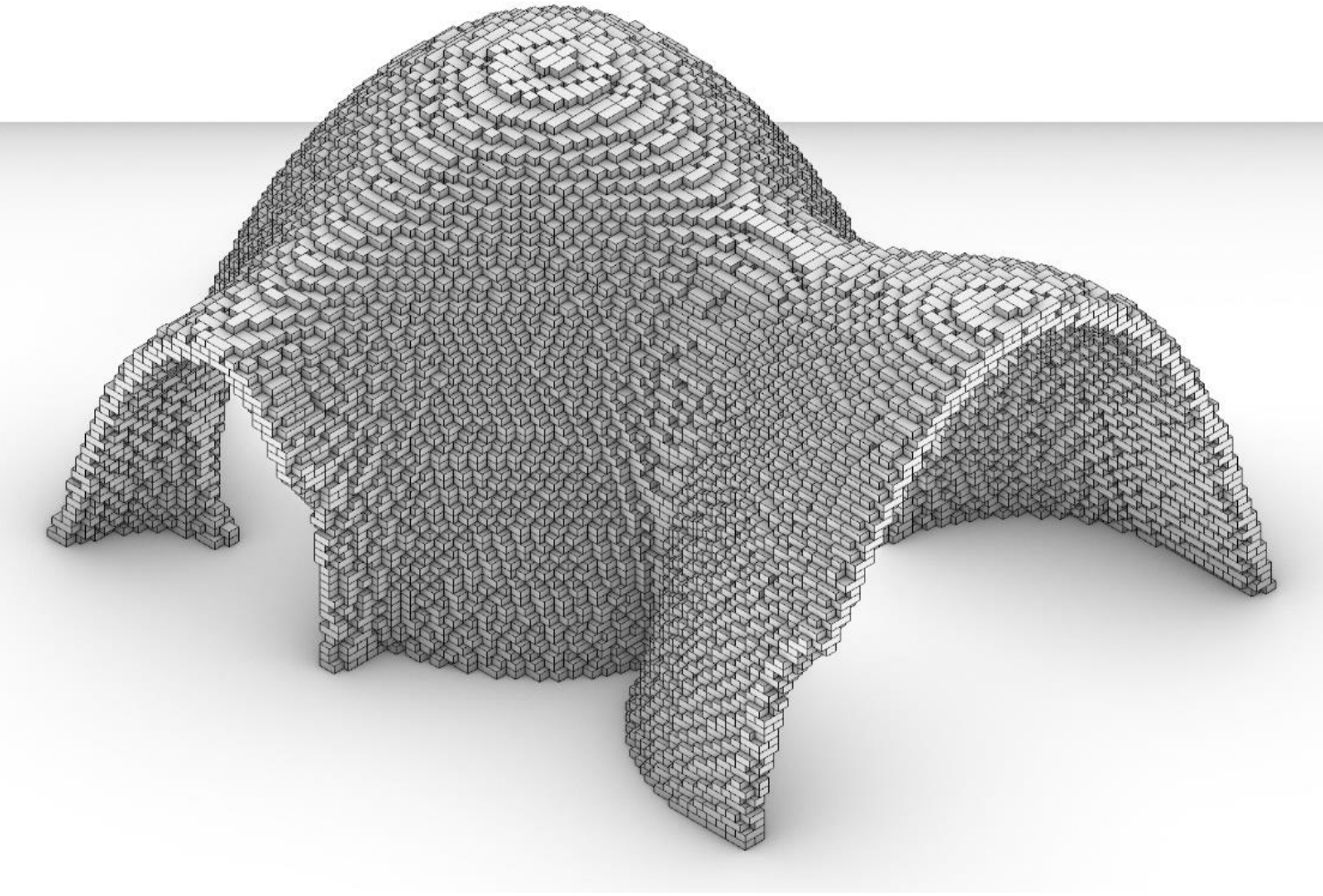
PYTHON ALGORITHMS

EARTHY (AR3B011) 2018-2019 Q1



Team 2.0
AMMAR TAHER
IVNEET SINGH BHATIA
JEROEN DE BRUIJN
PRETHVI RAJ
YAMUNA SAKTHIVEL

Contents

1 Introduction

The primary purpose of this document is to explain the two most prominent Python algorithms which were written. They were worked out in much more detail and are worth elaborating on. Both algorithms contribute to the building method, being a building compass and voxelization.

2 General information

The laptop used to run the algorithms is a HP Zbook Studio G5 Mobile Workstation with a 64-bit version of Windows 10 installed. It contains an Intel® i7-8750H "Coffee Lake-H" (hexa-core, 14 nm) Core™ CPU @ 2.2 GHz and an installed memory (RAM) of 16GB (1x16GB) DDR4 @ 2666 MHz. The algorithms were run in the Grasshopper version which is included in Rhino 6. To be specific, the following Rhino version was used: Version 6 SR8 (6.8.18240.20051, 28/08/2018).

