Reconfigurable Domes

Computational design of dry-fit blocks for modular vaulting

Selina Bitting[1], Shervin Azadi[2], Pirouz Nourian[3]
[1,2,3]Delft University of Technology
[1]s.bitting@student.tudelft.nl [2,3]{s.azadi-1|p.nourian}@tudelft.nl

In contrast to the contemporary aesthetic account, Muqarnas are geometrically complex variations of Squinches used for structural integration of rectilinear geometries and curved geometries. Inspired by the historical functionality of Muqarnas, we present a generalized computational workflow for generating dry-fit stacking modules from two-dimensional patterns in order to construct a dome. Similar to Muqarnas these blocks are modular in nature, complex in geometry, and compression-only in their structural behavior. We demonstrate the design of such structures based on the exemplary Penrose pattern and showcase the variations & potentials of this method in comparison to conventional approaches.

Keywords: *Muqarnas, Generative Design, Modular Design, Unreinforced Masonry Architecture, Penrose Tiling, Workflow Design*

INTRODUCTION

Muqaranas (Tabbaa 1985) are an archetype of the middle-eastern architectural vernacular, often recognized by their intricate design. They are created using various modules, and are difficult to replicate without specific knowledge and materials due to their intricacy as well as the relevant construction techniques. Muqarnas originated as a structural method for transitioning from a rectilinear space to a dome, as a geometrically more complex squinch (Koliji 2012). This method of using modules to transition between different types of geometries is the inspiration behind the proposed generative method as it takes into consideration the force flows through the topological connections of said modules (Chassagnoux 1970). Additionally, the different techniques used to understand and replicate traditional muqarnas are essential to the generalization of the construction method into a computational process.

Traditionally, muqarnas use a pattern which specifies the juxtaposition of modules in the horizontal plane. By reading the pattern, it is apparent which modules are located where in order to create the overall dome or squinch (Yaghan 2001, [4]). This relationship between pattern and block is replicated in the computational process which results in a dry-fit, pre-cast set of interlocking blocks that are held together via gravity; thus resulting in a compression-only structure. In order to translate the muqarnas into such a method, two vital elements are required: a 2D pattern and a specification of how the blocks relate to the pattern.

The generative method (see Algorithm 1 in Figure 1) begins with the creation of a Penrose pattern

Algorithm 1: Computational Muqarnas Procedure

```
ComputationalMuqarnas ():
    Tessellate a polygon of a given diameter with Penrose pattern
    Mark the vertex to be identified as an apex point in the pattern
    Find the centroids of all geometries belonging to the tessellation
    Organize all geometries radially from the apex point
    Enumerate all geometries by distance from the apex point and
      sequence within said group
    Traverse the tessellation (pattern) and identify ordered pairs
    Extrude each geometry of the pair according to its typology
    Displace the pair of extrusions to the corresponding height for
      their location in the pattern
```

Figure 1
Computational Muqarnas Procedure

[2], then navigates the pattern, and later extrudes the modules. The Penrose pattern is exemplary, but any other 2D pattern can be utilized instead (Castera 2003). This specified relationship between the pattern and the block results in a system of overlapping, stacked modules with an overall design that is indicative of traditional muqarnas (Sakkal 1988), but is general in its implementation. The modules are simple enough in design to be constructed with neither particular knowledge about materiality nor additional equipment.

In order to evaluate the different potential methods for creating the modules, three main criteria are identified as priorities for the design to accomplish. The first is to limit the number of distinct modules to a maximum of 20, and the second is to obtain the maximum amount of variation using these modules. This variation should result in an interesting pattern that has a similar aesthetic to the original muqarnas. The final criteria is that the method of construction be simple enough for a person to build it themselves, which therefore rules out structural systems in which one block is held in place by neighboring blocks, as one example, to instead favor stacking methods.

