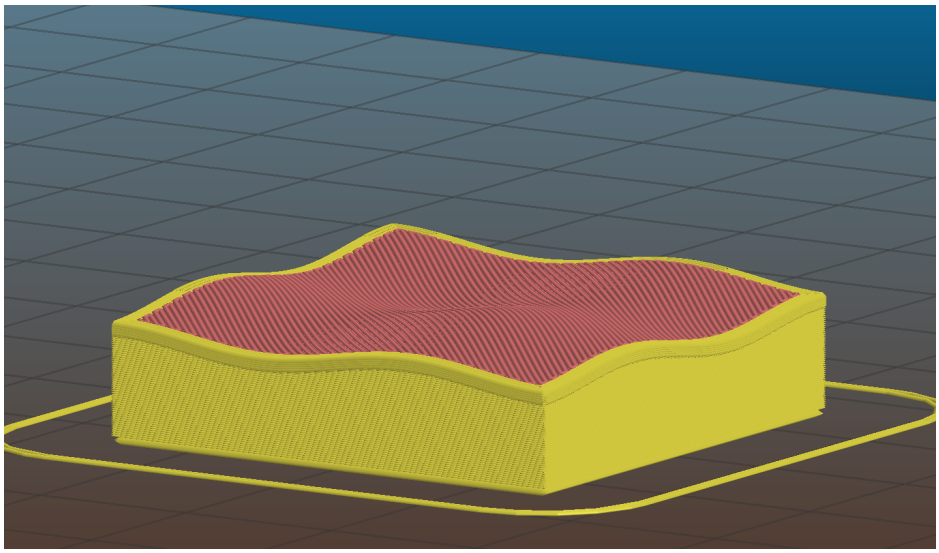
Non-planar slicing and Normal-to-Surface path programming



Louis ROULLIER - louis.roullier@ensta-bretagne.org

Composite Material, Manufacture and Structures Laboratory - Colorado State University
tutor : Donald W. Radford - dradford@rams.colostate.edu

ENSTA BRETAGNE

Lab-STICC

COLORADO STATE UNIVERSITY

Acknowledgements

Before starting this report, I would like to thank Dr Donald W Radford who is the head teacher of the Composite Materials, Manufacture and Structures Laboratory (CMMS) from Colorado State University (CSU). He enabled me to discover a new way of working and to apply my knowledge in robotics engineering to a field which really interests me. He also helped and supported me in my different approaches during the project.
I also would like to thank Dr Christian Jochum from ENSTA Bretagne. Indeed, he really trusted and encouraged me during all this internship. He made sure that the ENSTA Bretagne students' internship was as pleasant as possible : He helped us doing the administrative approaches and met the administration of the university to improve our experience.

Résumé

Durant ce stage de 4 mois réalisé dans le laboratoire de matériaux composites (CMMS) de l'université d'état du Colorado (CSU), j'ai du mener à bien un double objectif. Le but était de développer une méthode de tranchage non-planaire pour impression 3D tout en faisant en sorte que la buse reste normale à la surface d'impression. Pour ce faire, j'utilise un logiciel permettant de générer des GCODES contenant des couches planaires et non-planaires. Grâce aux commentaires présents dans ce fichier, je peux calculer les vecteurs normaux relatifs à chaque point en faisant un maillage de chaque couche. Par la suite, je transmets ces données à un logiciel nommé RoboDK par le biais d'un fichier de type .CSV. Ce dernier logiciel génère alors un code en RAPID interprétable par le robot et nous pouvons alors réaliser des tests d'impression.

Abstract

During this 4-month internship in the Composite Materials Laboratory (CMMS) at Colorado State University (CSU), I had a dual objective. The aim was to develop a non-planar slicing method for 3D printing, while ensuring that the nozzle remains normal to the printing surface. To achieve this, I used software to generate GCODES containing planar and non-planar layers. Thanks to the comments in this file, I can calculate the normal vectors for each point by meshing each layer. Then, I transmit this data to a software program called RoboDK via a .CSV file. This software then generates a RAPID code that can be interpreted by the robot, enabling us to carry out print tests.

Contents

1 Contextualization

The current state-of-the-art in structural composites processing for wind blade manufacture makes use of molds that define the shape of the finished composite component. Creating the heated molds onto which the composite is formed is costly, adding significantly to the cost of the final product and reducing the ability to make geometry-based design modifications during the product life cycle. When using conventional material laydown techniques, design relies on global ply orientations and current processes neither allow fibers to follow complex load paths in-plane or out-of-plane, limiting the mechanical performance. Additionally, structural cores, added between layers of reinforcement to boost out-of-plane stiffness, can be costly to procure and difficult to position and retain during shear web processing. Alternatively, additive manufacturing (AM) approaches can form the basis for processes which accurately produce, position and retain complex structural cores, resulting in reduced excess material, and substantial cost savings. Further, through incorporation of continuous reinforcement fiber, AM can offer a technical opportunity to develop composite structures which overcome limitations in fiber positioning, can enable innovative composite designs that cannot be commercially manufactured by other means, can reduce the process embodied energy, and can be manufactured with a significant reduction in the amount of complex tooling, resulting reduced cost, reduced weight, and reduced Levelized cost of energy (LCOE).

The main objective of this research project is additively manufacture the internal structure of a wind turbine blade, which will be integrated with conventionally produced composite aeroshells. This internal structure makes up almost 3,000 kg of the mass of a 60 m blade and uses molds valued in the millions of dollars. AM approaches incorporating both discontinuous and continuous fiber reinforcement will be applied, but the focus will be on concepts that enable design innovation and result in cost, weight, and embodied energy reduction of the resulting composite. The additive processes employed will include real-time consolidation and rigidization to overcome the need for heated molds or ovens, reducing the energy requirements, and enabling positioning of continuous fiber reinforcement out-of-plane without support tooling, overcoming current design limitations. The proposed activities build on two key areas of technology demonstrated at the laboratory-scale at Colorado State University (CSU): (i) out-of-build plane continuous fiber thermoplastic composite additive manufacture with radically reduced tooling; (ii) LASER-assisted local thermal processing of thermosets tailored for extrusion, enabling unsupported out-of-build plane processing.

