Algorithmic Generation of Interlaced Stellations

Fahreddin Başeğmez

Boston, USA; fbasegmez@gmail.com

Abstract

The Python-based 2D *simetri.graphics* library, SİMETRİ, enables users to apply symmetry transformations with ease. We discuss how to create star shapes, a classic example of rosettes with a rich history, and overview an interlacing algorithm for crafting intricate over-under patterns by using SİMETRİ. The algorithmic approach used is an alternative to the traditional grid-based or compass and straightedge methods commonly used to create star shapes in Islamic geometric patterns.

Introduction

Symmetry can be defined as invariance of an object under an operation. Symmetry in the Euclidean plane involves the manipulation of geometric objects by translations, reflections, rotations, and glide reflections. While there are several publicly available computational geometry libraries and programming tools that can be used to explore symmetry-related topics, most require a deep level of domain-specific knowledge and significant experience in programming. The open-source simetri.graphics library (see [2] for documentation and installation instructions), SİMETRİ, developed by the author, on the other hand, caters to novice Python programmers with a limited background in mathematics.

SİMETRİ is compatible with Linux, macOS, and Windows operating systems capable of running Python 3.9 or later and a XƎLATEX compiler. It serves as a versatile tool for creating technical illustrations (all the figures presented in this paper were generated using the SİMETRİ library), decorative designs, and generative art. The library features modules and facilities for generating frieze-patterns, wallpaper-patterns, tilings, turtle-graphics, Lindenmayer-systems, fractals, and parquet-deformations. Additionally, it includes a 2D geometric constraint solver, enabling the creation of circle-packing designs without the need for geometric formulas. Circle-packing can be utilized to construct extensive assemblies of star patterns. In this paper, we use star shapes with dihedral symmetry to demonstrate select capabilities of SİMETRİ. Rosettes can have dihedral or cyclic symmetry (see Figure 1). Dihedral symmetry encompasses both rotational and reflective symmetry and can be described as the symmetry of regular polygons.

The evolution of star shapes and their arrangements has a rich and extensive history, but the methods used in their creation are scarcely documented [8, 9, 10]. The remnants are the results of techniques passed down as secrets from father to son or master to apprentice. Modern books often explain the step-by-step creation of these shapes using regular grids (circular, hexagonal, square, etc.) or tools, like a compass and straightedge [4, 8, 12, 14]. By contrast, we illustrate our points with algorithmic and graphic examples along with the Python code that generates the graphics portion of the corresponding figure (see Figure 2).

Even though our approach is algorithmic, we are able to use grids to create kernels. The *CircularGrid* object in SİMETRİ provides methods to generate internal points by intersecting lines that connect the peripheral points of the grid (see Figure 3).

The Rosette Algorithm

Throughout, we use the term star interchangeably with the term rosette. For the interlaced form of their representation, we use the term *girih stars* (also spelled as gereh or gareh meaning ”knot” in Persian) [8, 9, 10].

```
import simetri.graphics as sg

def rosette(n, kernel, axis=sg.axis_x, cyclic=False):
    if cyclic:
        petal = kernel
    else:
        petal = kernel.mirror(axis, reps=1)
    star = petal.rotate(2 * sg.pi/n, reps=n-1)
    return star

canvas = sg.Canvas()
points = [(0, 0), (40, 0), (40, 40), (14, 40)]
points2 = [(0, 0), (30, 17), (60, 0)]
rosette1 = rosette(7, sg.Shape(points), cyclic=True)
canvas.draw(rosette1, line_width=2, fill=False)
rosette2 = rosette(5, sg.Shape(points2)).translate(130, 0)
canvas.draw(rosette2, line_width=2, fill=False)

canvas.display()
```