The motivation behind the proposed computational method presented in this paper is to explore and validate the feasibility of procedural design of muqarnas-like discrete blocks for low-tech modular construction of vaults/domes by means of dry-stacking. The remainder of the paper presents an exemplary implementation of a reproducible method for generating the 3D geometry of the blocks based on a Penrose tiling (a 2D tessellation) of the floor plan of the dome to be muqarnas-vaulted, based on the following meta-procedure:

BACKGROUND

Domes present a number of challenges to modularization in that while they are constructed out of standard bricks or stone blocks, successfully doing so requires expert knowledge of how to achieve strength and stability through ad hoc placement of bricks. Therefore a dome structure is either constructed by a highly skilled mason or imitated through some other material such as steel. Incorporating the structural performance in a computational method allows the block geometries to account for the flow of forces in the dome and to design a stacking system that accomplishes a similar form that is easy to construct. By doing so the process can be expanded to shapes that are not just circular domes but can also replicate transitional spaces such as squinches.

In contrast to conventional bricklaying methods, existing dry-fit systems incorporate structural evaluation in their design to confirm their validity, rather than gluing blocks together or using a mortar between them. Two such examples are the Armadillo Vault constructed by the Block Research Group (BRG) of ETH Zurich [5] and Block Moulds [6]. The Armadillo Vault is a complex example whereas Block Moulds is a more practical implementation of a dry fit system. However, both exemplify the spectrum of dry-fit sys-

tems and how simple or complex constructions can become relatively easy to construct and possible to disassemble.

In an effort to create structural muqarnas, referred to as "genuine" muqarnas by Yaghan in his paper "Self-Supporting Genuine Muqarnas Units", the first option is to use a radial method of creating blocks. Figure 2 shows the radial slicing in both plan and section, which leads to a basic block as shown in the same figure. The next step is to carve or add to the inner face of this block in order to make a pattern reminiscent of traditional muqarnas. However, due to the varying radii as slices are made along with the dome, this method results in far too many modules and thus fails the first of the design criteria. This method is based on a simplification of the method used by Yaghan which is an early example of generating genuine muqarnas computationally.

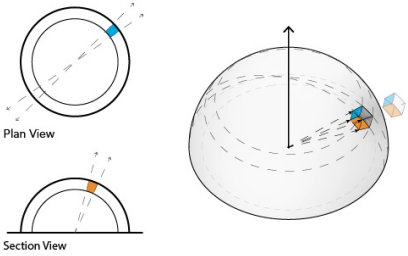


Figure 2
Diagram of the radial approach.

The second option is to use a stacking method derived a careful study of the vaulted dome of the "Masjed-e Jameh mosque" (Friday mosque) in Isfahan, Iran. The repeating shapes of the stone are simple and serve a structural purpose. These shapes are created using multiple simple, rectangular bricks. The main downfalls of this precedent are that it functions at a very large scale and that the system is not compression-only. A projected 2-Dimensional (2D) pattern of the Friday mosque dome demonstrates how each block sits over the top of a gap, showing that this design results in areas of tension. This indicates that it is not optimized for a structure comprised of only masonry.

A different approach to generating muqarnas is presented in Alaçam (Alaçam and Güzelci 2016) where the outline of the space is used to develop a plan projection. The pattern is generated by subdividing this outline into triangles. Afterwards, a surface is produced corresponding to the plan. This is a different approach which does not result in 3D blocks, and would therefore require additional steps to transition from a surface to modules.

The 2D pattern of the stacking approach resulted in a realization that virtually any pattern on the 2D plane can be navigated to generate block forms. In investigating the viability of this approach, a third option emerged. This option is to traverse patterns with different logics to generate geometries in a generalized workflow. The Penrose Pattern emerges as a promising pattern to couple with this pattern-navigating its 5-fold symmetry and lack of repetition. However, the entire pattern consists of only two types of triangles: kites and darts (Gardner 1989). This inherent modularity of the pattern meets the design criteria for modularity and variability, so the final criteria of structural self-sufficiency is all that remains for this method to fulfill all three design criteria.

Simple orthogonal extrusions of the triangles confirms that the variability is preserved in the 3-Dimensional (3D) form, however the structural criteria remains unfulfilled. This method is therefore accepted as the most promising approach, and the next challenge is to navigate the pattern in a way that produces a free-standing form which fulfils the structural design criteria.