2 Introduction

In the context of additive manufacturing of thermoplastics to design wind turbines, people need to rethink the concept of 3D printing. This results in changes in the printing material, in the machines used or in the path planning. Nowadays, classic 3D printers use a layer-by-layer approach. These layers are all planar. That is why people speak more about 2.5D printing. Within this project, I used a 6-axis robot in order to print non-planar layers and to implement normal-to-surface path. The material I used to print the most was glass fiber.
The classic process which print planar layers provides undesirable consequences such as staircase effect. This results in a reduction in the structure coherence and mechanical problems. This staircase effect also provides bad consequences about aesthetics : People can find rough structures due to striations. Other properties of an object can be affected because of staircase effect : friction, fluid-dynamics, and aerodynamics can be different. A concrete picture about staircase effect is given on figure 1.

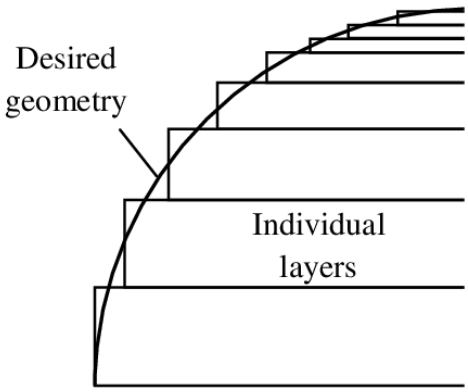


Figure 1: Concrete example of the consequences of staircase effect

By the way, the stair-stepping is way worse on surfaces with a low ramp angle than on those with a high ramp angle. A graph showing this phenomenon is available on [Alh18].
Furthermore, I tried to stay normal to the printing surface with the robot. This will enable people to have a better layer adhesion, to avoid collisions between the nozzle and the piece or to have lower roughness on the surface (and so better aesthetics properties). To adapt to the laboratory tools, I used Python, RoboDK and Slic3r to develop the process. These tools were very different from those I used during my second year at ENSTA Bretagne (I used CoppeliaSim for simulation, Cura for planar slicing...)
This work presents a slicer that is capable of generating a .CSV object which contains the position and the normal vector for every point. I will also explain how people can use this file to print a piece with RoboDK.

3 Generation of non-planar layers

To print a 3D object, people generally follow a three-steps approach. First, they design their piece with CAD softwares such as Catia or Autodesk. Then, they transfer the piece to an other software which is called a slicer. For example, at shcool, we are used to use Cura which is a planar slicer. This step enables to convert a 3D piece into a GCODE. This kind of file is readable by a 3D printer and contains the different points the nozzle should reach. Normally, software as Cura just slice the piece into very thin layers according to the user parameters (layer thickness...). In our context, to generate the non-planar layers, I use an open-source software called Slic3r. The details about how to use it was found with [Sli]. First, I will explain the theoretical approach to generate non-planar layers. I will particularly explain how to slice in a multidirectionnal way and how to generate a toolpath for a complex piece. Then, I will give a brief tutorial which shows how to use the software and why it will be useful for the rest of the process.

3.1 Non-planar slicing

The multi-direction toolpath is generated by slicing the model in different directions in order not some layers to be horizontal and so planar. This process of multidirectionnal slicing uses two main modules.
The first module takes the object and decomposes it by searching for closed concave loops. Therefore, it slices the original object into sub-volumes. In other words, this module is designed to do a mesh of the object. The second module is designed to calculate the best printing directions for each sub-volumes. the software uses regular planar slicing algorithm to slice them. Nevertheless, due to the sub-volumes sizes, people can still speak about non-planar slicing for the entire object.

3.2 Fully three-dimensional toolpath generation

Researchers Micali and Dornfeld (2016) devised a method to eliminate stair-stepping artifacts resulting from the layer-based structure in 3D printing. This method was found in [Alh18]. Their approach involves generating a 3D toolpath capable of following a complex, free-form surface. This approach was primarily designed for three-axis machines. It takes into consideration the nozzle's geometry to avoid collisions between the nozzle and the printed piece.
First, people need to generate the inverse toolpath offset to calculate the printability of a path.The nozzle is conceptually flipped upside down, with its tip tracing the printing surface to define an envelope shape using its body. This envelope surface serves as a printable and collision-free area. Points above the printing surface on the envelope are defined as unreachable by the extrusion head, rendering the current shape unprintable. A tolerance-based comparison between the envelope and the object surface determines whether the envelope can replace the object surface for toolpath generation. The toolpath is constructed by starting the process from one side of the surface and progressively filling along an edge until the opposite side is reached. This filling process keeps going with new starting points in unfilled areas until the entire surface is covered. By applying this methodology to a single shell, a complete 3D toolpath without any collision is generated.

3.3 How to mix planar and non-planar layers

This process uses the method of Huang and Singameneni (also found thank to [Alh18]. This method is based on a classification of the facets of an STL file. An STL (stereolithography) file is a common file format used in 3D printing and computer-aided design (CAD) to represent the geometry of a three-dimensional object. It defines the surface geometry of the object as a collection of interconnected triangles, creating a mesh representation. Each triangle is defined by its vertices (points in 3D space) and their corresponding normals (vectors indicating the direction of the triangle's surface).
The classification of each facet is based on the angle between the normal vector of the triangle and the z-axis. After that, each piece of the mesh belong to top, bottom or side surface. Next, all connected surfaces are defined as one single continuous top surface. The top surface is offset downwards along their facet normals to get the number of desired shell surfaces. To get the planar layers, the offset surfaces are subtracted from the original STL and a new one is created. The planar layers are then generated from the new STL.

3.4 Comparison of a piece with different types of slicing

In order to understand the real difference between a planar slicer and a non-planar slicer as Slic3r, I decided to visualize a piece using both types of slicer. In figure 2, the piece was sliced with the planar version of Slic3r. The way it works is comparable to classic slicers as Cura. In figure 3, the piece was sliced with the Slic3r non-planar version.

