# ME714 GENERATIVE DESIGN FOR DIGITAL MANUFACTURING

## Mini Project II

### IMPLICIT MODELING AND SLICING FOR DIGITAL MANUFACTURING

Ege Uğur Aguş
2096063

26 December 2021

## 1 Problem Definition

In this second mini project of the course, students will generate layers for the fabrication of parts filled with triply periodic minimal surfaces (TPMS) type of lattice structures. Outer surface of the parts are explicitly defined with STL files and the interior is implicitly defined by the type of TPMS structure and its unit size. Students are supposed to write a Python script to generate slices of the given parts to be fabricated on desktop 3D printers. Path on any layer should be as continuous as possible. The inputs of the script are listed as follows.

- STL file of the part
- Type of TPMS lattice structure
- Unit size of TPMS lattice structures in *mm*.
- Layer thickness in *mm*
- Number of shells around the boundary

Only Shapely, Numpy, Numpy-stl, Scipy and Matplotlib libraries are allowed in addition to native Python commands.

## 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.

```
import numpy as np
import math
from stl import mesh
import matplotlib.pyplot as plt
from shapely.geometry import Polygon, LineString
from shapely.geometry import Point as ShpPoint
```

- First, let us investigate the the object classes used in this project.*Point()* class is defined to be able to create objects that stores the coordinates of the respective points.*Plane()* class is defined the store every feature of the layers that are going to be created for 3D printing. Every plane represents the specific layer. *facet()* class is defined to represents the triangular faces in the given *STL* file.

```python
class Point():
    """
    Point class that represents the points.
    """

    def __init__(self,x,y,z):
        self.x = x
        self.y = y
        self.z = z

    def __eq__(self, other):

        return self.x == other.x and self.y == other.y and self.z == other.z


class Plane():
    """
    Plane class that represents the layers for the 3D printing process. Every information
    about the layer is stored in this class.
    """
    plane_id = 0

    def __init__(self,z):
        self.z = z
        self.bl_lines = []
        self.edge_vertex_list = []
        self.inline = []
        self.bl_poly = []
        self.shells = []
        self.inner_bl = []
        self.id = Plane.plane_id
        Plane.plane_id +=1

class facet():
    """
    Face class that represents the triangular faces of the given stl file.
    """
    face_id = 0

    def __init__(self,v0,v1,v2):
        self.v0 = v0
        self.v1 = v1
        self.v2 = v2
        self.id = facet.face_id
        facet.face_id += 1

    def _floor(self):
        return min(self.v0[2], self.v1[2], self.v2[2])
```

```
    def _ceil(self):
        return max(self.v0[2], self.v1[2], self.v2[2])
```

- Now we can investigate the supplementary functions. *find_in_list_of_list()* function is defined to check if the vertex, an specified object of the *Point()* class, exits in a list or not. *check_point()* function is defined to check if an *item* exists in a list of *edges*. *edge_mid_point()* function is defined to find the mid point of an edge.

```
def find_in_list_of_list(mylist, vertex):
    """
    A  function that search for a vertex, an object of the Point class, in a nested list.
    If one of the points in the nested list are closer than the threshold value to the
    vertex, then function returns the index of the founded point.
    """

    th = 1e-3
    for sub_list in mylist:
        for item in sub_list:
            if abs(vertex[0]-item[0]) <= th and abs(vertex[1]-item[1]) <= th:
                return (mylist.index(sub_list), sub_list.index(item))

    return False


def check_point(edges, item):
    """
    A function that checks if item exits in the edges or not.
    """
    if len(edges) == 0:
        return True

    th = 1e-3
    reverse_item = item.copy()
    reverse_item.reverse()

    for edge in edges:

        if abs(edge[0][0]-item[0][0]) <= th and abs(edge[0][1]-item[0][1]) <= th and abs(edge[1][0]-item[1][0]) <= th
            and   abs(edge[1][1]-item[1][1]) <= th:
                return False

        if abs(edge[0][0]-reverse_item[0][0]) <= th and abs(edge[0][1]-reverse_item[0][1]) <= th and abs(edge[1][0]-r
            and  abs(edge[1][1]-reverse_item[1][1]) <= th:
                return False

    return True

def edge_mid_point(p1,p2,height):
    """
```

```
    A function returns the mid point of an edge
    """

    rate = (p1.z-height)/(p1.z-p2.z)
    x = p1.x + (p2.x-p1.x)*rate
    y = p1.y + (p2.y-p1.y)*rate

    return x,y
```

- *shells()* function is defined to find inner shell contours of the boundary polygons for every layer. *parallel_offset()* function in the *Shapely* sometimes returns *MultiLineString* instead of *LineString* which is a bug causes because of one of the dependencies of *Shapely* library. Thus, *debug_multiline_string()* function is defined to handle this bug. When *MultiLineString* is obtained this function investigates the lines inside the *MultiLineString* and returns the meaningful line/lines as *LineString*.

```
def shells(Planes,shell_number, shell_thickness = 0.4):
    """
    Function that creates shells for the layers
    """
    for plane in Planes:
        if len(plane.bl_poly) != 0:
            for poly in plane.bl_poly:
                linear_ring = poly.boundary
                for i in range(1,shell_number+1):
                    offset_list = []
                    multiple = False
                    offset = linear_ring.parallel_offset(-shell_thickness*i, 'left', join_style=2)
                    if offset.geom_type == "MultiLineString": # Multilinestring returned
                        multiple = True
                        offset_list = debug_multiline_string(offset,poly)
                    else:
                        offset_list = [offset]

                    if multiple == False:
                        if poly.contains(offset_list[0]) == False: # Check if the offset is true
                            offset = linear_ring.parallel_offset(shell_thickness*i, 'left', join_style=2)
                            if offset.geom_type == "MultiLineString": # Multilinestring returned
                                offset_list = debug_multiline_string(offset,poly)
                            else:
                                offset_list = [offset]

                    elif multiple == True and len(offset_list) == 0:

                        offset = linear_ring.parallel_offset(shell_thickness*i, 'left', join_style=2)
                        if offset.geom_type == "MultiLineString": # Multilinestring returned
                            offset_list = debug_multiline_string(offset,poly)

                        else:
                            offset_list = [offset]
```

```
                    if len(offset_list) > 0:
                        for offset in offset_list:
                            plane.shells.append(list(offset.coords))
                            if i == shell_number:
                                inner_poly = Polygon([ (point[0], point[1]) for point in list(offset.coords)])
                                plane.inner_bl.append(inner_poly)

def debug_multiline_string(offset,poly):
    """
    Multiline string is a issue at Shapely while using 'parallel_offset' function. This funciton
    debugs when a multiline string returned from 'parallel_offset' function
    """
    output = []
    lines = list(offset.geoms)
    for line in lines:
        coordinates = list(line.coords)
        if len(coordinates) >= 3:
            x_diff = abs(coordinates[0][0]-coordinates[1][0])
            y_diff = abs(coordinates[0][1]-coordinates[1][1])
            norm = np.sqrt(x_diff**2+y_diff**2)
            if poly.contains(line) and norm > 1e-5:
                points = list(line.coords)
                points.append(points[0])
                shell = LineString(points)
                output.append(shell)

    return output
```