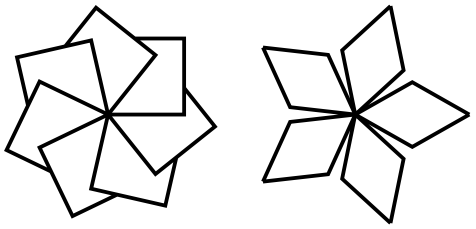


Figure 1: *Cyclic (C_7) and dihedral (D_5) rosettes.*

```
import simetri.graphics as sg

canvas = sg.Canvas()
Star = sg.stars.Star
radius = 50
gap = 300
for  i in range(10, 15, 2):
    star = Star(i, radius).level(4)
    star.translate(i * gap, 0)
    swatch = sg.random_swatch()
    lace = sg.Lace(star, offset=5,
                   swatch=swatch)

    canvas.draw(lace)

canvas.display()
```

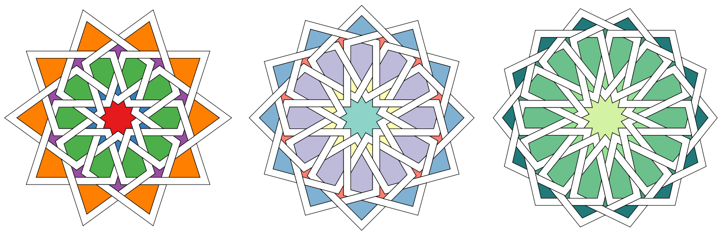


Figure 2: *Examples of traditional 10, 12 and 14-sided stars.*

Rosettes have no translational symmetry, and are considered to be *finite* (aka bounded) designs. They can have cyclic (only rotational symmetry) or dihedral symmetry (both reflective and rotational symmetry). The rosette algorithm in SİMETRİ has the following three main steps (Figure 4):

1. Form the starting shape (which we will call the *kernel*) and locate it in the first quadrant of the Cartesian coordinate system. Typically, the last point of the kernel lies on the x-axis (when assembling multiple stars, as illustrated in Figure 5, the kernel of a star may extend across the x-axis).
2. If dihedral, mirror the kernel about the x-axis to form a *petal*. If cyclic, the kernel is the petal.
3. Rotate the petal about the origin $n - 1$ times by $360/n$ degrees to form a rosette.

Once the kernel is identified, the second step of the algorithm creates mirror symmetry and the third step creates rotational symmetry (Figure 4). While rosettes with cyclic symmetry have handedness (aka chirality), those with dihedral symmetry do not (see [6, 7] for a detailed analysis of rosette designs).

```
import simetri.graphics as sg

grid = sg.CircularGrid(n=20, radius=100)
p1 = grid.intersect((0, 10), (2, 14))
p2 = grid.intersect((0, 6), (2, 14))
p3 = grid.points[0]
n=10
kernel = sg.Shape([p1, p2, p3])
petal = kernel.mirror(sg.axis_x,
                        reps=1)
star = petal.rotate(2*sg.pi/n,
                      reps=n-1)
lace = sg.Lace(star, offset=4)

canvas = sg.Canvas()
canvas.draw(lace, fill=False)
canvas.display()
```

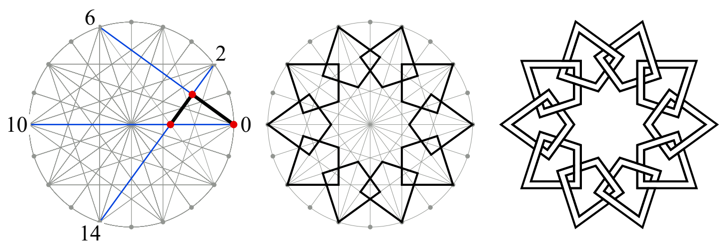


Figure 3: *Grid based star design. (The script generates the rightmost rosette.)*

Defining a P/A Type Star Kernel

The examples of stars used in historic buildings and artifacts exhibit commonalities in their design and proportions. Due to the scarcity of historic sources documenting the design process [8, 9, 10], many modern books use grid-based or compass-and-straightedge methods to recreate these patterns [4, 8, 12, 14]. In SİMETRİ, the number of petals represents a family of stars with different *levels* (stellates) (Figure 6). Figure 7 and Figure 8 summarize the different types of stars that can be formed by using two- and three-point kernels.