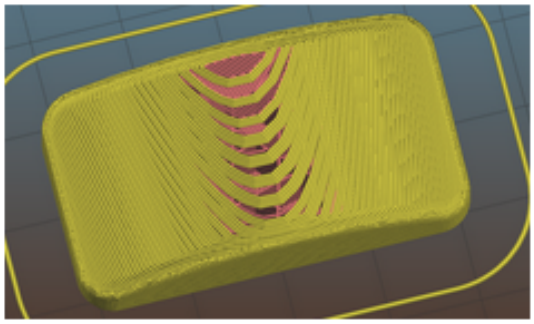


Figure 2: Slicing of the piece with a planar process

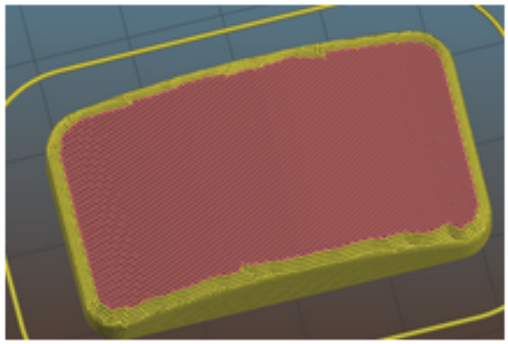


Figure 3: Slicing of the piece with a non-planar process

These following pictures show that the non-planar Slic3r functionality provide more coherent pieces. Indeed, the mechanical structure and aesthetics is clearly better in the second case.

After having visualized the piece in both cases, I can generate the Gcode. To do that, I have to put the "verbose" option in the software functionalities. It enables to have comments on the

GCODE which are very helpful for the following of the process. The beginning of a GCODE produced by Slic3r is given on the next page :

```
M109 S285 ; set temperature and wait for it to be reached
G21 ; set units to millimeters
G90 ; use absolute coordinates
M82 ; use absolute distances for extrusion
G92 E0 ; reset extrusion distance
G1 Z0.300 F7800.000 ; move to next layer (0)
G1 E-2.00000 F2400.00000 ; retract extruder 0
G92 E0 ; reset extrusion distance
G1 X234.607 Y225.055 Z0.300 F7800.000 ; move to first skirt point
G1 E2.00000 F2400.00000 ; unretract extruder 0
G1 F1800
G1 X235.118 Y224.545 Z0.300 E2.01642 ; skirt
G1 X235.750 Y223.989 Z0.300 E2.03555 ; skirt
G1 X236.571 Y223.406 Z0.300 E2.05845 ; skirt
G1 X237.515 Y222.859 Z0.300 E2.08324 ; skirt
G1 X238.652 Y222.350 Z0.300 E2.11156 ; skirt
G1 X239.264 Y222.154 Z0.300 E2.12617 ; skirt
```

Now, I have the different points of my different layers. The next step of the approach is to calculate the normal vectors to each point.

4 Normal-to-surface path programming

The very first step consists in using the previous process with Slic3r. Indeed, I used the different comments in the gcode to have a list of layers. If my program recognizes the words "END" and "layers" in the same comment, I add a layer to the list. A layer is characterized by a list of points belonging to it. For the process of finding the normal vectors to each points of the layers, I use two methods : A first one based on interpolation and a second one which creates a mesh for the layer. I will explain the both methods and compare them. For this example, I choose the piece available on figure 3 because it has planar and non-planar layers.

4.1 Interpolation of the layers

The first step of the interpolation is creating x and y profile in the non-planar layers. In other words, I cut my layer into slices of 0.3 mm according to x and y axis. The the value of 0.3 mm was decided by an ex-master student from my lab called Isaac Morris. Because it is easier at the beginning, I considered that my profiles are polynomial. To interpolate the curve, I used the least-square method which consists in minimizing the sum of the squares of the deviations. To reach this goal, I used Python classic library as Scipy which contains a module which implements this method. The code I implemented is available here :

```
from scipy.linalg import lstsq
def least_squares_surface(x, y, z, degree):
    x = np.array(x)
    y = np.array(y)
    z = np.array(z)
    A = np.column_stack([x ** i * y ** (degree - i) for i in range(
       degree + 1)])

    coefficients, residuals, _, _ = lstsq(A, z)

    def surface_function(x, y):
        return sum(coefficients[i] * x ** i * y ** (degree - i) for i
           in range(degree + 1))

    return surface_function, residuals,coefficients
```

This simple function consists in creating first a conception matrix with polynomial terms and then doing the linear regression. Nevertheless, this function takes into argument the degree of the expression. To determine it, I used the code on the next page :

```
def choose_degre(x,y,z):
    residuals_list=[]
    grid_x, grid_y = np.meshgrid(np.linspace(min(x), max(x), len(x)),
       np.linspace(min(y), max(y), len(y)))
    for i in range(15):
        surface,residuals,coefficients = least_squares_surface(x,y,z,
           degree=i)
        surface_z = surface(grid_x, grid_y)
        residus.append(residuals)
    ideal_deg=residuals_list.index(min(residuals_list))
    final_surface,residuals_finals,coefficients =
       least_squares_surface(x, y, z, degree=ideal_deg)
    final_z=final_surface(grid_x, grid_y)
    return ideal_deg,final_z
```

In this function, I first create grids of points in XY plan by using Meshgrid function from numpy module. Then, I calculate the least-square method residuals for a number of degrees that I choose (I take 14 degrees in this example). I finally find what degrees has the best residuals values (I want the less huge residuals) and I use it to calculate the best polynom associated to my surface. Nevertheless, when I printed my profiles thanks to Matplotlib, I discovered that there were some "problematic points" which perturbate the interpolation. An picture of these points are available on figure 4 (they are surrounded by a black circle):