2.1 Other scripts

Besides the two main algorithms, which will be explained in the next chapter, several other algorithms were written which helped to develop Python knowledge. Three examples are:

- Calculate the center of mass from multiple point masses, this determined the initial site location. It was was the centre of the population, so not the geographical centre. This was a small basic script which worked really well and was implemented.
- Rectangle packing, to fit the program within the site. The rectangle packing algorithm worked, however, to get the algorithm to generate a useful and logical plan a complex piece of code was required. Instead of trying to program all different factors which would be taken into account, it was much more efficient to manually design the plan while taking in account a computational approach.
- Convert a relation chart from an online Google Sheet to a bubble diagram, which visualised the program. This algorithm helped to get a better feeling for the program and the connections within it. One very interesting part of this algorithm is the part that imports data from the Google Sheet. Gathering data from an online source can be useful for many different projects, not merely architectural ones. Since it has the potential to be widely applicable a code snippet of the Python script is added here below.

```
1.  # Import csv reader to read out csv data
2.  from csv import reader
3.  # Import urllib2.urlopen to open an online source
4.  from urllib2 import urlopen
5.
6.  # Room numbers are in this column in the google sheet file
7.  col_room = 0
8.  # Room descriptions are in this column in the google sheet file
9.  col_description = 1
10. # Room areas are in this column in the google sheet file
11. col_area = 2
12.
13. # The url to the published google sheet as a csv file
14. google_url = "https://docs.google.com/spreadsheets/d/e/2PACX-
    1vTym6pgoQDIy8d6xNqF73WD_Bm_YBUh-
    LtWmMwFdpBIcfansyL1_TUoXGlF_qoro7Lg2GeYGwi_eR6x/pub?gid=130210499&single=true&outpu
    t=csv"
15.
16. # Try to open the csv file
17. response = urlopen(google_url)
18. # Read out the response
19. csv_object = reader(response)
20. # The csv reader does not support indexing, it doesn't return a list but an iterator
    over the rows
21. # Therefore convert the csv object to a list
22. csv_list = list(csv_object)
23.
24. # Loop trough each row of the csv object
25. for row in csv_list:
26.     # Get room data
27.     room_number = row[col_room]
28.     room_name = row[col_description]
29.     room_area = row[col_area]
```

3 Compass

A building compass will help bricklayers by pointing out where to place a brick and under which angle. The compass is based on an old Nubian technique and merely functions as a pointing device.

Figure 1: Old Nubian compass used in the building process, source: http://aionarchitecture.com

Figure 2: Final result, a dome built using a Nubian compass, source: http://aionarchitecture.com

An algorithm was written which generates bricks on a surface and calculates the compass data for each brick. On the figure below on the left, the generated bricks on an arbitrary surface can be seen, as well as the compass in magenta which highlights a specified brick. The bricks are stacked neatly and follow the curvature of the surface. At the input area, which can be seen below in the figure on the right, it is possible to: select which brick to highlight; define different brick sizes for different layers for structural purposes; create extra gaps between the bricks for light and ventilation on the specified layers; rotate all the bricks a given angle in degrees over their own z-axis on certain layers, this is for aesthetic purposes and self-shading of the facade.