Starting with level-2 (level numbering is zero-based to align with Python conventions), we compute the locations of the three points that would form the level-2 kernel. All other levels of the same family (P/A type as shown in Figure 6) are expansions or contractions of this level.

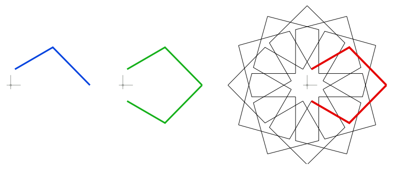


Figure 4: *Rosette: 1- Set the kernel, 2- Mirror it about the x-axis, 3- Rotate it about the origin.*

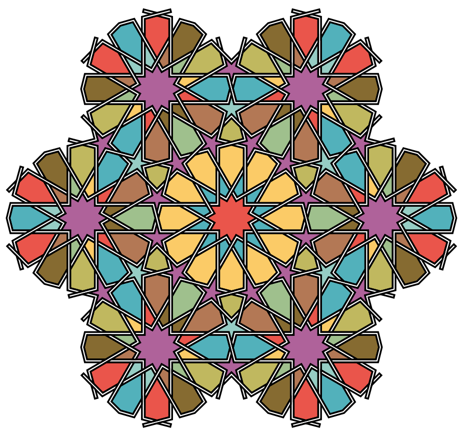


Figure 5: *An assembly of multi-kernel girih stars with a kernel crossing the x-axis.*

```python
import simetri.graphics as sg

canvas = sg.Canvas()
star = sg.stars.Star(n=10, inner_radius=20)

for i in range(5):
    canvas.draw(star.level(i), fill=False,
                                line_width=3)
    canvas.draw(star.kernel(i), line_width=5,
                          line_color=sg.blue)
    dx = 2 * star.level(i+1).circumradius
    canvas.translate(dx, 0)

canvas.display()
```

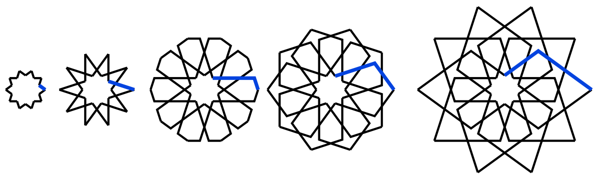


Figure 6: *The first five levels of a 10-sided star, with their corresponding kernels highlighted in blue color.*

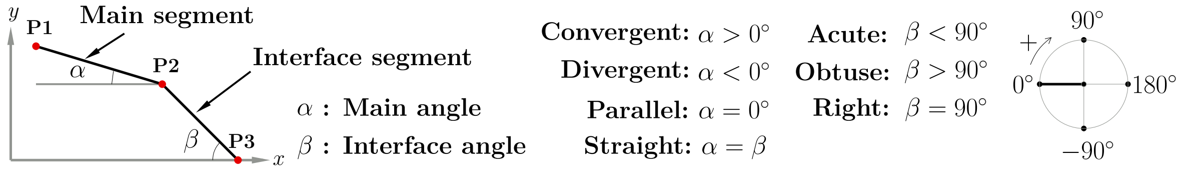


Figure 7: *Three-point kernel segments. Figure 8 uses the initial letters of the angle types for classification.*

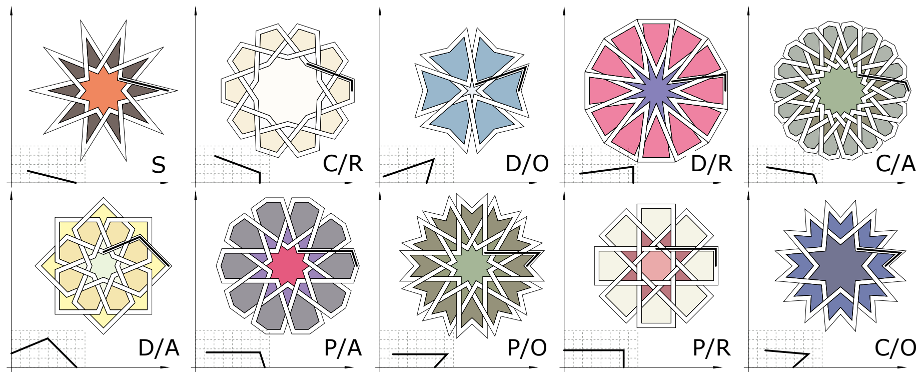


Figure 8: *Kernels: C:convergent, D:divergent, P:parallel, A:acute, O:obtuse, R:right-angle, S:straight.*

The algorithm to compute the kernel of a level-2 P/A type star in SİMETRİ includes the following steps (see Figure 9 for the abbreviations and code).