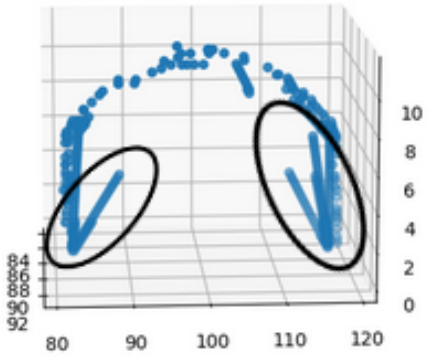


Figure 4: An x-profile and its problematic points

To fix this problem, I chose to delete the problematic points of the profile to enhance my model. To do that, I used the RANSAC algorithm which is a robust model estimation technique that is mainly used to estimate a mathematical model from a data set that contains outliers (values that

fit the model) and outliers (values that don't fit the model). My version of RANSAC algorithms is given here :

```
def ransac(points, iterations, threshold,ideal_deg):
    points=points.tolist()
    best_model = None
    best_inliers = []

    for i in range(iterations):
        sample = random.sample(points, ideal_deg+1) # S lection de
           deg_ideal+1 points pour un polyn me de degr  deg_ideal
        ys = [point[1] for point in sample]
        zs = [point[2] for point in sample]

        model = np.polyfit(ys, zs, ideal_deg)

        inliers = []
        for point in points:
            if abs(point[2] - np.polyval(model, point[1])) < threshold
                and point[2]<max(z_first_x):
                inliers.append(point)

        if len(inliers) > len(best_inliers):
            best_model = model
            best_inliers = inliers

    return best_model, best_inliers
```

First, I select random points (ideal degree+1 points for a polynom with an ideal degree). Then, I find the inliers points with a distance below a certain threshold. Then, I update the best model and the best inliers if the number of them is better than in the previous best model. People have to adapt this algorithm in function of the nature of the profile : Here, I give an example for an x-profile. To use the same program for a y-profile, people have to replace line 32 by the instruction "xs = [point[0] for point in sample]" and line 35 by the instruction "model=np.polyfit(xs,zs,ideal)" The result of the interpolation for an x-profile is given on figure 5 (on the next page) :

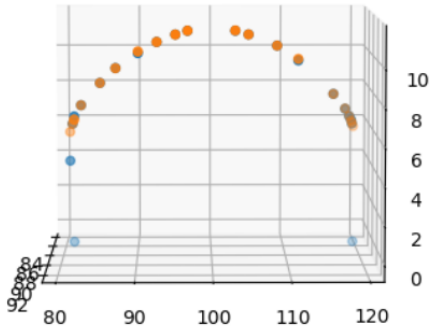


Figure 5: An x-profile and its interpolation

In this figure, people can see two colors : The blue curve is the real one and the orange is the interpolated one. What is interesting is the little difference between these two curves. Nevertheless, I am aware that the use of RANSAC algorithm here is a little bit approximative : I don't know the importance of the deleted points in the mechanical structure of the piece. I considered that they were part of a transition between different profiles which could not be always true.

Nevertheless, I thought that the better way to verify the reliability of the method was to print the piece and analyze the mechanical structure. Therefore, I kept going the approach by calculating the normal vectors for the non-planar layers.

4.2 Calculation of the normal vectors for the entire layers

During this approach, I tried to calculate the normal vector for each point by calculating a gradient : Each point belong to an x-profile and a y-profile. I calculate the gradient for both profile by calculating the partial derivatives according to x and y. Then, I just evaluate the partial derivatives in the point I want to calculate the normal vector. Thanks to that, I can establish a gradient matrix (with a third row composed of 1) which I normalize after that. Then, I calculate the sum of the x-profile gradient and the y-profile gradient. The result of this process is shown on figure 6:

Thanks to this figure, people can see that there is absolutely no convergence in the vector field. Indeed, that comes from the calculation of the normal vectors for each point (the sum of the two gradients don't really give a normal vector to the point) and from the deletion of some points in RANSAC algorithms. That is why I had to develop an other method to calculate the normal vectors for a non-planar layer.

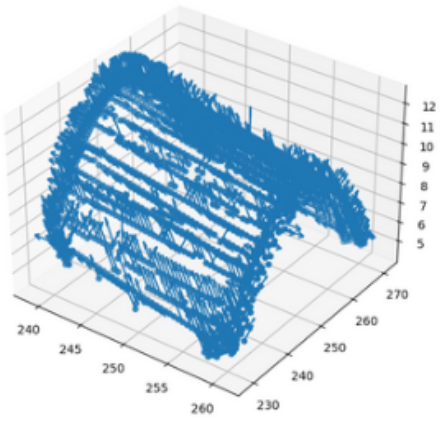


Figure 6: Normal vectors of every points in the layer

4.3 generation of a mesh for each layer

After having generated a list of layers, I decided to mesh the layers. It enabled me to have a plan that closed every point of them. I chose a Delaunay triangulation for my mesh. Indeed, the triangles are quite small, so I can have a better precision. In addition to that, this is quite a fast and easy process in Python : I just have to import the Delaunay section from the Scipy.spatial library. The lines of Python codes which enable to have the mesh (a list of triangles) are given there :

```
from scipy.spatial import Delaunay

def Delaunay(x,y,z)
    surface_2d_points = np.column_stack((x, y))
    tri = Delaunay(surface_2d_points)
    surface_3d_points = np.column_stack((x, y, z))
    surface_3d_triangles=surface_3d_points[tri.simplices]
    return surface_3d_triangles
```

Here, x,y and z are the lists containing the coordinates of each point in the layer. You can have them by searching for the elements in the layers list.
Nonetheless, it is important to explain the theoretical part about Delaunay triangulation. More details are given on [CD] The first step is to create a Voronoï diagram. It consists in subdivising the layer into n cells. This subdivision is based on this assertion comparaison: A point q belongs to the p_i cell if d(q,p_i) is inferior to d(q,p_j). Here, d(p,q) symbolizes the euclidian distance between p and q. An example of Vornoi diagram can be shown on figure 7.

The next step is to create the triangulation : We just have to connect the points of all neighboring Voronoi cells. The final mesh can be seen on figure 8.