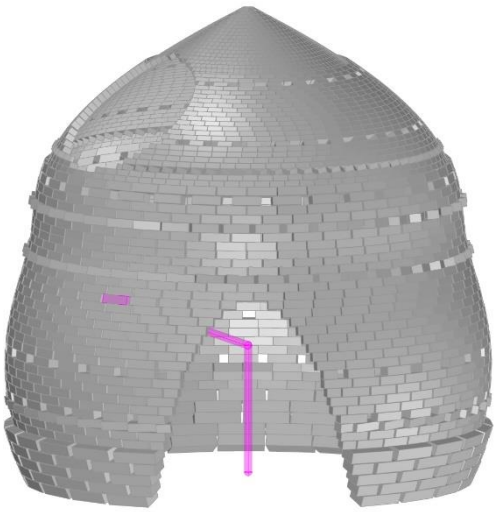


Figure 4: The generated bricks on a surface

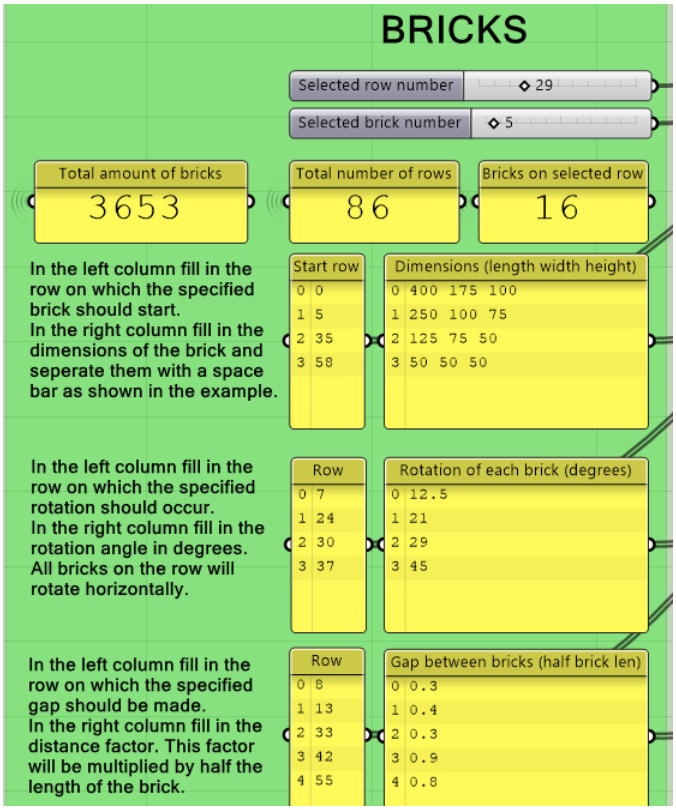


Figure 3: Input of Grasshopper script

As mentioned before, the algorithm calculates the compass data for each brick. It is numerical data, which contain the settings of the compass. Applying this data to the compass would allow the bricklayers to see where and under which angles each brick should be positioned. The data consists out of five variables, being: **C1** compass arm length; **C2** compass arm inclination; **C3** compass arm azimuth; **B1** local brick inclination, so the inclination of the brick relative to the inclination of the arm of the compass; **B2** local brick azimuth, so the azimuth of the brick relative to its own central z-axis; the size of the brick. The output and visualisation can be seen in the figures below.