FRAMEWORK

Traditionally, muqarnas are superficial rather than integrated structural elements of a building commonly made of wood or stucco. The blocks are then essentially "glued" to the building via additional stucco or nails (Yaghan 2007) according to a 2D plan projection, which is the same process as with their structural counterparts. The computational method focuses on this particular element to expand and generalize the process in order to generate any number of patterns or tessellations into domes, or possible vaults.

Within the scope of this paper, we assume that the final form is a dome-like structure, with the outer ring as the main support and the apex in the center. Consequently, the symmetry of the Penrose Pattern becomes an advantage for this input condition.

METHODOLOGY

In this section we will present the details of the computational workflow in the following order: construction of the Penrose pattern, sorting the modules to model the topological structure of the pattern, identification of the rings in the topological structure, navigation of the pattern, and finally construction of the 3D mesh of each one of the modules. The inherent modularity and variability of the Penrose Pattern prove it to be favorable, as it provides a limited set of modules while preserving the variety in the designs. Each structural module is constructed out of two triangular tiles in the Penrose pattern to ensure stackability. The pattern is navigated from the outermost edges to the centroid in order to create pairs of triangles. This logic allows modules to overlap, and therefore support each other structurally.

Step One: Pattern Generation

The initial step is to dynamically generate a Penrose Pattern. Given a diameter, the number of subdivisions, and the number of sides to the initial polygon, an initial set of acute triangles are created (see figure 3)

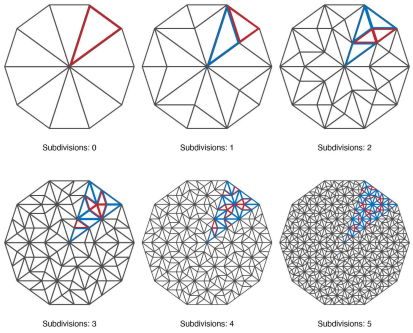


Figure 3
Generated Penrose pattern subdivision method illustrated.

$$\phi = \frac{1+\sqrt{5}}{2} \tag{1}$$

The algorithm which performs this function is largely based on a python script previously written by Josh Preshing [1]. Within his implementation, triangles have two options for color: red and blue. Red indicates an acute triangle and blue indicates an obtuse triangle. The algorithm also stores the triangle corners as complex numbers, which contain the information of the (x, y) coordinates for each of the corners A, B, and C. These coordinates are stored as python complex numbers, meaning that the imaginary value of A corresponds to it's x coordinate, and the real value corresponds to it's y coordinate. The following shows the tuple with all subsequent information:

$$(\text{color}, A, B, C) \tag{2}$$

For the creation of a dome, the number of subdivisions is set equal to 5, the diameter is equal to 6.6 meters, and the number of sides is equal to 10. The algorithm stores the triangles as complex numbers in creating the acute triangles, to then perform the subdivision algorithm with those triangles using an algorithm which subdivides an input triangle by the golden ratio ϕ , and then appends the relevant data, *(color, A, B, C)*, of the newly generated triangles to the results and outputs the list of all the newly subdivided triangles.

Algorithm 2, figure 4, begins with creating a polyline that connects the three corners of the triangle. Then the centroid of the triangle is found using to following equation [3]:

$$O_x = \frac{A_x + B_x + C_x}{3} O_y = \frac{A_y + B_y + C_y}{3} \tag{3}$$

This centroid is essential to understanding the location of the triangle in relation to the center of the overall pattern. The distance to the center of the pattern provides a method of constructing a triangulation between the centroid of the pattern, the centroid of the triangle, and the horizontal seam in the

Figure 4
Triangle Data Algorithm

Input	Data Type	Input Name: Notes
$triangles$	List of Tuples	The subdivided triangles, in this case subdivided by the golden ratio as input for this algorithm
Output	**Data Type**	**Output Name: Notes**
$singleTri$	List of polygons	Within the list are polylines or meshes related to the generated pattern triangles. In this case, polylines.
tri_info	List of floats	This list of lists contains the distance from the triangle centroid to the pattern centroid, the arcsine and the cosine thereof

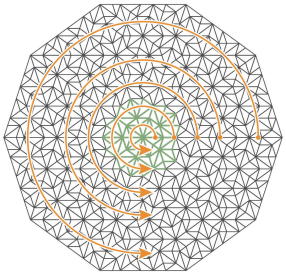
Problem: Draw the triangles resulting from the subdivision, then find the centroids of the shapes in order to obtain distance and radial data to later sort and locate the shapes.