1. Compute α, β, γ, θ, t, and x using the number of petals and the radius.
2. Locate P_1 using the radius and θ.
3. Define $line_1$ as a horizontal line that goes through P_1 and $line_2$ as a mirror copy of it about the x-axis.
4. Rotate $line_2$ about the origin by γ radians and intersect it with $line_1$ to find the intersection point P_x.
5. Using the P_x, x, and β, locate the P_2 and P_3. (P_1, P_2, P_3) defines the kernel.

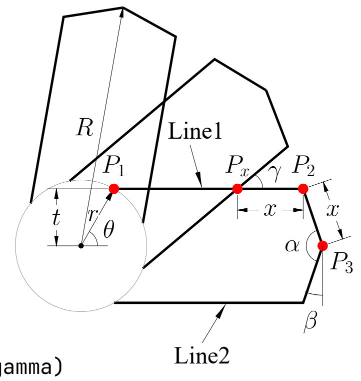
References [3], [5], and [6] provide a different method for computing the kernels of different types of star shapes. The method described above can be easily modified in SİMETRİ to cover most types of kernels depicted in Figure 8 as well.

```python
import simetri.graphics as sg

def get_kernel(n, radius):
    half_angle = sg.pi / n
    alpha = (n - 2) * half_angle
    beta = (sg.pi - alpha) / 2
    gamma = 2 * half_angle
    theta = 3 * half_angle
    t = radius * sg.sin(theta)
    x = t / sg.cos(beta)

    x1 = radius * sg.cos(theta)
    y1 = t
    p_1 = p1 = (x1, y1)
    p_2 = (x1 + 1, y1)
    p_3 = (x1, -y1)
    p_4 = (x1 + 1, -y1)
    line1 = (p_1, p_2)
    line2 = sg.Shape([p_3, p_4]).rotate(gamma)
    px = sg.intersect(line1, line2)
    p2 = (px[0] + x, y1)
    p3 = (p2[0] + (x * sg.sin(beta)), 0)

    return sg.Shape([p1, p2, p3])
```



n : Number of petals

r : Inner radius

R : Circumradius

t : Petal thickness / 2

x : Extension length

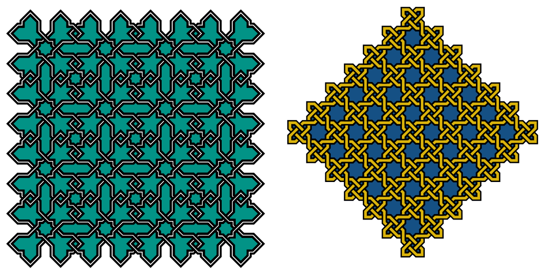
$$\alpha = (n-2)\cdot\pi/n$$
$$\beta = (\pi-\alpha)/2$$
$$\gamma = 2\cdot\pi/n$$
$$\theta = 3\cdot\pi/n$$
$$t = r\cdot sin\theta$$
$$x = t/cos\beta$$

Kernel $\equiv (P_1,\ P_2,\ P_3)$

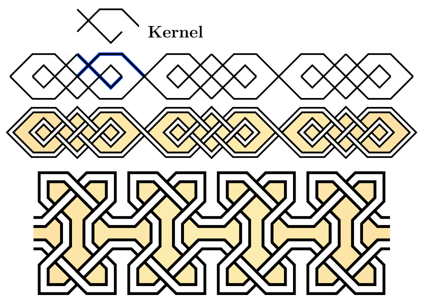
Figure 9: *Defining a P/A type star with level-2 kernel.*