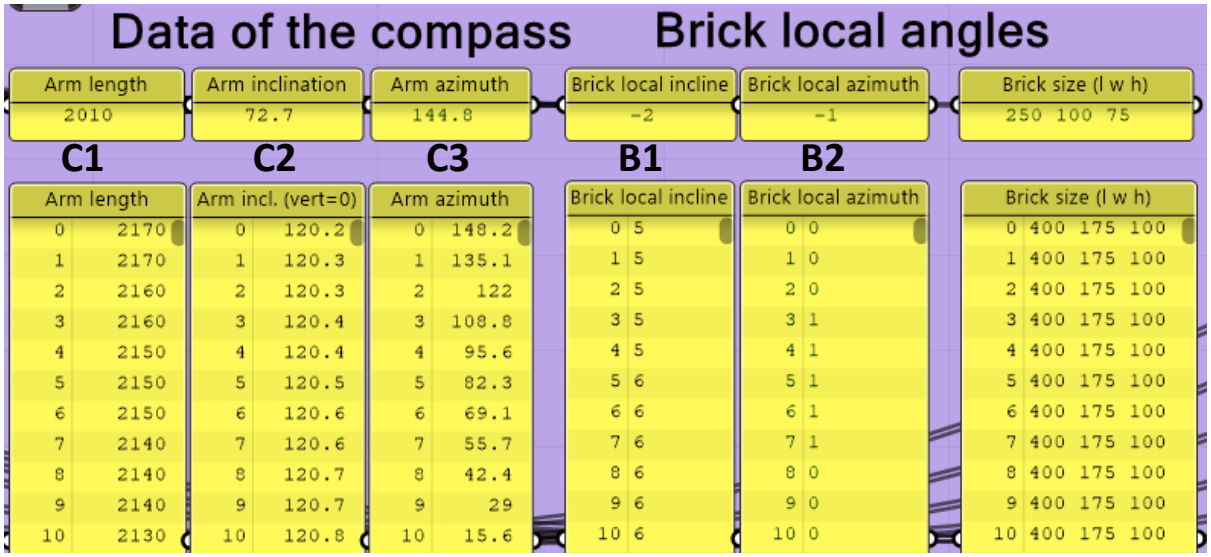


Figure 5: Numerical output of Grasshopper script

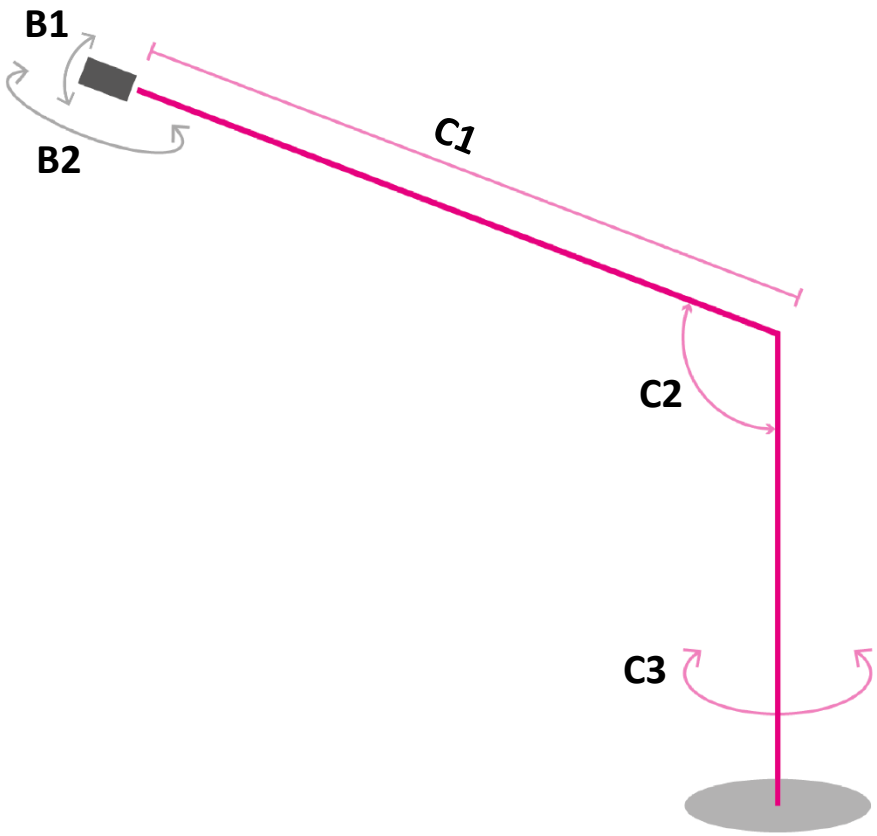


Figure 6: Visualisation of the compass data

3.1 Building method limitations

The output values are rounded off, because the masons are not able to set the values with the same accuracy as the computer is able to. The length of the arm for example is rounded to 10 mm and the angles of the compass arm to 0,1 degree. In order to visualise how the rounded data would influence the appearance another algorithm was written which places each brick based on the rounded values. The top figure on this page shows the bricks when they are placed using the numerical output as shown on the previous page. The bricks hardly move and the shape looks nearly exactly similar to the original generated bricks. The figure below shows how the bricks will be positioned when the angles of the compass arm are rounded to whole degrees. Gaps between some bricks appear and the rounding of data becomes visible.

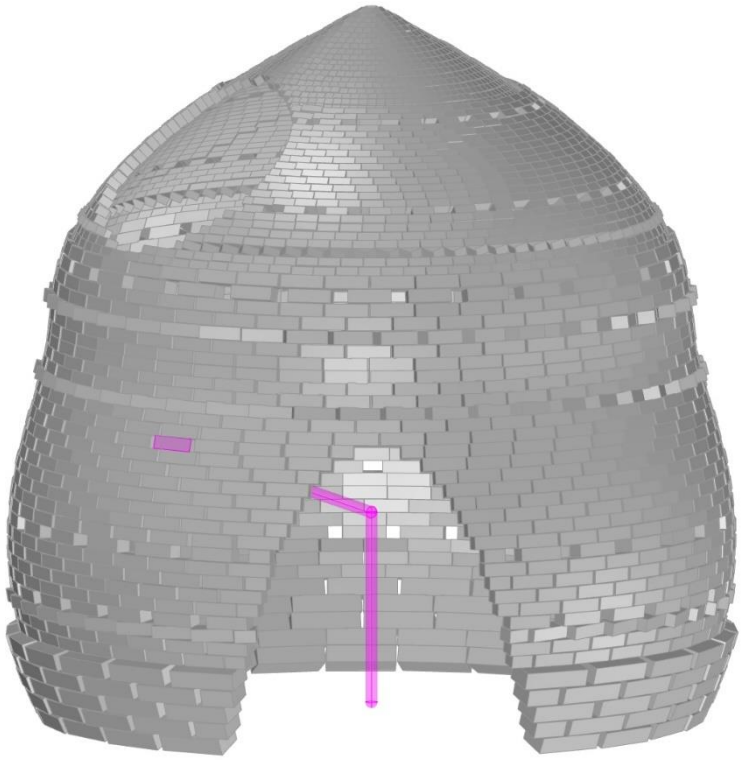


Figure 7: Bricks placed using angles which are rounded to 0,1 degree.