- *tpmsCreator()* funciton is created to find inline structure for every layer which are the corresponding slices of the *TPMS* structures inside the boundary polygons. This functions uses *axes.contour* objects from *Matplotlib*, which is an inefficient method to obtain inline structures, since these objects contain much more information than needed. One can do this operation more efficiently by obtaining implicit function's results for specified grid values and investigating the transitions between negative and positive values, since *zeros* are the searched results and they lie at those transitions.

```
def tpmsCreator(fn, limits, Planes, weight , tool_dia = 0.4, res = 1):

    """
    Function that find the contours of a given implicit function for every layer
    in the specified limits.
    """

    cset_list = []

    xmin, xmax, ymin, ymax, zmin, zmax = limits

    x_grids = int(((xmax-xmin)/(tool_dia*res))+1)
    y_grids = int(((ymax-ymin)/(tool_dia*res))+1)
```

```python

    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    A1, A2 = np.linspace(xmin, xmax, x_grids) , np.linspace(ymin, ymax, y_grids)
    B = np.linspace(zmin, zmax, len(Planes)) # number of slices
    X,Y = np.meshgrid(A1,A2)

    for z in B: # Create contours in the XY plane
        Z = fn(X,Y,z, weight)
        cset = ax.contour(X, Y, Z+z, [z], zdir='z')
        cset_list.append(cset)

    for plane in Planes:
        lines = cset_list[plane.id].allsegs[0]

        for line in lines:
            for poly in plane.inner_bl:
                inside_points = []
                for point in line:
                    if poly.contains(ShpPoint(point)):
                        inside_points.append(point)

                if  len(inside_points) > 1 :
                    plane.inline.append(inside_points)
```

- *gcodeWriter()* function is defined to create *G-Code* for 3D printing. Function is created for *Ultimaker 3 Extended*.The function generates the code for each layer in order of outer bounds, shells, and inline.

```python
def gcodeWriter(filename,Planes,layer_thickness):
    """
    Function that creates G-code files for Ultimaker Extended 3
    """

    extruderRatio = 0.04
    Z = 0.27
    extruder_pos = 0
    f = open(filename[:-3] + ".gcode",'w')
    counter = 0


    # Start of the G-Code
    f.write(";START_OF_HEADER\n;HEADER_VERSION:0.1\n;FLAVOR:Griffin\n")
    f.write(";TARGET_MACHINE.NAME: Ultimaker 3 Extended\n")
    f.write(";EXTRUDER_TRAIN.0.INITIAL_TEMPERATURE:205\n")
    f.write(";;EXTRUDER_TRAIN.0.NOZZLE.DIAMETER:0.4\n;EXTRUDER_TRAIN.0.NOZZLE.NAME:AA 0.4\n")
    f.write(";BUILD_PLATE.TYPE:glass\n;BUILD_PLATE.INITIAL_TEMPERATURE:60\n;END_OF_HEADER\n")
    f.write("T0\n")
    f.write("M82\n")
    f.write("G92 E0\n")
```

```
    f.write("M109 S205\n")

    for plane in Planes:
        if len(plane.bl_poly) > 0:
            f.write("\n;LAYER:%d\n"%(counter))
            for poly in plane.bl_poly: # Draw outer wall for the layer
                points = list(poly.exterior.coords)
                f.write(";Outer-Wall\n")
                f.write("G0 F9000 X%.3f Y%.3f Z%.3f\n"%(points[0][0],points[0][1],Z))
                for n in range(len(points)-1):
                    x_diff = points[n][0] - points[n+1][0]
                    y_diff = points[n][1] - points[n+1][1]
                    distance = np.sqrt(x_diff**2 + y_diff**2)
                    extruder_pos += distance * extruderRatio
                    f.write("G1 F2000 X%.3f Y%.3f E%.5f\n"%(points[n+1][0],points[n+1][1],extruder_pos))

            for shell in plane.shells:
                f.write("\n;Shells\n")
                if len(shell) > 0:
                    f.write("G0 F9000 X%.3f Y%.3f Z%.3f\n"%(shell[0][0],shell[0][1],Z))
                    for n in range(len(shell)-1):
                        x_diff = shell[n][0] - shell[n+1][0]
                        y_diff = shell[n][1] - shell[n+1][1]
                        distance = np.sqrt(x_diff**2 + y_diff**2)
                        extruder_pos += distance * extruderRatio
                        f.write("G1 F2000 X%.3f Y%.3f E%.5f\n"%(shell[n+1][0],shell[n+1][1],extruder_pos))

            for inline in plane.inline:
                f.write("\n;Inline\n")
                if len(inline) > 0:
                    f.write("G0 F9000 X%.3f Y%.3f Z%.3f\n"%(inline[0][0],inline[0][1],Z))
                    for n in range(len(inline)-1):
                        x_diff = inline[n][0] - inline[n+1][0]
                        y_diff = inline[n][1] - inline[n+1][1]
                        distance = np.sqrt(x_diff**2 + y_diff**2)
                        extruder_pos += distance * extruderRatio
                        f.write("G1 F2000 X%.3f Y%.3f E%.5f\n"%(inline[n+1][0],inline[n+1][1],extruder_pos))

        Z += layer_thickness
        counter += 1

    # End of the G-Code
    f.write("M140 S0\n")
    f.write("M107\n")
    f.write("M82\n")
    f.write("M104 S0\n")
    f.write("M104 T1 S0\n")
    f.write(";End of Gcode\n")
    f.write(';SETTING_3 {"extruder_quality": ["[general]\\nversion = 4\\nname = Fine #2\\ndef\n')
```

```
    f.write(";SETTING_3 inition = ultimaker3_extended\\n\\n[metadata]\\nquality_type = normal\n")
    f.write(";SETTING_3 \\nposition = 0\\ntype = quality_changes\\n\\n[values]\\nwall_thickne\n")
    f.write(';SETTING_3 ss = 10\\n\\n", "[general]\\nversion = 4\\nname = Fine #2\\ndefinitio\n')
    f.write(';SETTING_3 n = ultimaker3\\n\\n[metadata]\\nquality_type = normal\\nposition = 1\n')
    f.write(';SETTING_3 \\ntype = quality_changes\\n\\n[values]\\n\\n"], "global_quality": "[\n')
    f.write(';SETTING_3 general]\\nversion = 4\\nname = Fine #2\\ndefinition = ultimaker3_ext\n')
    f.write(';SETTING_3 ended\\n\\n[metadata]\\nquality_type = normal\\ntype = quality_change\n')
    f.write(';SETTING_3 s\\n\\n[values]\\nadhesion_type = none\\n\\n"}')
```