Algorithm 2: Triangle Data Algorithm

```
1  TriangleData (triangles):
2      foreach color, A, B, C in triangles do
3          Apt ← map x,y,z to 3D plane
4          Bpt ← map x,y,z to 3D plane
5          Cpt ← map x,y,z to 3D plane
6          Dpt ← calculate centroid of the triangle and map x,y,z to
             3D plane
7          DSa ← arcsine of the centroids
8          DCa ← arcosine of the centroids
9          singleTri ← construct polyline geometry
10         tri_info ← append relevant locating data to the list
11     return singleTri , tri_info
```

Figure 5
Diagram of the triangle sorting.

pattern (as can be seen in figure 5), to then locate the triangles in relation to one another. The arcsin and arccos provide the information needed to sort triangles radially counterclockwise starting at the seam in the pattern, which is performed in the following step of the code.



Input	Data Type	Input Name: Notes
aw	Integer	Weighted number conveying the importance of the angular location, , in the implementation 1 was used
dw	Float	Weighted number conveying the importance of the distance from the pattern centroid, in the implementation 10^8 was used
Output	**Data Type**	**Output Name: Notes**
$order\_values$	Weighted Sorting List	This list contains a weighted value which locates the triangle in a relative radial system rather than a standard Cartesian system.

Algorithm 3: Triangle Sorting Algorithm

```
1  TriangleData (s, d, n):
2      foreach ti in tri_info do
3          sdeg ← convert arcsine to degrees
4          cdeg ← convert arcosine to degrees
5          if 90 ≥ sdeg ≥ 0and 0 ≥ cdeg ≥ 0 then
6              deg = sdeg
7          else if 90 ≥ sdeg ≥ 0  and  180 ≥ cdeg ≥ 90 then
8              deg = cdeg
9          else if 90 ≥ sdeg ≥ 0  and  180 ≥ cdeg ≥ 90 then
10             deg = cdeg
11         else if 0 ≥ sdeg ≥ −90  and  90 ≥ cdeg ≥ 0 then
12             deg = 360 - cdeg
13         else if 0 ≥ sdeg ≥ −90  and  90 ≥ cdeg ≥ 0 then
14             deg = 360 + sdeg
15         order_value = aw · deg + dw · ti[0]
16         order_values ← append order_value to a list
17     return order_values
```

Figure 6
Triangle Sorting Algorithm

Figure 7
Pattern Navigation Algorithm

Input	Data Type	Input Name: Notes
l	Integer	This number indicates which level or ring the geometry in list tris is located at
$tris$	Geometry	A list of the triangle geometries ordered by location
Output	**Data Type**	**Output Name: Notes**
all_pairs	Ordered List	This list contains the triangle geometries paired together to form the resultant pairs of shapes from the pattern navigation

Problem: Sort the arcsine and cosine data according to what quarter of the circle they lie in

Algorithm 4: Pattern Navigation Algorithm

```
1  TriangleData (s, d, n):
2      tris_leveled ← initiate an empty list
3      this_level ← initiate an empty list
4      foreach i, t in enumerate(tris) do
5          this_level ← append t
6          if levels[i] != levels[(i + 1)%len(tris)] then
7              tris_leveled ← append this_level
8              this_level ← set equal to an empty list
9      order ← list of pairs which are the manually determined pattern
10     all_pairs ← initiate an empty list
11     foreach o in order do
12         first_pairs ← index o[0] in tris_leveled
13         second_pairs ← index o[1] in tris_leveled
14         this_pairs ← zip the first_pairs and second_pairs data
             together
15         all_pairs ← append the this_pairs value
16     return all_pairs
```