Interlacing

Many ancient girih designs, as well as Celtic knots, are presented as interlaced patterns.These intricate patterns, often featuring over-and-under weaving, can be observed in a variety of historical artifacts and structures, including architectural elements, textiles, pottery, and manuscripts. Although our focus is on interlaced star shapes, SİMETRİ can be used to create interlacing figures of frieze-patterns, wallpaper-patterns, some tilings, and other decorative patterns. The Lace objects can be constructed by using closed or open polygonal chains, or both (see Figure 10, and Figure 11a). However, not all geometry is suitable for creating interlaced designs. If more than two segments intersect at a common point, SİMETRİ's interlacing algorithm cannot complete the process. See Figure 11b for two examples of ”malformed” geometry.



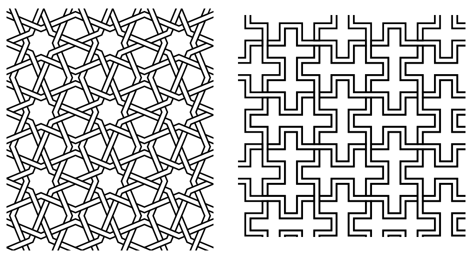
(a)



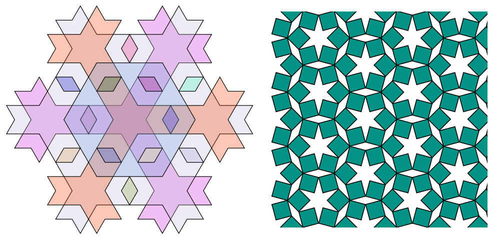
(b)

Figure 10: *Examples of interlaced wallpaper-patterns (a) and frieze-patterns (b).*

The topology of intersecting paths is explained in [1, 5]. A crucial step in our algorithm is the identification of the *Intersections* of main and offset *Divisions*, details of which are described in the Merging Line Segments and Sweep Line Algorithm sections. Figure 12 provides a step-by-step graphical representation of the algorithm employed in SİMETRİ, and the following section details each step.



(a)



(b)

Figure 11: *Interlacing of open polygonal chains (a).*
Designs that cannot be automatically interlaced by SİMETRİ (b).

Steps of the Interlacing Algorithm

1. Clean and merge (see Figure 13 and the next section for an explanation) the input geometry and define Polyline objects, which can be either open or closed.
2. Offset Polyline objects in both directions to form ParallelPolyline objects.
3. Compute all intersections of the Polyline objects within the ParallelPolyline objects and create Section objects. Section objects are defined by two intersections at their endpoints, resulting in Division objects. If a Division object is an open polygonal chain, its endpoints are treated as intersections, thereby defining Section objects at the beginning and end of the chain.
4. Generate Overlap and Fragment objects by employing a graph representation of the offset Division objects and their sections (see Figure 12, step 5). This graph is structured as an undirected cyclic graph, with intersections serving as nodes and sections functioning as edges. The cycles within this graph define the *Fragments*, *Overlaps*, and *Outline*. Furthermore, the *is_overlap* attributes of the Section objects are assigned during this step.
5. Select any Polyline object to begin assigning *is_over* attributes to its Section objects, ensuring their parity is tracked (by alternating True and False values) while iterating through all sections of the Polyline under process.
6. Choose the next Polyline object that intersects with at least one of the previously processed Polyline objects, and repeat the same iteration process as described in the previous step.
7. Continue the process until all Polyline objects have been handled.
8. Form Plait objects utilizing the cycles of an undirected cyclic graph created from connected Section objects.
9. The final interlaced design is a combination of the computed Fragment and Plait objects.