- *TPMS* structures are defined respectively. *function_returner()* function is defined to match the input name of the type of the *TPMS* structure with its specified function. New *TPMS* structures can be added to this project by defining their implicit function just like other defined *TPMS* structures and adding its key to *function_returner()* function. *w_finder()* function returns the *weight*, the multiplier for the sizing unit cell of the *TPMS* structure, value.

```
def schwarz(a,b,c,w=1):
    x, y, z = w*a, w*b, w*c
    return np.cos(x)+np.cos(y)+np.cos(z)

def gyroid(a,b,c,w=1):
    x, y, z = w*a, w*b, w*c
    return np.cos(x)*np.sin(y)+np.cos(y)*np.sin(z)+np.cos(z)*np.sin(x)

def double_gyroid(a,b,c,w=1):
    x, y, z = w*a, w*b, w*c
    return 2.75*(np.sin(2*x)*np.sin(z)*np.cos(y) + np.sin(2*y)*np.sin(x)*np.cos(z) \
                 + np.sin(2*z)*np.sin(y)*np.cos(x)) -\
            (np.cos(2*x)*np.cos(2*y) + np.cos(2*y)*np.cos(2*z) + np.cos(2*z)*np.cos(2*x))

def diamond(a,b,c,w=1):
    x, y, z = w*a, w*b, w*c
    return  np.sin(x)*np.sin(y)*np.sin(z) + np.sin(x)*np.cos(y)*np.cos(z) + \
        np.cos(x)*np.sin(y)*np.cos(z) + np.cos(x)*np.cos(y)*np.sin(z)

def iwp(a,b,c,w=1):
    x, y, z = w*a, w*b, w*c
    return np.cos(x)*np.cos(y) + np.cos(y)*np.cos(z) + np.cos(z)*np.cos(x) + 0.25

def w_finder(desired_size,default_size):

    return default_size / desired_size

def function_returner(tpms_type):

    if tpms_type == "schwarz":
        default_size = 6
        return schwarz, default_size

    if tpms_type == "gyroid":
        default_size = 6
```

```
        return gyroid, default_size

    if tpms_type == "diamond":
        default_size = 6
        return diamond, default_size

    if tpms_type == "ıwp":
        default_size = 6
        return ıwp, default_size
```