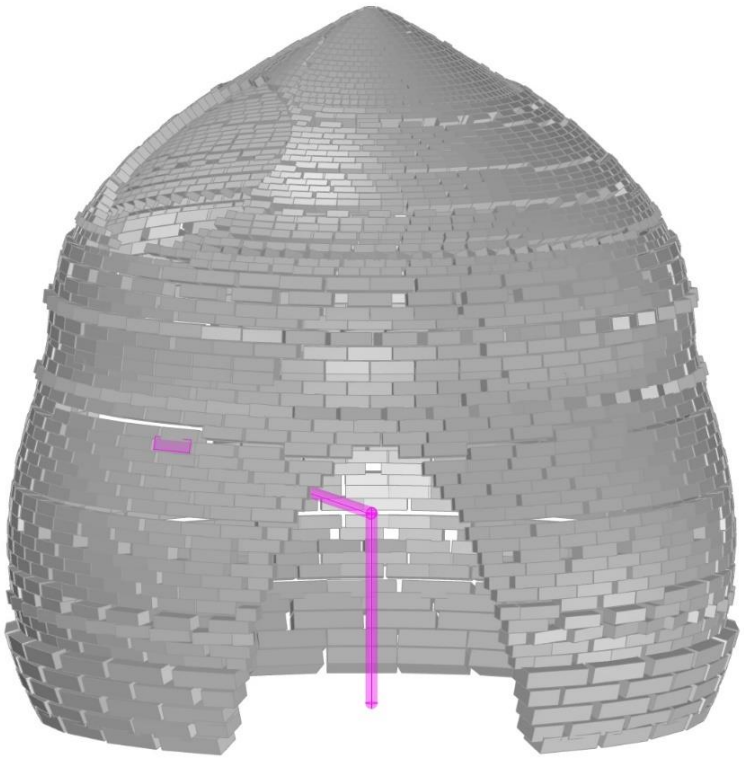


Figure 8: Bricks placed using angles which are rounded to 1 degree.

3.2 Pseudocode

To explain the written algorithm briefly a pseudocode of the simplified version of the Python script is placed below. This pseudocode does not explain the implementation of different brick sizes or the extra openings and rotations on certain layers.

1. Function to create a brick, which requires: the brick type, a curve, a point on the curve, a rotation in degrees to rotate all the bricks on this curve.
    1.1. Consider if whole or half a brick should be created. Plus check on the specified point the angle of the curve and the normal of the surface.
    1.2. Move the brick outwards until the edge of the brick touches the surface.
    1.3. Return a box which represents the brick.
2. Create an empty list which will hold all the created bricks.
3. Define the start point of the arm of the compass.
4. Create a bounding box around the surface.
5. Construct a plane just above the bottom of the bounding box.
6. Loop until the plane reaches the top of the bounding box.
    6.1. Intersect the plane with the surface.
    6.2. Calculate how many points should be created on the intersection curve. The amount of points should be equal and based on the length of half a brick.
    6.3. Divide the curve in a list of points, only if the row number is even include the endpoints. These end points might be needed later if half bricks need to be placed.
    6.4. Disregard all odd points from the list of points.
    6.5. Check if the row number is even and if the curve is not closed.
        6.5.1. Call function to create a brick and give variable to create a half brick. Add brick to the list of bricks.
        6.5.2. Delete the outer points from the list.
    6.6. Loop through every point which is still in the list of points.
        6.6.1. Call function to create a brick and give variable to create a full brick. Add brick to the list of bricks..
    6.7. Get the top face of the first brick on this curve. Then get the edge centre point which is closest to the start point of the arm of the compass.
        6.7.1. Move the plane up to the same height as this point.
7. Calculate the data for each brick relative to the start point of the arm of the compass.
    7.1. Calculate the arm length in mm and round it to 10 mm.
    7.2. Calculate the inclination (vertical) angle and round it to 0,1 degrees.
    7.3. Calculate the azimuth (horizontal) angle and round it to 0,1 degrees.

3.3 Algorithm information

It takes about 0,7 seconds to generate the initial 3.653 bricks and all the data required.

The Grasshopper file and the Python script can be found at: https://gitlab.com/Prethvi/team2-skill-centre/tree/master/Skill%20Center/Compass

3.4 Future improvements

The developed algorithm generates bricks aligned with the surface. There are two challenges with this approach. Firstly, the surface needs to be smooth, so a mesh should be really fine and the normal of each face needs to be properly set otherwise the bricks are placed in a very odd manner. Secondly, the settings of the compass for each brick can deviate, which will take a lot of extra time when constructing it on site. In order to make an implementation of this technique realistic, the surface should be designed with the parameters of the compass in mind so the compass does not need to be adjusted for every brick. Another option would be to let the algorithm rationalize the data so the settings of the compass do not need to be changed to often. The potential downside of the latter option is that there is less control of the shape and unwanted changes can occur.

4 Voxelization

Voxelizing the building and placing bricks on these voxels will help the bricklayers to construct the building as if it were a building designed with LEGO®. Due to limitations of computational power only a part of the building is voxelized with the written algorithm.