Step Two: Triangle Sorting

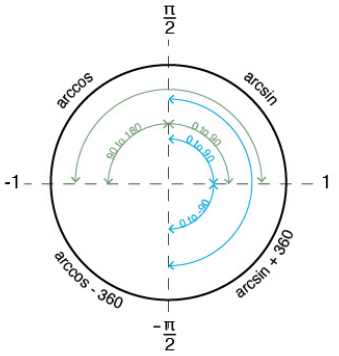
The values of the distances along with the arcsin and arccos are the inputs to algorithm 3, figure 6, which gives priority to the sorting method. In this algorithm, distance has higher importance than angle $(dw > aw)$, which allows the sorting of the individual triangles to be from the center of the pattern to the outside, as well as counterclockwise from the horizontal seam on the right (figure 5).

For the organization portion, the algorithm uses the information previously appended to *tri_info* in order to locate the points radially. These values are converted to degrees, and then ranges are defined in order to determine in what quadrant of the circle the

triangle is located. To elaborate, the domains of arcsin and arccos are shown below, and illustrated in figure 8:

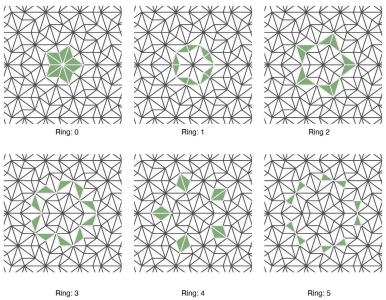
$$\arcsin\{x\} \Rightarrow \left[\frac{\pi}{2}, \frac{-\pi}{2}\right] \tag{4}$$

$$\arccos\{x\} \Rightarrow [0, \pi] \tag{5}$$



Associated with arcsin is the y-coordinate, and associated with the x coordinate is arccos. Therefore all four quadrants of a circle are included by evaluating them together. Referring to figure 8 can help to visualize the sorting code.

Next, the deg values are defined according to which quadrant each triangle is, and thereafter each triangle is sorted throughout the entire pattern according to degrees and distance from the center.



Step Three: Pattern Rings

Triangles equidistant from the center are located within the same ring, see figure 9. All triangles are first sorted by indexing each triangle. Then the distance of the triangle prior and the current triangle to the centroid are compared to determine if they are in the same ring, in which case the index increases by 1.

Step Four: Pattern Navigation

The method developed for the pattern navigation is to select a pair of adjacent triangles and then define one of those triangles as an angled piece and the other as a flat piece (algorithm in figure 7). This process is performed manually through trial and error, as not all combinations allow the pairs to overlap in a way which reaches the center of the pattern. Ideally this process begins at the outer edge and works toward the center until all shapes are incorporated in a sequence.

A single module is comprised of a flat piece and an angled piece. The flat piece is the largest part of the module, best described as an orthogonal extrusion of the triangle. The angled piece is the part of the module which is extruded to a point. The point in question is the corner of the triangle closest to the centroid of the pattern. The next pair of triangles form the following module, wherein the current angled piece becomes defined as the flat piece for the next module and the next adjacent triangle is the angled piece of the module, and so on and so forth until the centermost triangles have been included in the extrusion process to make modules. In this way, flat pieces sit overtop angled pieces, to form a system of modules that stack and distribute their load downwards to the next module. The method is elaborated upon in step six, where the modules are extruded.

The first section of the algorithm iterates through the rings such that each ring is given an index, and the triangles within the ring are indexed as well. The second section, for *o* in *order*, takes each pair in the manually determined pattern navigation and identifies those triangles in the list *tris_leveled*. The next step is to take the first and second triangles and com-

Figure 8
Diagram showing the sorting of calculated values into circle quadrants.