- Now we can investigate the *Main* function that will perform the operation specified in the problem definition.

```
def Main(stl_file_name, tpms_type, unit_size,shell_number, layer_thickness, tool_dia = 0.4, inline_res = 1, \
          render = True, render_res = 10, show_bl = True, show_shells = True, show_inline = True ):

    global facet
    global Plane
    global Point

    function , default_size = function_returner(tpms_type)
    weight = w_finder(unit_size, default_size)

    stl_mesh = mesh.Mesh.from_file(stl_file_name)
    triangles = stl_mesh.points

    max_global_x = max(max(stl_mesh.v0[:,0]), max(stl_mesh.v1[:,0]), max(stl_mesh.v2[:,0]))
    min_global_x = min(min(stl_mesh.v0[:,0]), min(stl_mesh.v1[:,0]), min(stl_mesh.v2[:,0]))

    max_global_y = max(max(stl_mesh.v0[:,1]), max(stl_mesh.v1[:,1]), max(stl_mesh.v2[:,1]))
    min_global_y = min(min(stl_mesh.v0[:,1]), min(stl_mesh.v1[:,1]), min(stl_mesh.v2[:,1]))

    max_global_z = max(max(stl_mesh.v0[:,2]), max(stl_mesh.v1[:,2]), max(stl_mesh.v2[:,2]))
    min_global_z = min(min(stl_mesh.v0[:,2]), min(stl_mesh.v1[:,2]), min(stl_mesh.v2[:,2]))

    facets = [ facet(triangle[0:3], triangle[3:6], triangle[6:9]) for triangle in triangles ]   # Define facets

    number_of_layers = int((max_global_z-min_global_z)/layer_thickness)     # Define number of layers
    Planes  = [Plane(min_global_z+(k*layer_thickness)) for k in range(number_of_layers)]    # Define planes

    for facet in facets:
        plane_to_slice = math.ceil((facet._floor()-min_global_z)/layer_thickness)
        while plane_to_slice <= Plane.plane_id -1:
            if Planes[plane_to_slice].z > facet._ceil():
                break

            verticies = [facet.v0,facet.v1,facet.v2]
            intersection = []
            check_points = [1 if verticies[i][2] > Planes[plane_to_slice].z else -1 if verticies[i][2] \
                                < Planes[plane_to_slice].z else 0 for i in range(3)]
```

```
            zeros = check_points.count(0)    # How many verticies are on the slicing plane

            if zeros == 0:    # None of the verticies are on the slicing plane
                for j in range(1,4):
                    if check_points[j-1] != check_points[j-2]:
                        p1 = Point(*verticies[j-1])
                        p2 = Point(*verticies[j-2])
                        x ,y = edge_mid_point(p1, p2, Planes[plane_to_slice].z)
                        intersection.append([x,y])

            elif zeros == 1:    # One vertex of the surface is on the slicing plane
                zeros_index = check_points.index(0)
                if check_points[(zeros_index+1)%3] != check_points[(zeros_index+2)%3]:
                    p1 = Point(*verticies[zeros_index])
                    p2 = Point(*verticies[(zeros_index+1)%3])
                    p3 = Point(*verticies[(zeros_index+2)%3])
                    x,y = edge_mid_point(p2, p3, Planes[plane_to_slice].z)

                    intersection.append([p1.x,p1.y])
                    intersection.append([x,y])

            elif zeros == 2:   # Edge of the surface is on the slicing plane
                zero_indicies = [i for i, x in enumerate(check_points) if x == 0]
                p1 = Point(*verticies[zero_indicies[0]])
                p2 = Point(*verticies[zero_indicies[1]])
                edge_line = [[p1.x,p1.y],[p2.x,p2.y]]

                if  (check_point(Planes[plane_to_slice].edge_vertex_list,edge_line)):
                        Planes[plane_to_slice].edge_vertex_list.append(edge_line)
                        intersection.append([p1.x,p1.y])
                        intersection.append([p2.x,p2.y])

            else:
                pass

            if len(intersection) > 0:
                Planes[plane_to_slice].bl_lines.append(intersection)
            plane_to_slice += 1

    for plane in Planes:
        lines = plane.bl_lines
        if len(lines) >= 3:
            while len(lines) > 0:
                loop = []
                current_line = lines.pop(0)
                for point in current_line:
                    loop.append(point)

                searching_point = current_line[1]
```

```
            while True: # Construct the loop
                index = find_in_list_of_list(lines,searching_point)  # Check next element of the loop

                if index == False:  # Loop is ended
                    if len(loop) > 3:
                    # print(plane.id)
                        poly = Polygon([ (point[0], point[1]) for point in loop])
                        plane.bl_poly.append(poly)
                    break

                else:
                    line = lines.pop(index[0])
                    line.pop(index[1])
                    line = line[0]
                    loop.append(line)

                    searching_point = line

limits = (min_global_x, max_global_x, min_global_y, max_global_y, min_global_z, max_global_z)

shells(Planes,shell_number,tool_dia)

tpmsCreator(function, limits, Planes, weight ,tool_dia , inline_res)

plt.cla()
plt.clf()

if render == True:
    ax = plt.axes(projection='3d')
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')

    if show_shells == True:
        l1 = [Planes[i] for i in range(1,len(Planes),render_res)]
        for plane in l1:
            for loop in plane.shells:
                X =[]
                Y =[]
                Z =[]
                for point in loop:
                    X.append(point[0])
                    Y.append(point[1])
                    Z.append(plane.z)

                ax.plot3D(X,Y,Z,'red')

    if show_bl ==  True:
```

```
136             l1 = [Planes[i] for i in range(1,len(Planes),render_res)]
137             for plane in l1:
138                 X =[]
139                 Y =[]
140                 Z =[]
141 
142                 for poly in plane.bl_poly:
143 
144                     x,y = poly.exterior.xy
145                     Z = [ plane.z for i in range(len(x))]
146 
147                     ax.plot3D(x,y,Z,'black')
148 
149         if show_inline == True:
150             l1 = [Planes[i] for i in range(1,len(Planes),render_res)]
151             for plane in l1:
152                 for loop in plane.inline:
153                     X =[]
154                     Y =[]
155                     Z =[]
156                     for point in loop:
157                         X.append(point[0])
158                         Y.append(point[1])
159                         Z.append(plane.z)
160 
161                     ax.plot3D(X,Y,Z,'green')
162 
163         xmin, xmax, ymin, ymax, zmin, zmax = limits
164 
165         ax.set_zlim3d(zmin,zmax)
166         ax.set_xlim3d(xmin,xmax)
167         ax.set_ylim3d(ymin,xmax)
168 
169         plt.show()
170 
171     gcodeWriter(stl_file_name,Planes,layer_thickness)
```