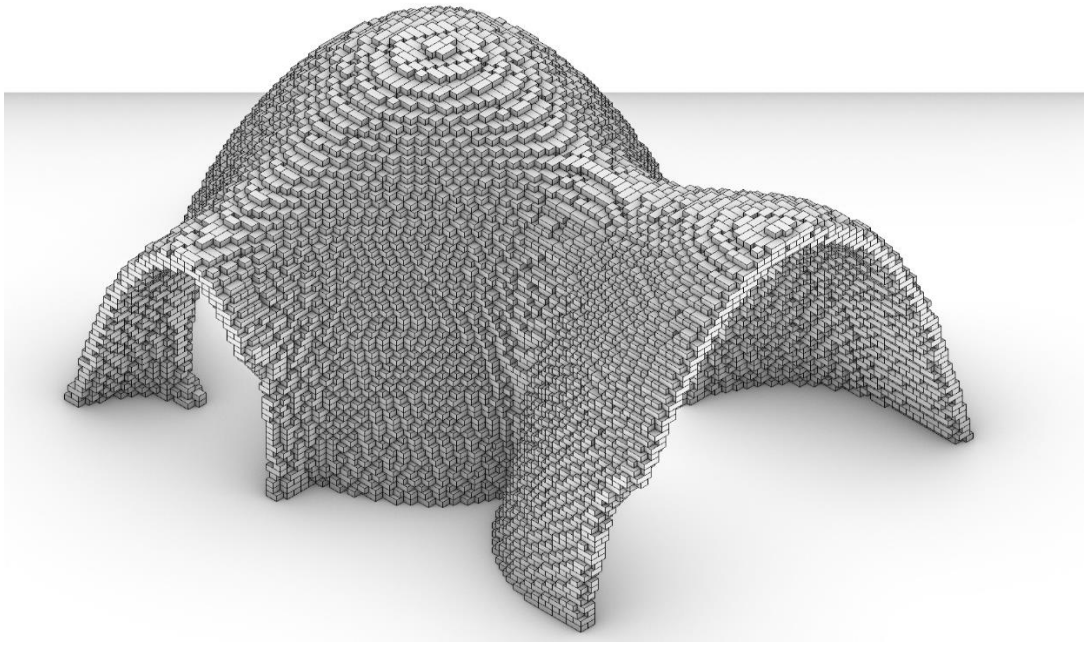


Figure 9: Output of the algorithm, bricks placed on a voxelized mesh

As an example a top view of the first and second layer generated by the algorithm are shown here below. In the second layer, the contours of the previous layer can be seen in magenta which makes it even more easier for the masons to construct the building.