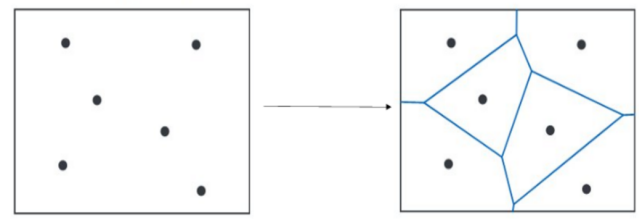


Figure 7: Voronoï diagram

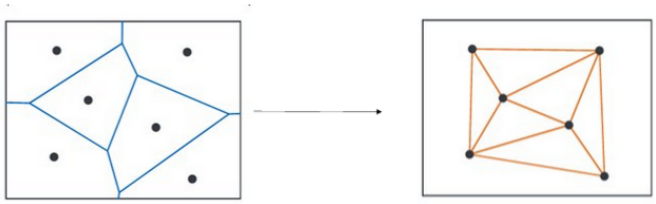


Figure 8: Delaunay mesh

4.4 Calculation of the normal vector of each triangle

Now that I have all of my triangles, I will try to calculate the normal vector to each of them. There is two ways to do it. The first one is to consider that every triangle belongs to a plan. The aim is to find a normal vector to this plan. We all know that a typical equation for a plan is $ax + by + cz + d = 0$. In addition to that, we know that a,b and c can be considered as the coordinates of a normal vector. Therefore, if I have three points (with their coordinates), I can just do an easy matrix calculation. An other method consists in calculating the cross product of the vectors generated by the triangle vertices. If S1,S2 and S3 are the three vertices of the triangle, then the vector $\vec{S1S2} \times \vec{S2S3}$ is normal to the triangle. In order to make the following part steps easier, I normalized the vectors.

4.5 Calculation of the normal vector of each point

I have the normal vector for each triangle thanks to the previous step. The next step is to do the same for the points. Thanks to my mesh, every point is a vertice of several triangles. Therefore, I can consider my normal-to-point vector as the mean of the normal vectors to the triangles in which the point is involved. Moreover, this this average is weighted by the distances between the points and the center of my triangles. Concretely, this approach is quite interesting

because it enables to have convergence with the normal vectors. The result of the method can be shown on figure 10. I also added the result with the interpolation method (figure 9) in order people to see the difference between the both process.

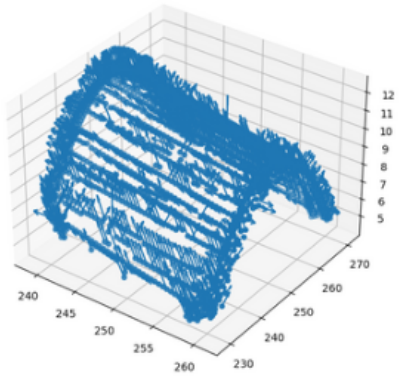


Figure 9: Interpolation method for the generation of normal vectors

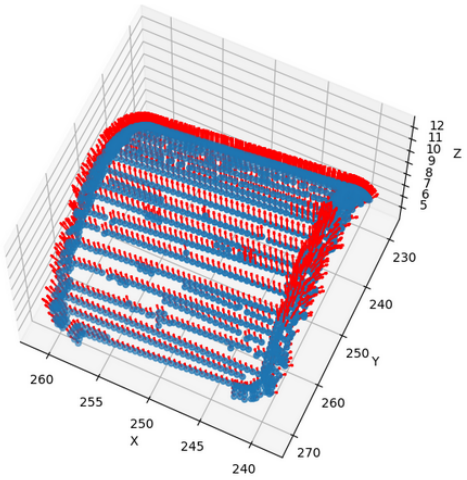


Figure 10: Delaunay triangulation method for the generation of normal vectors

From these two figures, people can see that there is a better convergence of the normal vectors with the method using the Delaunay triangulation. The convergence will enable to make the movements of the robot easier and to have a continuous path. With the following figures(figures 11 and 12), people can see the continuous path and orientation of the robot.

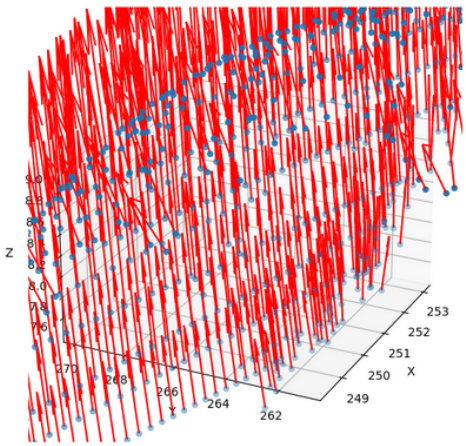


Figure 11: continuous path generated with Delaunay triangulation method - version 1

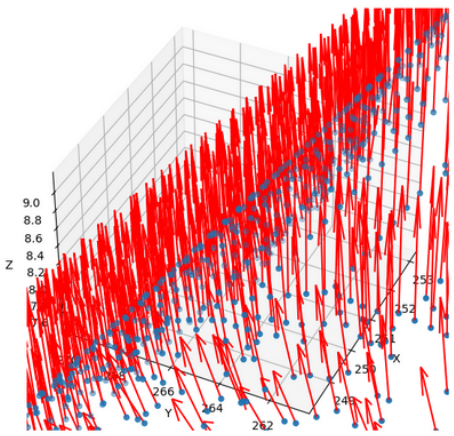


Figure 12: continuous path generated with Delaunay triangulation method - version 2

4.6 Verification of the method

The next step of the process is to verify the method for each point of the path. The approach consists in taking three points very closed together. I take the point from which I want to verify the normal vector and I verify its abscissa and its ordinate. Then, I take the point which have the closest abscissa and the closest ordinate. Thanks to these three points, I can generate two vectors (the origin is the point from which I want to verify the normal vector). I can calculate the dot product between the normal vector previously calculated and these two vectors. If the scalar product is less than 0.15, I consider that the normal vector is correct for the point. Otherwise, I search the points around my point I want to verify the normal vector (I use a Python code which enables to automate this search by calculating distance). Then, I calculate the mean of the normal vectors from these points. Nevertheless, I didn't have to correct the normal vectors for a huge amount of points. For a planar layer, I didn't have no mistake for my points. For a non-planar layer, I had a mistake for a very little amount of points (three percent of the layer). Indeed, for a non-planar layer, I had 64 problematic points for more than 1800 tested points.

5 Simulation of the process

The next step is to simulate the process. This is very important because it enables to be aware of the different behaviours of the robot. In order the simulation to be efficient, I have to recreate the real work environment. That is why I decided to use a RoboDK scene created by Isaac Morris. In this scene, people can find the two ABB robots, the turntable (which is equivalent to a printing bed) and other elements from the real environment. A picture of the RoboDK scene is available on figure 13.