- Let's investigate the *Main* function.
  - **Lines 4-6**: Define *global* to be able to use classes inside *Main* function.
  - **Lines 8-9**: Find specified *TPMS* structure type and its multiplier for desired unit cell size.
  - **Lines 11-23**: Read the input *STL* file. Find global maximums and minimums for axes. Define *facets* to be able to slice them at the next steps.
  - **Lines 25-26**: Find how many layers there should be and create a *Plane* for each layer.
  - **Lines 28-75**: For each *facet* check its floor and ceiling $z$ value only planes that have a $z$ value between those values are going to slice that *facet*. Then for every plane that slice the specified *facet*, compare the $z$ values of the vertices of the *facet* and the z value of the plane. There are three possible intersection types. If none of the vertices of the *facet* is on the slicing plane, then simply find the intersection points between plane and edges of the edges of the *facet*. If one of the vertex of the *facet* is on the slicing plane, then check if other two vertices are on the same side of the slicing plane or not. If they are on the same side of the slicing plane,

then *facet* only slice plane at the one point and we can neglect their intersection. If not, the vertex on the slicing plane and the intersection point of the other edge of the *facet* and slicing plane are the vertices of the intersection edge. If two vertices of the *facet* are on the slicing plane than those vertices are the vertices of the intersection line, however same intersection edge can be observed for another *facet* too. Thus, check if it is found before, if not add them to the edge vertex list. *( After finishing the project, I realized that edges of the boundary polygons shares their vertices with two of the other edges, thus calculating every vertex of every edge was unnecessary, since after finding first edge's vertices we know the one vertex of the next edge. Thus, just keeping the facet vertices in the memory in this step is enough. We could use them instead of intersection edge vertices in the next step to decrease the number of operations. )*

- **Lines 77-104**: At previous step we found every intersection edge in planes. For a boundary polygon edges share a vertex, thus find the boundary polygons by checking their common vertices.
- **Lines 106-113**: Define limits since finding *TPMS* structures require high computational especially with high resolution. Thus, define the limits of the problem, since we do not need *TPMS* structures outside of those boundaries. By using respective functions obtain shells and *TPMS* structures. *tpmsCreator()* creates *plt.figure* objects which require high memory, clear them from memory.
- **Lines 115-163**: Rendering part of the *Main()* function. I added some extra inputs to be able to investigate outputs properly. One can observe outer boundary, shells and *TPMS* structures easily. Moreover, one can change the resolution for the render since rendering every layer will result with decrease in performance.
- **Line 171**: Call the function that generates g-code file of the object.

# 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
  - STL File: **DrilledCube.stl** a cube that is drilled
  - *TPMS* structure: *schwarz*
  - *TPMS* unit cell size: 18 mm
  - Layer thickness: 0.1 mm
  - Number of Shells: 3
  - function: *Main('DrilledCube.stl','schwarz',18,3,0.1,render_res=2)*

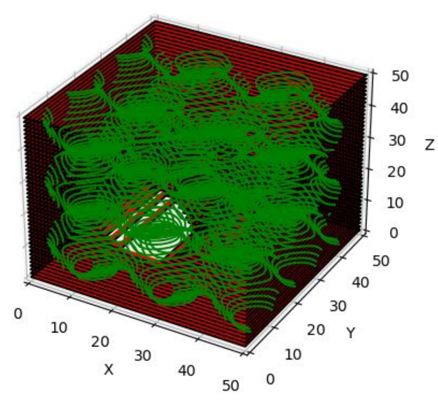


Figure 1: Basic Case Output (Render Resolution = 2)

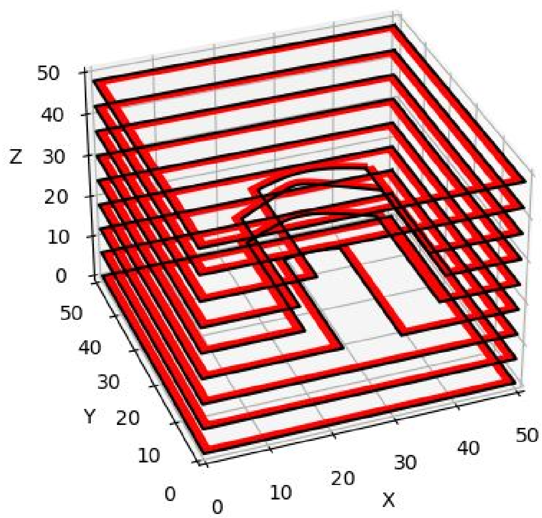


Figure 2: Basic Case Rendering with Hidden Inline (Render Resolution = 10)

- In Figure 1. black lines are the outer boundary, red lines are the shells and the green lines are the *TPMS* structures. In Figure 2. *TPMS* inlines are hidden for showing detailed boundary and shells. In Figure 3. 55th layer of the object is shown. It is clearly seen that there are 3 shells and *TPMS* structures start from most inner shell.

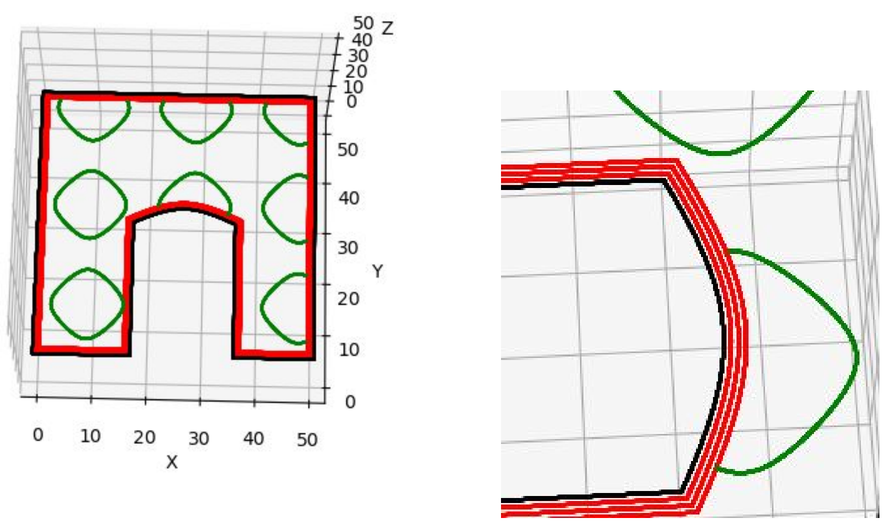


Figure 3: 55th Layer

- In Figure 1. black lines are the outer boundary, red lines are the shells and the green lines are the *TPMS* structures. In Figure 2. *TPMS* inlines are hidden for showing detailed boundary and shells. In Figure 3. 55th layer of the object is shown. It is clearly seen that there are 3 shells and *TPMS* structures start from most inner shell.