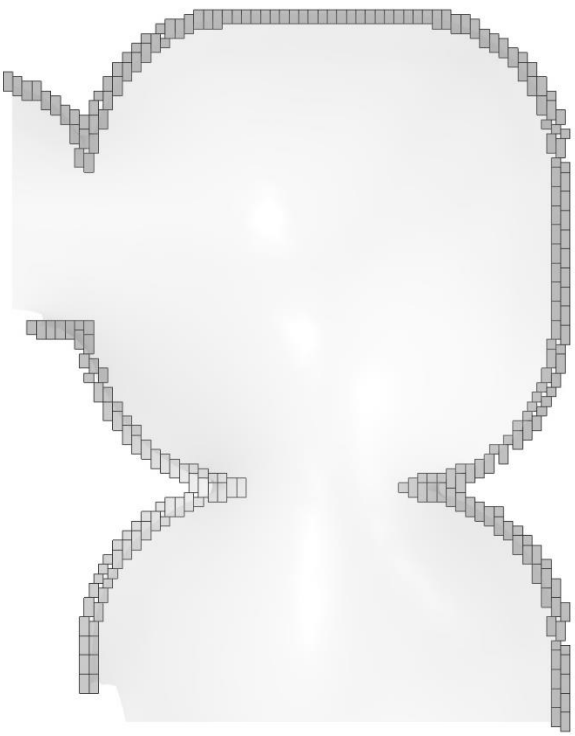


Figure 10: First layer

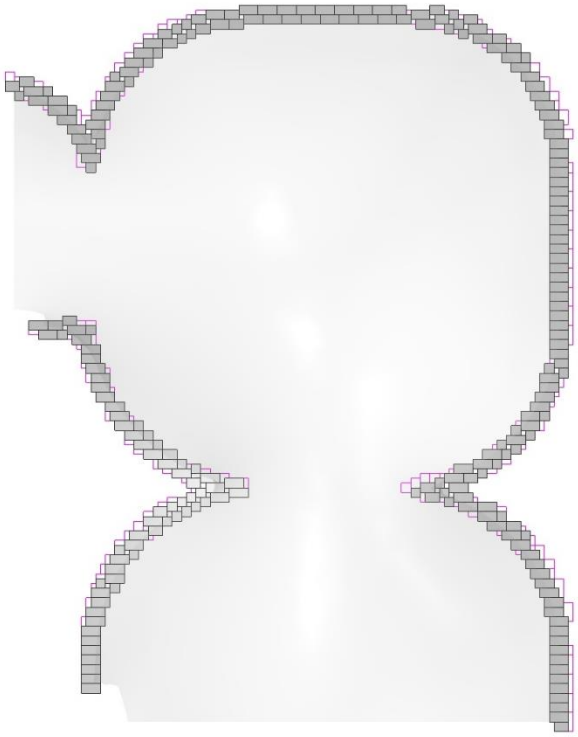


Figure 11: Second layer, contours of previous layer are visible

4.1 Building method limitations

It is not possible to create the desired shape with the specified bricks using voxelization. Bricks will fall because they are not supported by other bricks as can be seen in the section below. A possible solution could be to disregard a number of the top layers or construct the roof using a different method or material. Another potential option could be to use bricks which are less high. It would allow creating openings with a more graduate slope, as what is required at the top layers. However, it is unsure if flat bricks would be strong enough to transfer the loads. A different solution would be to change the shape, to make the section look like a pointed arch.

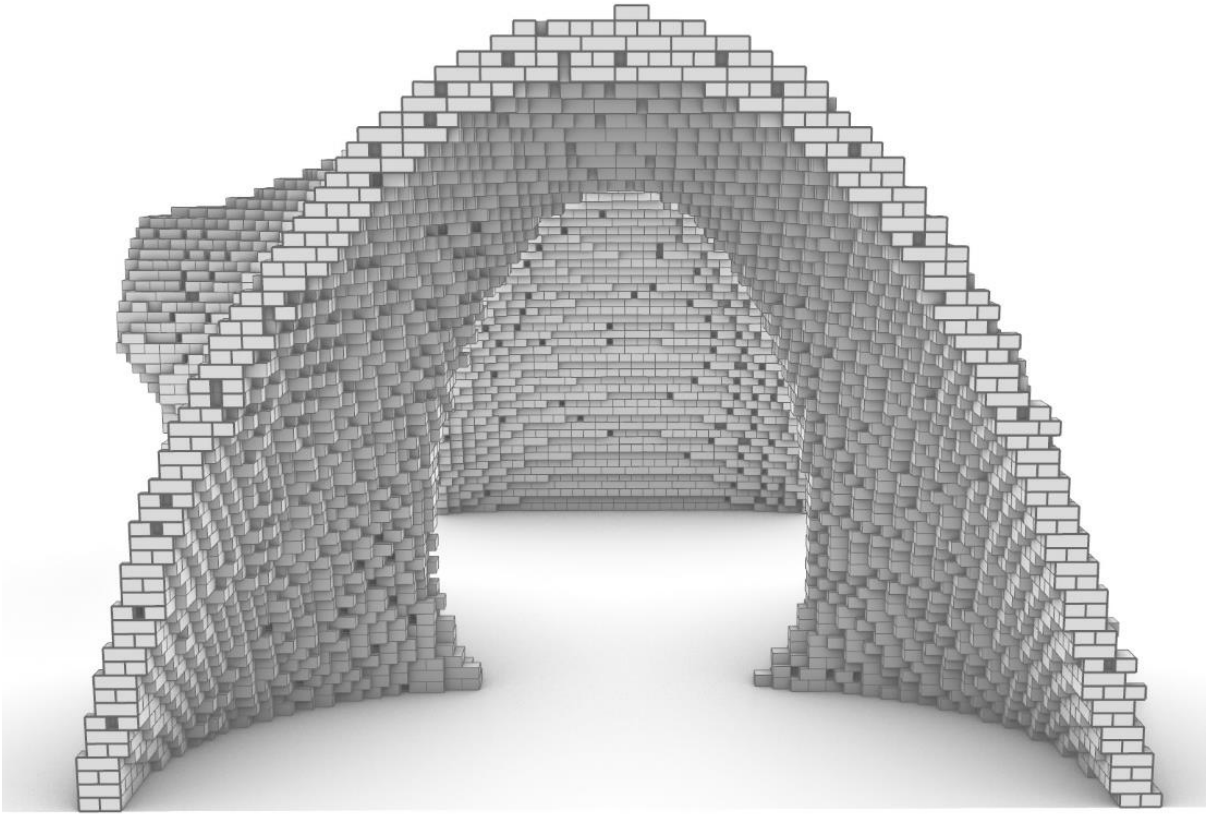


Figure 12: Section trough the generated bricks

4.2 Pseudocode

To explain the written algorithm a pseudocode of the Python script is placed below.