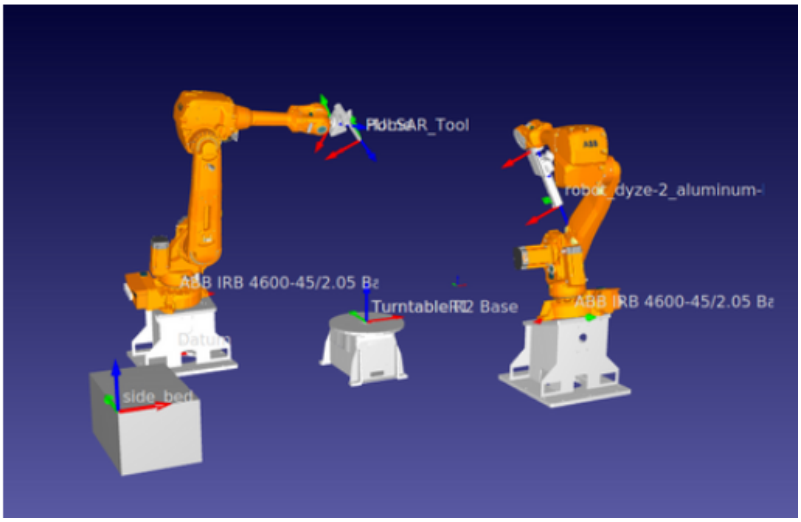


Figure 13: Scene for simulation

The main difference between the two robots is the extrusion head they use : the robot on the left is qualified as robot 1 and the one on the right is called robot 2. Thank to the previous steps, I have a list of points which corresponds to the path of the robot and a list of normal vectors (for every point of the path). Given that the simulation scene is on RoboDK, I decided to use the RoboDK library of Python to predict the different behaviors of the robot. The first step of the simulation process is to connect to the robot of the scene and to choose the frame I am interested in (here I take the frame of the turntable which is my printing bed) and the robot speed. Then, I establish my strategy :
- For the orientation of my tool, the main goal is to determine the good rotations : I have several normal vectors and I know that the z-axis coordinates of my turntable frame are (0,0,1). I can calculate the angle between the z-axis of my printing bed and my normal vector. To reach that goal, I can just calculate a scalar product between these normalized vectors which gives me the cosinus of the angle. The instruction I give to the robot is to do a rotation (with the angle I previously found) around x-axis. For my situation, it is better to consider a rotation around an only axis. I chose the x-axis from my different experiments. Moreover, I had to determine the sign of my angle in function of the place the tool was on the piece. I calculate the cross product between the z-axis of the printing bed frame and the normal-to-surface vector. Then, the sign of the angle will be the one of the cross product x-coordinate.
- For the position of the tool, I just take the coordinates of the points my path and I add an offset. This offset is just the position of the print bed center according to the scene main frame (which is called datum). The results of the simulation can be seen in the following figures. They show simulations for both robots in different cases : printing of a planar layer for robot 1 (figure 15), printing of a planar layer for robot 2 (figure 14), printing of a non-planar layer for robot 1 (figures 16 and 17), printing of a non-planar layer for robot 2 (figures 18 and 19).