- By using *https://gcode.ws/*, I checked the g-code and it is shown as successful. In Figure 4. it can be observed.

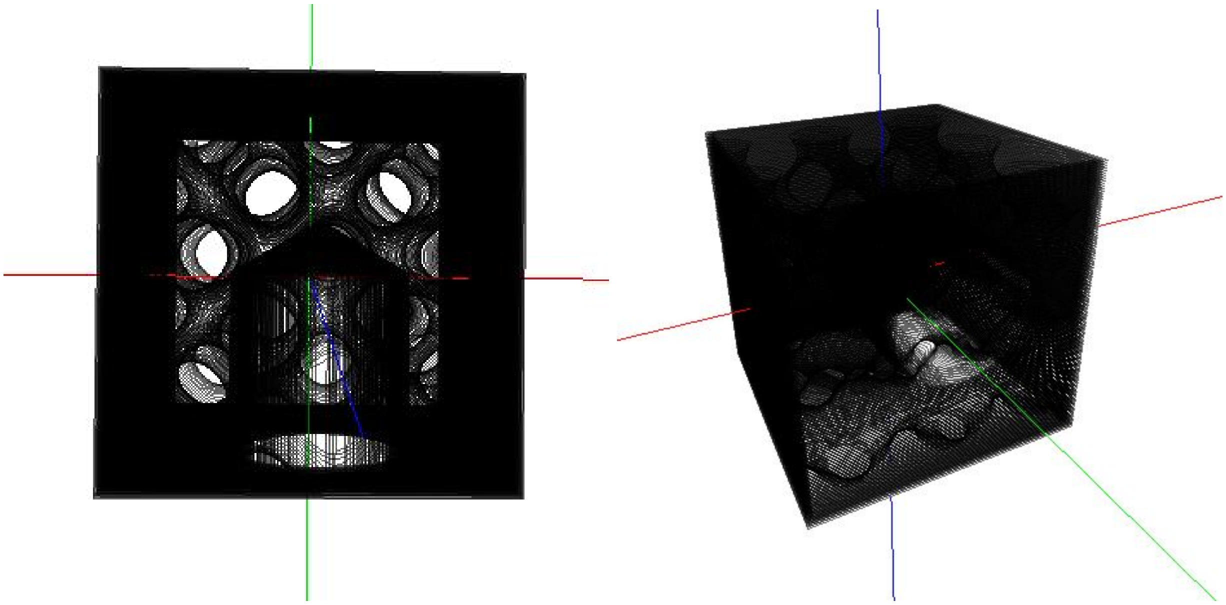

Figure 4: G-Code Visualization of Drilled Cube

- Complex Case: *Bunny*
  - STL File: **bunny.stl**. *STL* file of a *Bunny*
  - *TPMS* structure: *gyroid*
  - *TPMS* unit cell size: 36 mm
  - Layer thickness: 0.1 mm
  - Number of Shells: 2
  - function: *Main('bunny.stl','gyroid',36,2,0.1,render_res=10)*

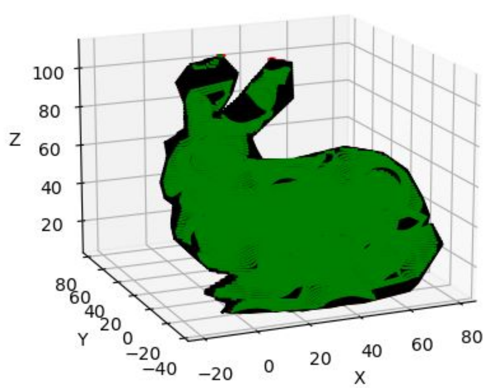


Figure 5: 'bunny.stl' Output (Render Resolution = 5)

- Input *STL* files and output *gcode* files are shared with the python script.
- G-Code visualization of *bunny.stl* with layer thickness is 0.5 mm with *TPMS* structure of diamond is given in the Figure 10. and Figure 11. I obtained the G-Code visualisation with 0.5 mm layer thickness since with 0.1 mm layer thickness inside of the bunny was not visible.

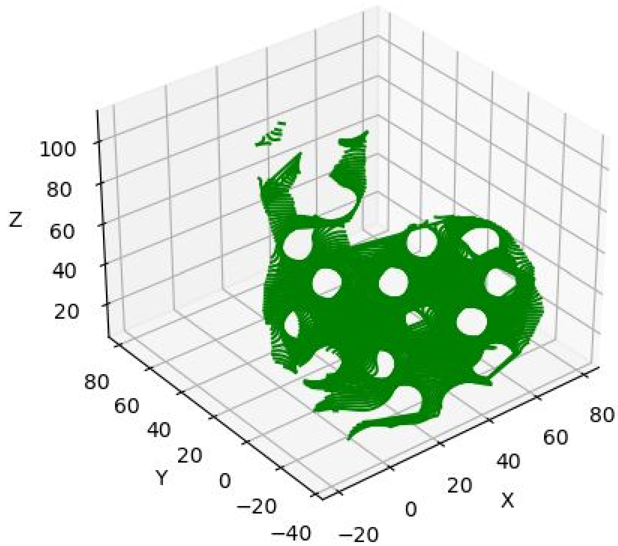


Figure 6: 'bunny.stl' only *TPMS* Structure (Render Resolution = 5)