Figure 9
Grouping triangles into rings.

bine them together so that a triangle with index 0 of level 0 is paired with the triangle of index 0 of level 1, and so on. The result is pairs of triangles that step their way from the centroid out. Figure 10 shows one of these 'strings' beginning at the outside of the pattern and working its way inwards. Figure 11 shows the adjacent 'strings' as well for context.

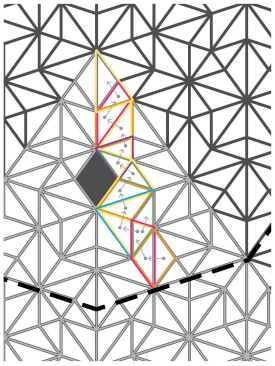


Figure 10
Plan showing a singular string of triangle pairs progressing towards pattern centroid.

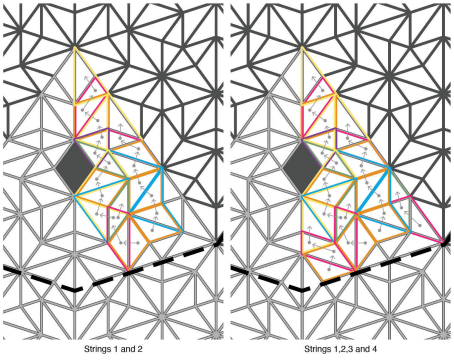


Figure 11
Plan showing adjacent, shorter strings.

Step Five: Adjustments for Triangle Displacement

The 'strings' within the paired triangles have different quantities of modules. The steps in the process which displace and extrude the modules rely on an initial displacement of the triangles to their relevant distances from the top of the dome. This can be seen in figure 10 where some strings extend from the outside to the very center of the pattern. However, there is, for example, a string that ends approximately three rings from the top. This translates to:

$$z = 3 \cdot bH \tag{6}$$

Therefore the values are manually redefined so that the associated index (i) of the strings accounts for the distance from the top of the dome. This updated value is instead stored as a new value, *z*, so that the original *i* is not lost. The modules are then vertically displaced according to this value.

Step Six: Module Extrusion

Each module, as explained before, consists of an angled and a flat piece. A for loop iterates through the list of pairs to perform basic extrusions on the triangles based on whether they are the angle or flat piece. Regardless of the type of piece associated with the triangle, the initial step is to retrieve the vertices of the triangles.

Angled Piece. For the angled piece, the next step is to sort the vertices and determine which vertex is closest to the center of the pattern, now referred to as the origin. The input for the *getCorners* function is the polyline of the angled piece, and the output is *(cornerPoints[0],cornerPoints[1], minPoint)* where *minPoint* indicates the vertex closest to the origin.

This output provides the two inputs, *(cornerPoints[0],cornerPoints[1])*, for the *duplicatePoints* function. This function displaces the vertices vertically by the block height *(bH)*. This provides the final data needed to construct a vertical polyline rectangle to be extruded toward the vertex closest to the origin. This extrusion is performed in a loop that iterates over the pairs of triangles. This concludes the process for creating the angled portion of the module, visualized in the top sequence of images in figure 12.

Flat Piece. A similar yet simpler process constructs the flat pieces of the modules. For the flat piece an arbitrary curve is used as the input for the *ExtrudeCurveStraight* function to produce an orthogonal extru-

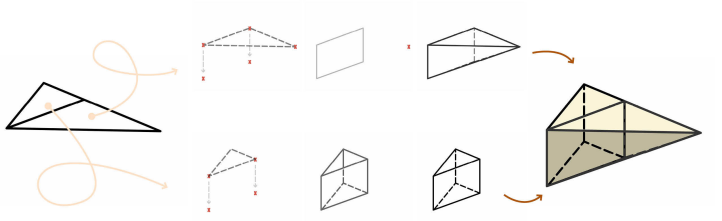


Figure 12
Diagram showing the steps performed by the algorithm to extrude the module geometries.

sion. The initial step is to retrieve an arbitrary corner using the function *getOneCorner*. Then, the function *duplicatePoint* makes a duplicate point displaced by block height *bH*. This results in the orthogonal line to guide the extrusion of the flat piece. The resultant flat portion of the module is shown in the bottom sequence of images in figure 12.