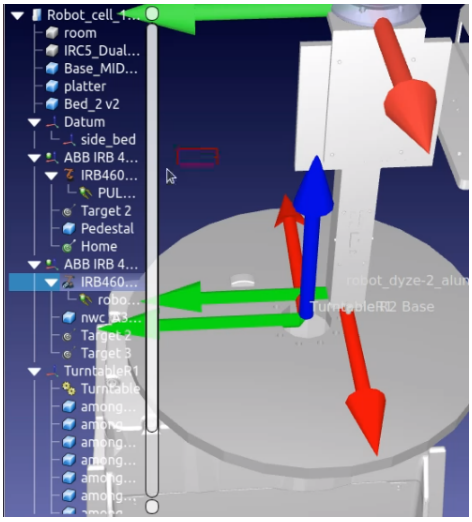


Figure 14: Printing of a planar layer with robot 2

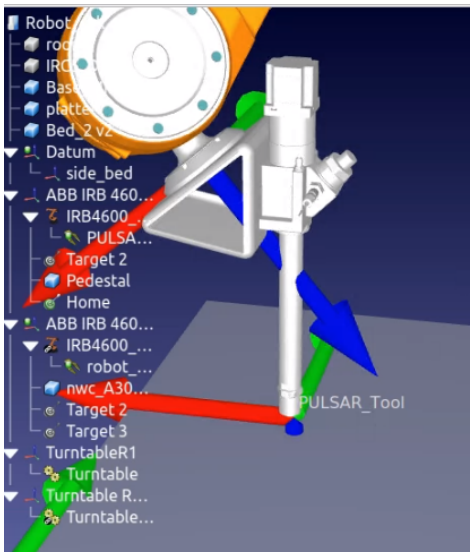


Figure 15: Printing of a planar layer with robot 1

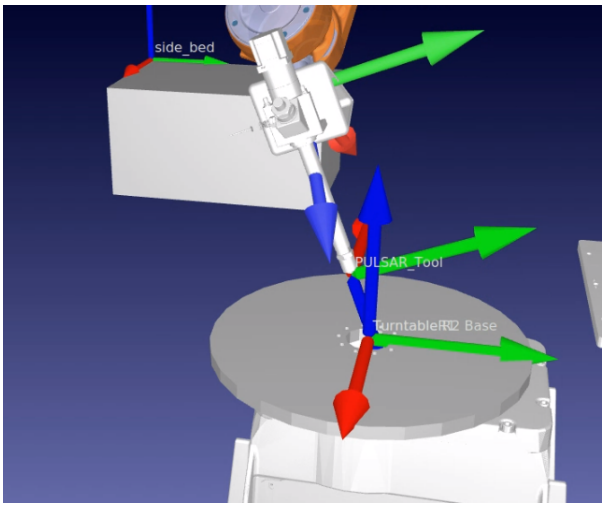


Figure 16: Printing of a non-planar layer with robot 1

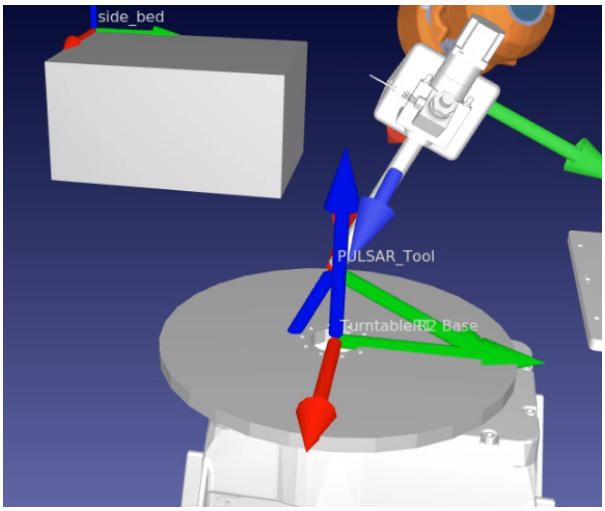


Figure 17: Printing of a planar layer with robot 1

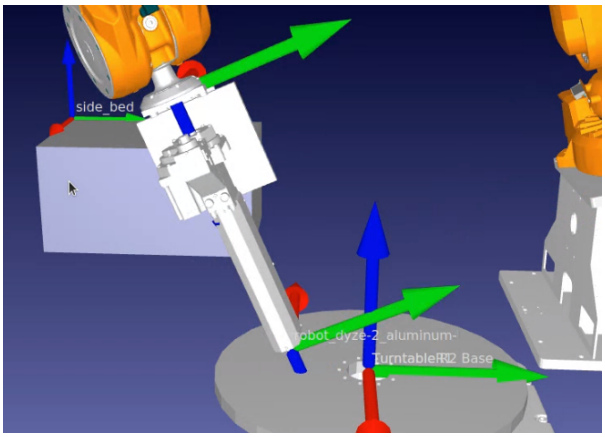


Figure 18: Printing of a non-planar layer with robot 2

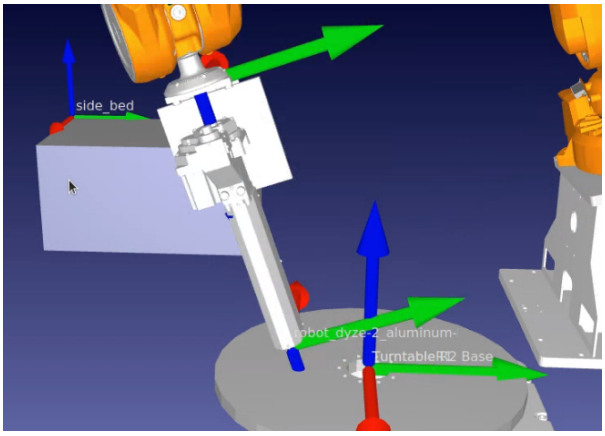


Figure 19: Printing of a planar layer with robot 2

On these figures, people can see that the planar and non-planar layers can be printed by the robots. At the beginning, the robots could print only one side of the piece. Nevertheless, the figure show that this problem was solved. The source of the mistake was that I did not change the sign of the angle in function of the tool position on the piece. Some of the videos of the different printings are available in the following link (I added an offset on z-axis in order to be clearer) :
- https://youtu.be/CWvhjipm5hA (printing of a non-planar layer with Robot 1)

However, some of the points of the path were not reachable by the robot. Indeed, their position and the orientation of their normal vector were represented as an unreachable target. To identify the positions of the unreachable targets, I plotted them in a graph which can be seen on figure 20.

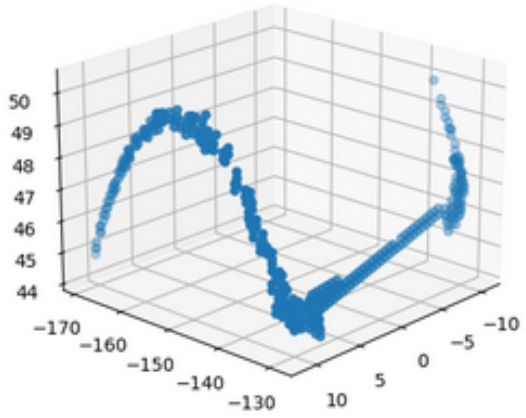


Figure 20: Distribution of the unreachable targets on the piece

On the figure above, people can see that the unreachable targets are on the contour of the piece. An unreachable target can come from two problems : the position of the point or the orientation of its normal vector. First of all, I tried to adjust the position of my piece on the print bed. Indeed, I put my piece in the very center of the turntable. It resulted in a reduction in the amount of unreachable targets (which can be seen on figure 21)

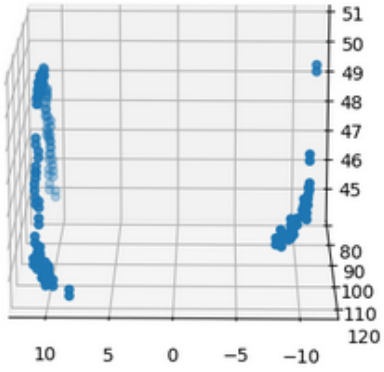


Figure 21: Distribution of the unreachable targets on the piece after centering the piece on the turntable

On this figure, people can see that the problems stay on the contour of the piece. People can guess that there could have been a bordure effect in the calculation of the normal vectors in the precedent part. To fix the problem, I decided to filter the normal vectors in the outline of the piece : I created a border zone in my piece (which encompasses all the normal vectors present between my outline and 2 mm before it). Then, I calculate the mean normal vector of that zone. Then, I replace every of the unreachable normal vectors by the mean normal vector of the zone. The difference before and after the filtering is shown on the following figures (figures 22 and 23):

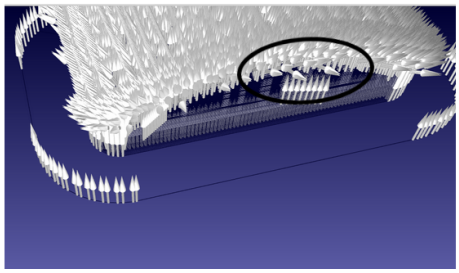


Figure 22: Distribution of the normal vectors on the piece before filtering

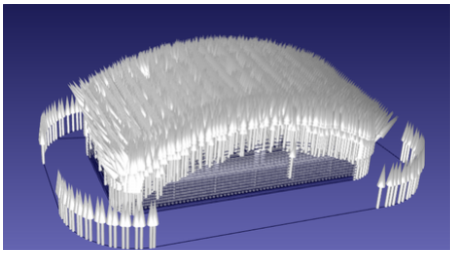


Figure 23: Distribution of the normal vectors on the piece after filtering

Here, people can see on figure 22 some surrounded normal vectors which represent unreachable targets. After filtering, I still have a convergence in my vector field and all the targets are reachable. That problem shows the importance of the simulation step because it would have been dangerous to directly launch the program on the robot : Indeed, I could have reached singularities that could have ultimately disrupted the robot's operation.