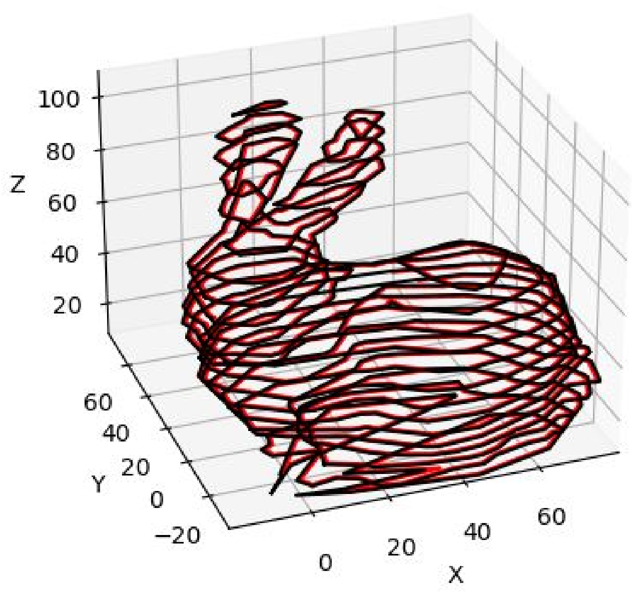


Figure 7: 'bunny.stl' Outer Boundary and Shells (Render Resolution = 50)

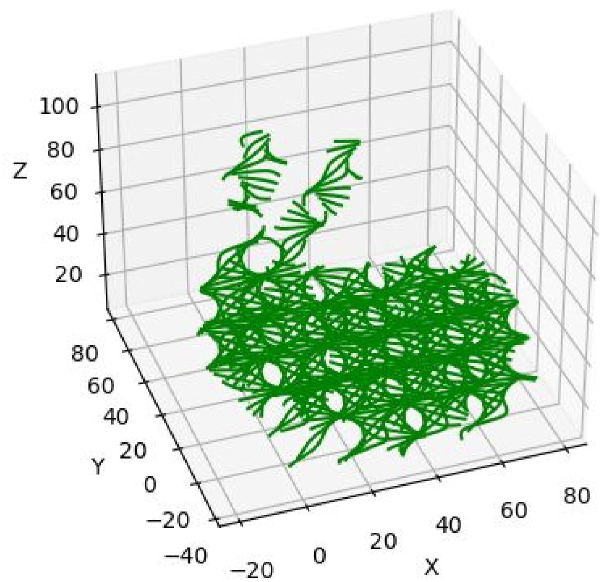


Figure 8: 'bunny.stl' with *TPMS* structure: *diamond* (Render Resolution = 50)

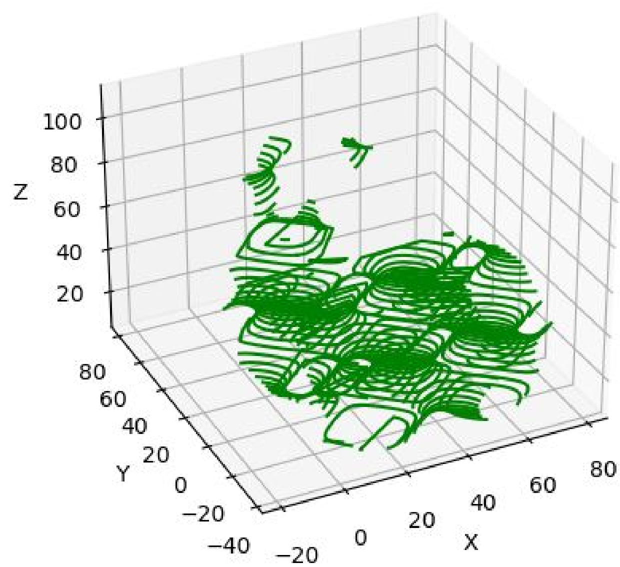


Figure 9: 'bunny.stl' with *TPMS* structure: *schwarz* (Render Resolution = 50)