Displacement of the Modules. The displacement of the modules is performed in the negative z-direction, due to the overall methodology first addressing geometries near the pattern centroid and progressing outwards.

The flat and angled pieces are displaced individually by the same distance. The adjusted levels list is used in order to define the magnitude of the displacement, and all geometries are moved accordingly. The method of stacking can be seen in figure 13 in order to clarify how the 2D plan translates to a 3D form.

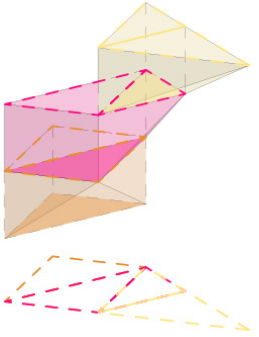


Figure 13
Diagram showing three stacked modules, and their original geometries projected below.

DISCUSSION & FUTURE WORK

An overarching element which the methodology lacks is compatibility with a variety of inputs. The process as exists at this stage, the inputs limit the variety of results, which could otherwise expand in scope to explore a variety of patterns and block shapes. The traversal scheme, currently manual, is essentially represented with tree graphs. Thus, one future direction is to utilize graph traversal methods, particularly breadth first search, to pair triangles of the Penrose pattern and generate modules.

Although the results show modules which can be stacked and result in a free-standing dome, there is no structural evaluation incorporated within the methodology. This is an essential step to be incorporated in the future iterations of the methodology as it represents a major hurdle to translating the generated scheme into a physical construction. Two options for enhancing this portion of the methodology are the dynamic relaxation method [7] and the force density method (Schek 1974), both of which are form-finding methods which take the tessellated pattern as a network and relax it under a set of forces until it reaches equilibrium. Incorporating this step also allows the modules to be reshaped in order to approximate a relaxed form which accommodates for the material properties and structurally feasibility, ultimately optimising the form.

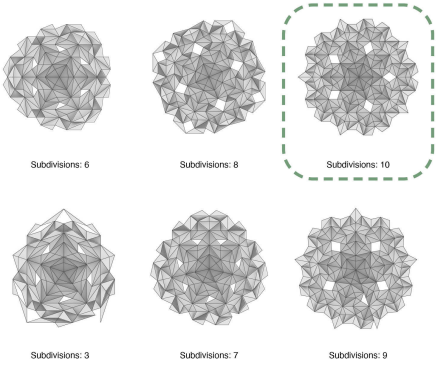
Figure 14 is our proposed meta-procedure for topologically generalizing the proposed method so that it can work on input shapes that are not necessarily as symmetrical as the presented polygon.

Figure 14
Road-map of Computational Muqarnas Procedure

Algorithm 5: Road-map of Computational Muqarnas Procedure

1 **ComputationalMuqarnas** *(*InputPolygon*)*:
2 Tessellate the floor plan of the space to be covered by muqarnas blocks as a mesh
3 Mark the [singular] vertices to be transformed into apex points in the discrete muqarnas vault
4 Find the faces incident to these vertices
5 Construct a dual graph encoding the connectivity between the face of this mesh
6 Run a graph traversal method to find a path from every apex vertex towards the nearest boundary edge.
7 For every edge of the found paths on the dual graph, choose the corresponding faces and extrude them into prisms
8 Optimize the shape of the dual graph by means of the Force Density Method and extract the resultant heights of nodes from the method
9 Move and duplicate a copy of the prisms above themselves into stepped distances in accordance with the heights found above
10 Coalesce every such pair of consecutive prisms into one 3D block
11 Scoop out excess mass out of the blocks from below.

Figure 15
Views from below of different generated domes.