6 Implementation on the real robots

For the tests on the real robots, I established several strategies. The first one was to connect to the real robot and to apply my simulation codes directly. I tried to connect to the robot via RoboDK with the official documentation. Nevertheless, I did not succeed in connecting one of the robot because there was additional security on them. I also tried to write programs in the robot language (which is called RAPID and is specific to ABB robots). Nevertheless, I did not have much time to learn this language or to try to convert my Python codes in RAPID The strategy I finally found is to use the software RoboDK. After a weekly meeting, one of my colleague explained me how to use it : You just have to create a .CSV file which contains the position of every points and their normal vectors. After that, you can create a curve on RoboDK and putting it under the frame of the printing bed. The software will adapt to the coordinates of the points to the frame and will automatically convert that into a RAPID program. Therefore, I just had to create a .CSV file from my points and their normal vectors (which is quite easy with Python). Once I created the RAPID code thanks to the previous step, I can transfer the code to the robot with a software like Filezilla. Finally, I did some printing tests with this approach. Some videos of these tests are available here :
- https://youtube.com/shorts/9q1EIVa05Zs
- https://youtube.com/shorts/C1kUaQ2uu5g
The first link show a test without printing anything. The second test show a test which was designed to print an airtruss (within the project of a colleague). The final result of the print is shown on figure 24:

Figure 24: Printing test within the project of a colleague

In this figure, the part I printed is the white one. The blue support had already be done by a 3D classic planar printer. From my point of view, the piece is interesting because the angles are respected and the mechanical structure seems solid. Yet, it would have been interesting to do additional mechanical tests (tensile strength test, flexural strength test...) to verify the efficiency of the process.

7 Summarizing of the global approach and prevision of the next steps

During this four-month internship, I had to develop a method which combines a process of non-planar slicing ans an approach of normal-to-surface path programming. For the first step, I use the software Slic3r which enables to create Gcodes mixing planar and non-planar layers. Moreover, the comments in the file are very helpful for the following part : They enable to have the points in every layer of the piece. The second step consists in calculating the normal vectors for each point of the path. Once I have this data I can generate a .CSV file which is used by RoboDK to create a curve. This is converted into a RAPID code understood by the robot. Finally, the RAPID code can be transferred to the robot and the print test can start. In order to be more efficient and to make the process easier, I created a graphic interface for converting the non-planar Gcode into a .CSV file. This tool made with the Tkinter Python library can be seen on figure 25

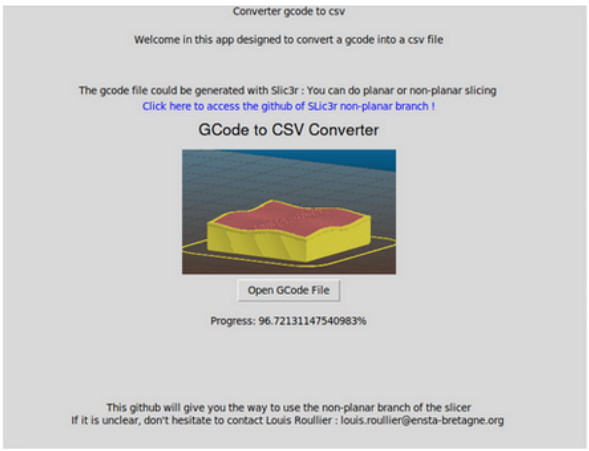


Figure 25: Graphical interface for converting the Gcode tothe .CSV file

on this interface, you can choose the Gcode you want by selecting it on your computer (I included a button for that). Then, I included a link about the use of Slic3r : It is a Github which gives details about the way to download the software and to use all the functionalities. The user can find the .CSV file in the "downloads" section of his computer. Moreover, I also implemented a virtual machine in collaboration with an ENSTA Bretagne student (Louis-Nam Gros). It enables the lab to launch our different applications without downloading Ubuntu or even Python.
To continue the approach, people could do more printing tests using the process described here. They can compare the structure of the piece if there with normal-to-surface path programming and without. They can also automate the full process with middlewares as ROS (first version). Indeed, the ABB robots have ROS packages which can be very useful. For example, the packages enable to connect directly to the robot without generating the .CSV file.

8 Conclusion

As a conclusion, this internship was the occasion for me to discover how to work in a lab. Indeed, I was confronted to a research topic and I had to develop my own approach to fulfil the objective : I had to establish a global strategy and to decide what weeks would be dedicated to what step. Thus, I developed organization and rigor skills. Moreover, my internship was in a lab and my topic had not been treated by my colleague before I came. Therefore, I didn't have much advice on the methods to use to reach my goals. That is why I made some mistakes like calculating the normal vectors of each layer through interpolations. Nevertheless, I learnt how to react and to change my methods after making mistakes. Finally, as I said previously, I had to adapt to my lab tools : All of my colleagues were used to these lab tools and I wanted them to use my process. Therefore, I could not develop my codes, simulation on software like CoppeliaSim (that I used at school). Thus, I also developed adaptation skills.
Furthermore, it was a very interesting human experience. Indeed, it was the first time I had to go working in a foreign country. Therefore, it was the occasion for me to discover a new way of working and to develop my skills in English. Finally, my internship tutor did not send us back the evaluation sheet. That is why I don't have it in the annexes.

List of Figures

Bibliography

[Alh18] Daniel Alhers. 3d printing of nonplanar layers for smooth surface generation. 2018.

[CD] Annabelle Collin and CÉCILE DOBRZYNSKI. Méthode de delaunay.

[Sli] Github of slic3r : https://github.com/zip-o-mat/slic3r/tree/nonplanar/.