1. Define voxel size.
2. Create bounding box around the mesh.
    2.1. Increase size of bounding box to match voxel size.
3. Create voxel centre points in the whole bounding box. Structure the data as a 3D grid where [0],[0],[0] would refer to the first voxel and [0],[0],[1] refers to the voxel right above it.
4. For each voxel centre point check the distance to mesh.
    4.1. If the distance is outside the given range set the reference to that point (so [0],[0],[1] for example) to False.
5. Go through all layers. For even layers structure the data aligned with the X-direction and for odd layers structure the data aligned with the Y-direction.
    5.1. For each voxel reference (for example u=5; v=9; w=3; [u],[v],[w]) in the layer check if the reference is not False. If it is the case get the centre point of the neighboring voxels from the data structure. To create a full size brick all points between the references [u],[v],[w] and [u+2],[v+4],[w] are needed.
    5.2. Check if enough voxel centre points are available to place a full brick. Else see if there are enough points to place a ¾ brick. Else check a ½ brick.
        5.2.1. If a brick can be placed set all the references of the voxels inside the brick to false, so the bricks won't intersect.

4.3 Algorithm information

It takes about 18 seconds to generate 1.119.960 voxel centre points and 13.089 bricks.

The Grasshopper file, Python script and two animations showing all the brick layers can be found at: https://gitlab.com/Prethvi/team2-skill-centre/tree/master/Skill%20Center/Voxelize

4.4 Future improvements

As can be seen at the section on the previous figure, there are holes in between the bricks. The algorithm could be improved to prevent having gaps inside the structure. Besides that, a check could be added to ensure at least half of the brick is resting on top of other bricks. Another two improvements which will contribute to the constructability would be to use a different colour for every brick size and generate numerical output containing the amount of bricks per layer.

The following improvement has the potential to drastically decrease the required processing time, because it will generate a significant lower number of points compared to the original algorithm. Instead of creating all the voxels within the bounding box immediately, write a dividing boxes algorithm. To get an idea of how this might be achieved the following pseudocode is written:

1. Create a bounding box around the mesh. Scale the bounding box up to the closest power of two voxel size, so 2, 4, 8, 16, etc. This ensures when the boxes will be divided in step 3 it ends up with boxes of precisely a single voxel size.
2. Add the bounding box to a list of boxes.
3. Keep looping until the list of boxes is empty.
    3.1. Loop through each box in the list.
        3.1.1. Delete this box from the list of boxes.
        3.1.2. Check if this box is within the specified distance to the mesh (they are marked using a green ✓ on the figure below, the ones which are too far are shown with a red **X** and are disregarded).
            3.1.2.1. Check if the length of the diagonal of this box is at least two times the length of the diagonal of a single voxel.
                3.1.2.1.1. Divide the box in four equal boxes.
                3.1.2.1.2. Add the four boxes to the list of boxes.

In the figure below this process is visualized. The grey curve represents a section through the mesh. Using black different line types the boxes are shown.

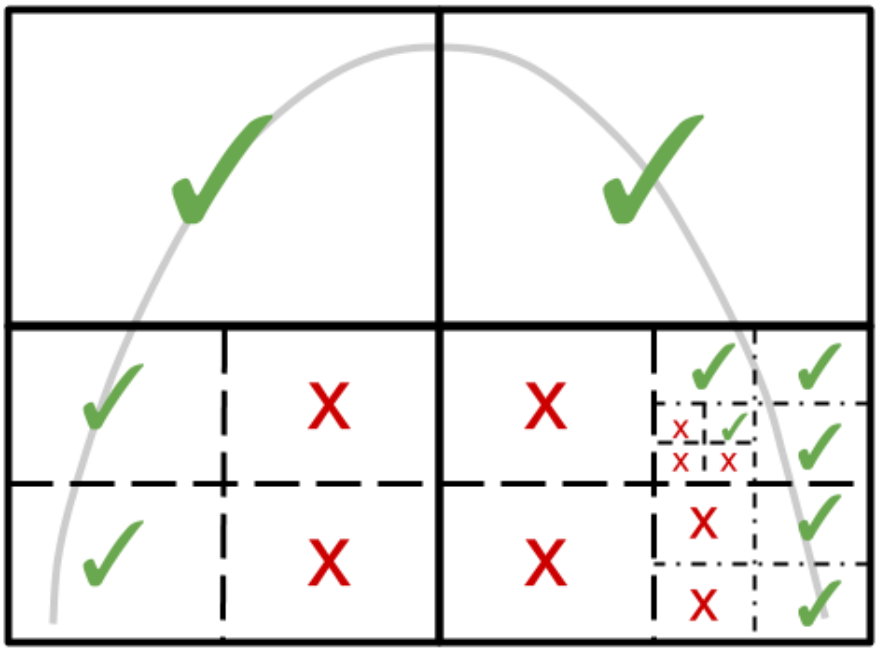


Figure 13: Suggested improvement for the voxelization process