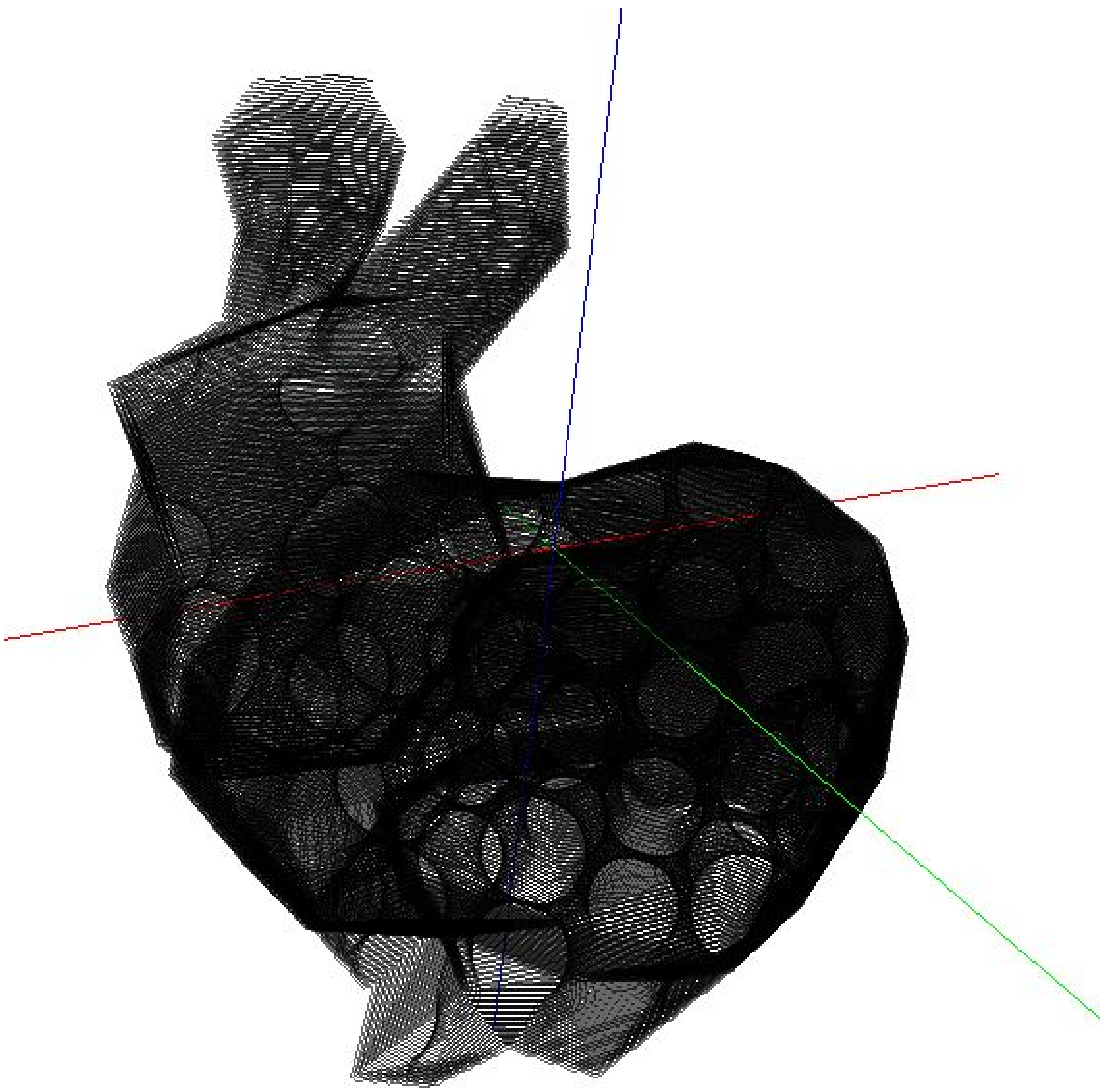

Figure 10: G-Code visualization of *bunny.stl*

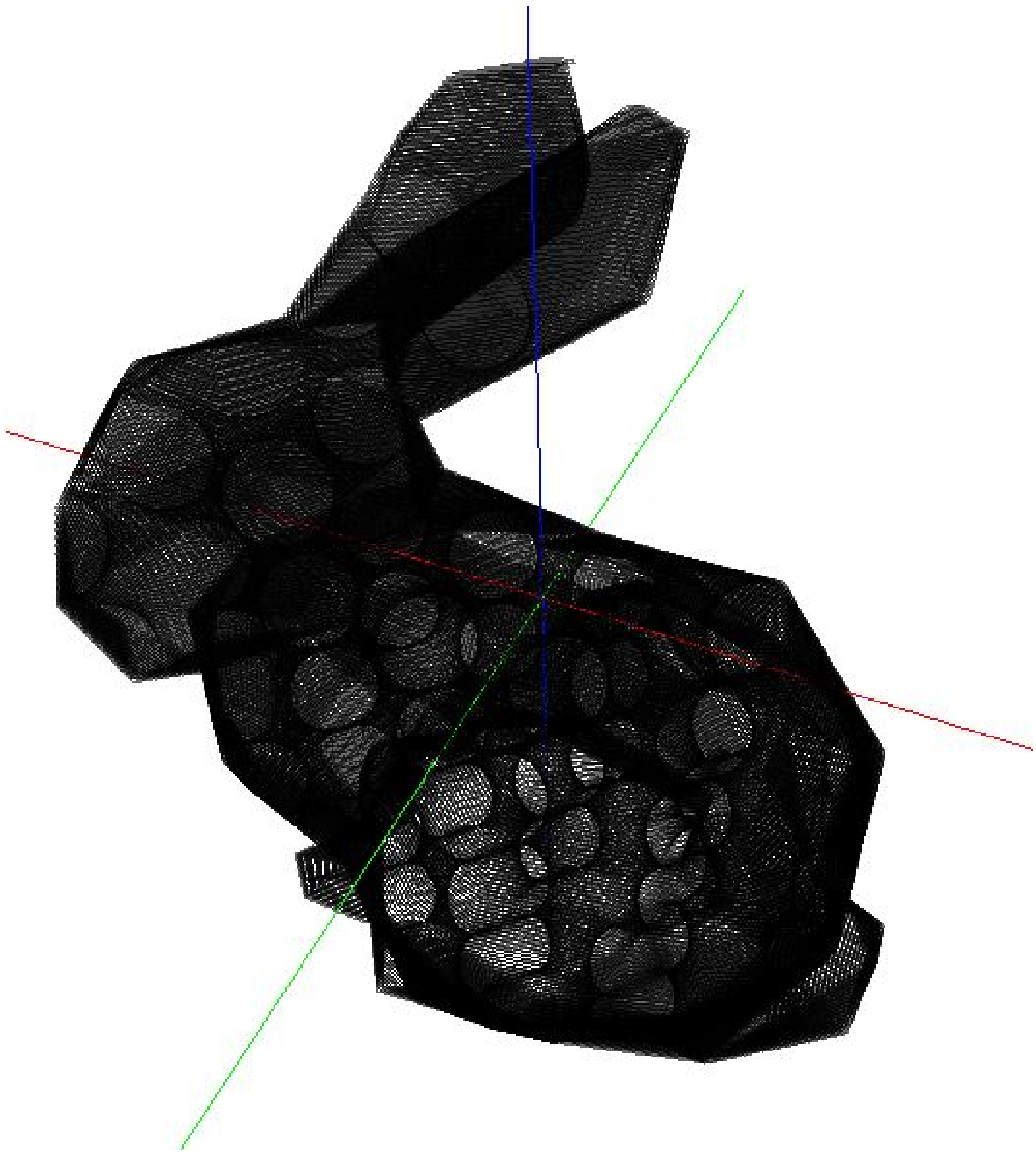

Figure 11: G-Code visualization of *bunny.stl* 2