The resultant blocks of this process will simplify and optimize block construction methods. Dry stacking modules allow for total demountability which is in congruence with the intentions and goals of the circular construction practices. Circular spaces or transitions between rectilinear and circular spaces can be generated with this modular method to provide the easy to construct dry-fit blocks. Additionally, a further generalization of the method has the potential to produce re-configurable vaulting systems (Alaçam et al. 2017). As this methodology was initially developed in the context of the MSc EARTHY Studio at TU Delft (Nourian et al. 2020), these blocks were originally intended to be made of cast earth blocks. Within that educational context, the main task was to develop a low-tech construction process for the domes of a heritage complex dedicated to Syrian heritage. Throughout the course it was determined that exploring the intersect of muqarnas and computation would be aligned with the intention behind the heritage complex. In designing the process which resulted in this paper, it was further realized that this process had the potential to be generalized and reformulated to qualify for other contexts based on the variety of spatial quality that it can potentially spawn and not just cultural significance. Nevertheless, the

methodology stays true to its origin as it is still compatible with using local material for manufacturing the simple blocks that are not only circular as they are dry-fit, but also are aligned with the skill level of the local workforces as their assembly process is based on simply stacking them.

CONCLUSION

As opposed to traditional muqarnas modules which are attached to the building superficially for decoration, these computationally generated modules are produced for simplifying the construction process of a masonry vault by making it easy for people without sophisticated masonry skills to make complex form active shells out of a few types of easily mass-produced blocks. The proposed method relies upon prescribed pattern navigation which relates to the pattern navigation utilized in traditional muqarnas. The method and the forms generated by it are therefore derivations of the traditional designs rather than imitations, which was the original intention. A variety of designs are achieved, shown in figure 15, and show great promise should the methodology be further developed.

REFERENCES

Alaçam, S and Güzelci, O 2016 'Computational Interpretations of 2D Muqarnas Projections in 3D Form Finding', *no source given*

Alaçam, S, Güzelci, OZ, Gürer, E and Bacınoğlu, SZ 2017, 'Reconnoitring computational potentials of the vault-like forms: Thinking aloud on muqarnas tectonics', *International Journal of Architectural Computing*, 15(4), pp. 285-303

Castera, JM 2003 'Play with infinity', *Meeting Alhambra, ISAMA-BRIDGES Conference Proceedings*, pp. 189-196

Chassagnoux, A 1970, 'Persian vaulted architecture: morphology and equilibrium of vaults under static and dynamic loads', *WIT Transactions on The Built Environment*, 16

Gardner, M 1989, *Penrose Tiles to Trapdoor Ciphers ...and the Return of Dr.Matrix*, The Mathematical Association of America, Washington, DC

Koliji, H 2012, 'Revisiting the squinch: From squaring the circle to circling the square', *Nexus Network Journal*, 14(2), pp. 291-305

Nourian, P, Azadi, S, Hoogenboom, H and Sariyildiz, S 2020, 'EARTHY, Computational Generative Design for Earth and Masonry Architecture', *RUMOER*, pp. 47-53

Sakkal, M 1988, 'An introduction to Muqarnas domes Geometry', *Structural Topology 1988 núm 14*

Schek, H.J. 1974, 'The Force Density Method for Form Finding and Computation of General Networks', *Computer Methods in Applied Mechanics and Engineering*, 3, pp. 115-134

Tabbaa, Y 1985, 'The Muqarnas Dome: Its Origin and Meaning', *Muqarnas*, 3, pp. 61-74

Yaghan, M 2001, 'The Muqarnas Pre-designed Erecting Units: Analysis, Definition of the Generic Set of Units, and a System of Unit-Creation as a New Evolutionary Step', *Architectural Science Review*, 44, pp. 297-318

Yaghan, DMA 2007, 'Self-Supporting Genuine Muqarnas Units', *Architectural Science Review*, 48(3), pp. 245-255

[1] https://preshing.com/20110831/penrose-tiling-explained/

[2] https://www.maths.ox.ac.uk/node/865

[3] www.mathopenref.com/pageurl.html

[4] http://www.shiro1000.jp/muqarnas/default-.htm

[5] https://block.arch.ethz.ch/brg/

[6] https://www.blockmoulds.com/

[7] https://github.com/Pirouz-Nourian/VectoRelax_Vectorized_Dynamic_Relaxation/tree/v1.3