Merging Line Segments

When separate line segments are combined to form a design, one may have disjoint segments that share common points or collinear overlapping segments that need to be combined. This requires a systematic approach to cover all possible cases. Vertices with floating-point coordinates also require special care while determining congruent points and collinear segments. Unfortunately, creating computational geometry algorithms that can cover all cases is nearly impossible even if arbitrary-precision arithmetic is used. If the dimensions of the segments are too big or too small, then the default tolerances used in SİMETRİ may fail. This merging algorithm is similar to the sweep line algorithm, as explained in the following section, and consists of the following three steps:

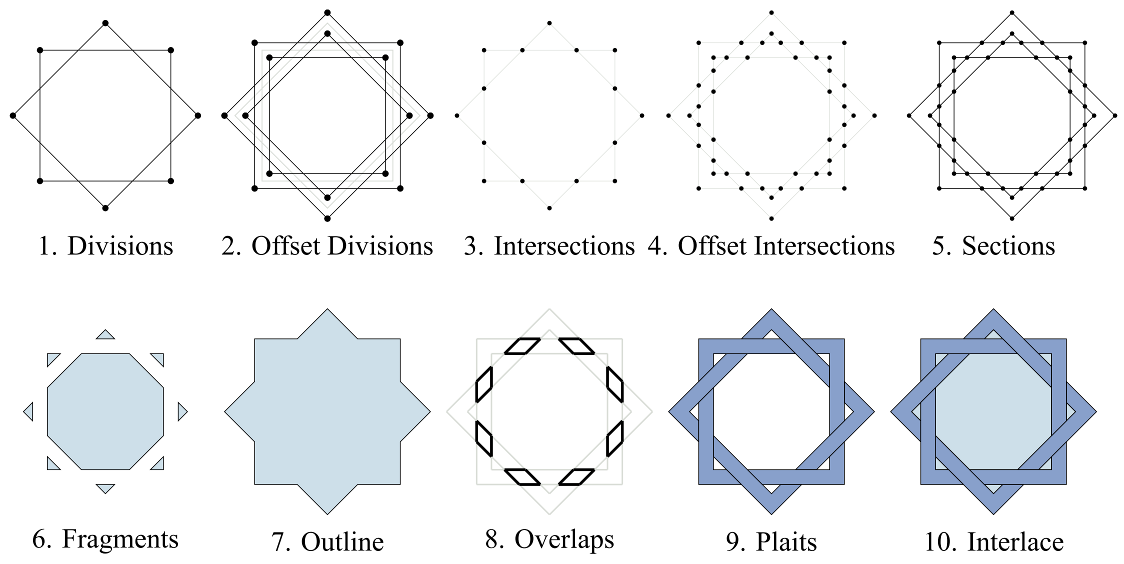


Figure 12: *A pictorial representation of the interlacing algorithm used in SİMETRİ.*

1. Group all segments into subgroups according to their orientation angle.
2. In each group, reduce the overlapping and/or chained segments into a single segment.
3. Chain all remaining segments and the newly merged segments together where possible by using the specified tolerances.

Results of this merging process can be closed or open polygonal chains, or both.

Sweep Line Algorithm (aka Plane Sweep Algorithm)

Once all segments are cleaned and properly connected (see Fig 13 for an example of cleaning and connecting segments) to form polygonal chains, the next step involves determining their intersection points. A naive algorithm that checks each segment against all other segments has a time complexity of $O(n^2)$. In 1976, Shamos and Hoey published an algorithm, called *sweep line*, with a time complexity of $O(n\ log\ n)$. Our implementation is similar to their sweep line algorithm, but varies in the way filtering and sorting of the segments are implemented. Our implementation uses Numeric Python (Numpy) arrays to filter and sort the segments instead of self-balancing binary search trees [11].

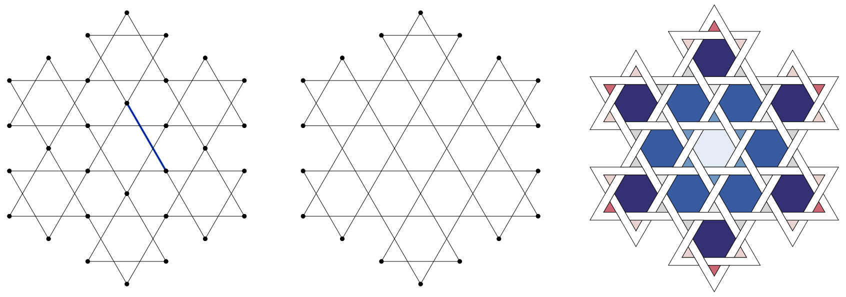


Figure 13: *Initial segments (kernel shown in blue), cleaned and merged segments, and the final design.*

Concluding Remarks

We demonstrated the algorithmic generation of girih stars using the SİMETRİ library. While dihedral symmetry was our primary focus, the SİMETRİ library also includes modules with methods and classes for affine and linear transformations, as well as support for tilings, wallpaper, and frieze groups. All examples in this paper use straight-line segments, but SİMETRİ also supports curves, arcs, sine waves, and ellipses.

