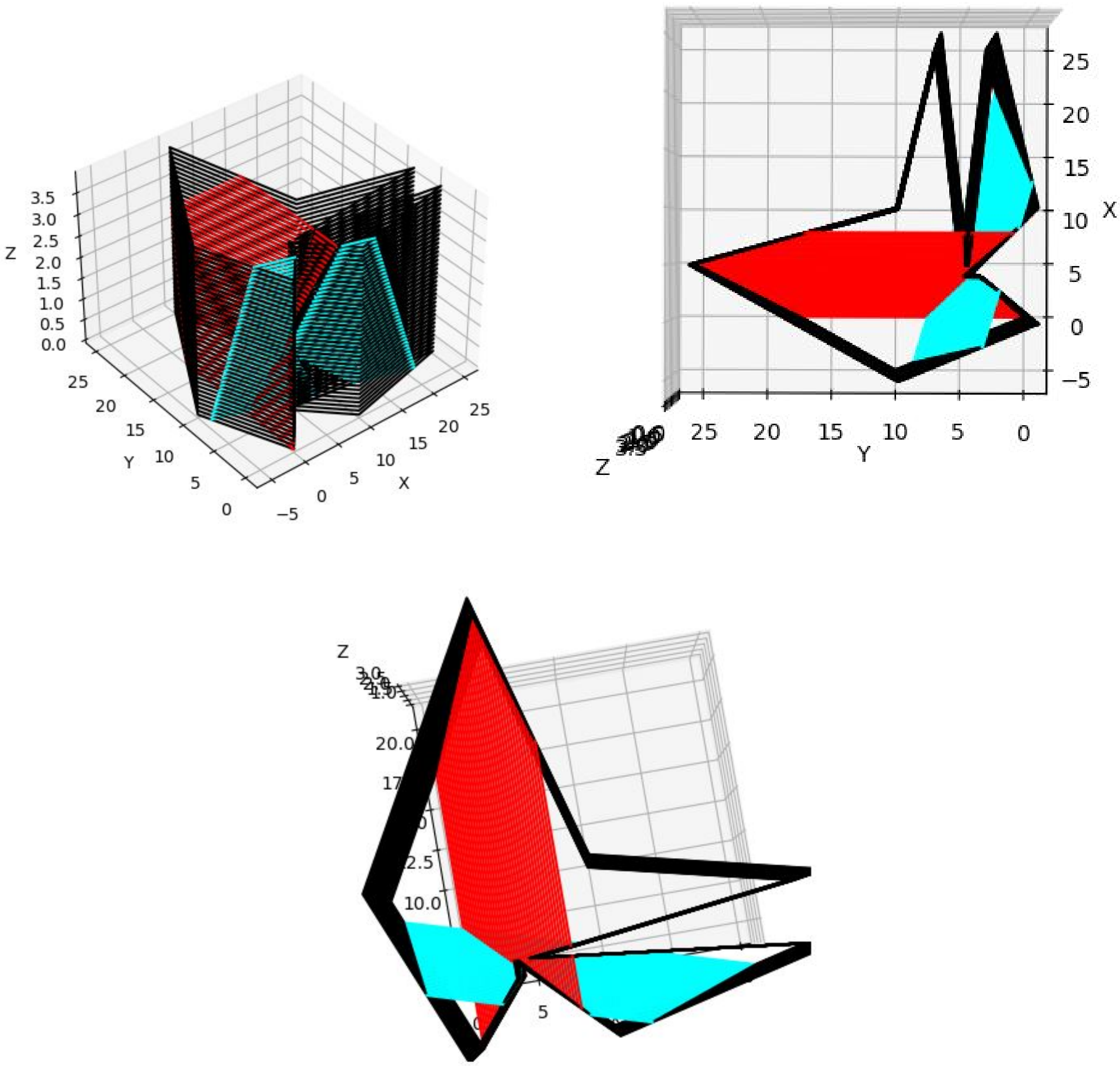

Figure 7: Final Complex Case