Currently, SİMETRİ uses TikZ [13] and XƎLaTeX as its graphics rendering engine. In the near future, we aim to introduce new features to SİMETRİ, enabling the generation of vector graphics without dependence on TikZ and XƎLaTeX. These enhancements will involve the creation of a graphical user interface, along with the integration of tools for generating animations that include audio elements by using external libraries such as FFmpeg and SuperCollider. Geometry offers visual symmetry and proportion, while music embodies rhythm, harmony, and dynamic variation. By bridging these elements, one can produce compositions that resonate with both visual and auditory appeal, weaving mathematical precision and artistic creativity together seamlessly.

We hope that SİMETRİ will be an invaluable resource for mathematicians, educators, and hobbyists alike, regardless of their level of expertise. While familiarity with the Python programming language is advantageous, the library is crafted with an intuitive interface that can be mastered in a short amount of time. Our goal is to make complex mathematical concepts and pattern generation accessible to a wider audience.

Acknowledgements

We are grateful for the insightful feedback provided by Aylin Sertkaya, Cyrus Veeser, Hakan Elmalı , Deniz Başeğmez, and the anonymous reviewers.

References

[1] H. Abelson and A. A. DiSessa. *Turtle geometry*. MIT Press, 1986.

[2] F. Başeğmez. “simetri.graphics.” https://simetri-graphics.github.io/simetri/. 2025. Accessed: Apr. 2025.

[3] J. Bonner and C. S. Kaplan. *Islamic geometric patterns*, 1st ed. New York, NY: Springer, Jul. 2017.

[4] E. Broug. *Islamic Geometric Design*. London, England: Thames & Hudson, Sep. 2013.

[5] B. Grünbaum and G. C. Shephard. “Interlace Patterns in Islamic and Moorish Art.” *Leonardo*. vol. 25. no. 3/4. 1992.

[6] C. S. Kaplan. “Computer Generated Islamic Star Patterns.” *Bridges: Mathematical Connections in Art, Music, and Science*. R. Sarhangi, Ed. Southwestern College, Winfield, Kansas: Bridges Conference, 2000. pp. 105–112. http://archive.bridgesmathart.org/2000/bridges2000-105.html.

[7] A. Lee and A. S. August. *The Geometric Rosette : Analysis of an Islamic Decorative Motif*. 2014. https://api.semanticscholar.org/CorpusID:24485202.

[8] M. Majewski. *Practical Geometric Pattern Design: Geometric Patterns from Islamic Art*. Independently Published, 2020.

[9] G. Necipoğlu. *The arts of ornamental geometry*. ser. Muqarnas, Supplements. G. Necipoğlu, Ed. Leiden, Netherlands: Brill, 9 2017.

[10] G. Necipoğlu and M. Al-Asad. *The Topkapi scroll*. ser. Sketchbooks & Albums S. Santa Monica, CA: Getty Research Institute, Dec. 1995.

[11] F. P. Preparata and M. I. Shamos. *Computational Geometry*. Springer-Verlag New York Inc., 1985.

[12] S. E. Sönmez. *Anadolu Selçuklu sanatının geometrik dili: Anadolu Selçuklu ve beylikler dönemi geometrik desen analizleri*. Ketebe Yayınları, 2020.

[13] T. Tantau. “TikZ & PGF.” https://tikz.dev/. 2024. Accessed: Dec. 2024.

[14] M. M. Vela. *How to Draw the Mosaics of Alhambra*. Spain: Editorial ALMIZADE, 2022.