Mathematically Developing Building Facades

An Algorithmic Framework

Inês Caetano[1][0000-0003-3178-7785] and António Leitão[2][0000-0001-7216-4934]

[1] INESC-ID/Instituto Superior Técnico, University of Lisbon, Lisbon, Portugal
ines.caetano@tecnico.ulisboa.pt
[2] INESC-ID/Instituto Superior Técnico, University of Lisbon, Lisbon, Portugal
antonio.menezes.leitao@tecnico.ulisboa.pt

Abstract. The importance of Algorithmic Design (AD) is growing due to its advantages for the design practice: it empowers the creative process, facilitating design changes and the exploration of larger design spaces in viable time, and supports the search for better-performing solutions that satisfy environmental demands. Still, AD is a complex approach and requires specialized knowledge. To promote its use in architecture, we present a mathematics-based framework to support architects with the algorithmic development of designs by following a continuous workflow embracing the three main design stages: exploration, evaluation, and manufacturing.

The proposed framework targets the design of buildings' facades due to their aesthetical and environmental relevance. In this paper, we explain the framework's structure and its mathematical implementation, and we describe the predefined algorithms, as well as their combination strategies. We focus on the framework's algorithms that generate different geometric patterns, exploring their potentialities to create and modify different facade designs. In the end, we evaluate the flexibility of the framework for generating, modifying, and optimizing different geometrical patterns in an architectural design context.

Keywords: Algorithmic Design; Mathematical Framework; Higher-order Functions; Facade Design.

1 Introduction

Algorithmic Design (AD) is a design approach based on algorithms [1]. Compared to manual approaches, AD provides greater flexibility, supports more complex geometries, and handles larger amounts of information. AD also facilitates design changes and automates repetitive tasks, therefore enabling design optimization by automating the search for better-performing design solutions [2]. Still, AD requires specialized knowledge that most architects do not have, namely, programming experience. To make AD more attractive to the architectural community, we need to make it more accessible by providing ready-to-use algorithms that can be combined in arbitrary ways. We focus on building facades and we present a mathematics-based framework for the design of facades.

Building facades are the outer layer of buildings, separating the indoor spaces from the outside ones and, therefore, having many associated functions, namely environmental performance [3, 4], structural behavior [5], cultural identity [6, 7], and urban communication [8, 9]. Lately, facade design has become an increasing complex task due to the current design trend to create intricate geometries/patterns and the growing design constraints, namely environmental requirements, economic limitations, and tight deadlines. AD helps dealing with these constraints, reducing the time and effort needed to explore different design solutions. To facilitate the adoption of AD in architecture, we propose a flexible mathematics-based framework that tames the complexity of AD techniques in the design exploration, analysis, and optimization of facade solutions. The framework is structured in a fivefold classification inspired by previous research [10–13], containing several algorithms addressing the facade design process.

2 Methodology

The idea of an algorithmic framework had as inspiration previous works [14–16] proving that sets of algorithms can be generalized and reused in the exploration of new designs. These strategies, known as *modular programming* and *design patterns* [14, 15], promise to mitigate the limitations architects still face when using AD, especially to reduce the time and effort spent with the algorithmic task: they avoid writing algorithms from scratch for each new design, while preventing extensive and potentially error-prone programming efforts. However, when the pre-defined algorithms are not well-structured, it becomes difficult to combine them. To solve this problem, we propose a mathematics-based algorithmic framework for facade design that provides, for each scenario, a set of algorithms and combination strategies suiting the different design stages. Despite not considering all possible scenarios, which would be an unviable task, the framework addresses the most common ones and can be adapted to more specific ones.

The framework development was twofold: (1) defining a mathematical theory for facade design and (2) implementing the theory in a framework containing predefined algorithms targeting building facades, that can be combined in arbitrary ways. The resulting framework promises to (1) promote the use of AD in facade design, (2) solve interoperability issues between design and analysis tools, and (3) guide the selection of the algorithms/strategies that best suit a design scenario.

3 Mathematics-based Framework

The architectural practice depends on several external factors like the design brief's specificities and both environmental and economical requirements. This means different projects require different approaches either using an AD or a non-AD approach. As we address the AD one, we must handle design in a way that a computer understands. Given that computational tools are based on instructions transmitted through Programming Languages (PLs) and that most PLs are inspired by the universal language of mathematics, we consider the latter's formalism to:

1. structure the AD theory;
2. implement the different algorithms;
3. organize both 1. and 2. in an algorithmic framework specialized in facade design.

The mathematics-based framework must be able to handle the design (1) *variability*, adapting to the ever-changing design requirements, (2) *diversity*, embracing and evaluating diverse design problems and scenarios, and (3) *coherency*, correctly integrating the design information in a single workflow. Regarding its structure, the framework organizes the predefined algorithms in a fivefold classification, which we describe in the next sections.

4 Framework Implementation

The framework organizes the algorithms based on their type and role in the facade design process in the following categories: *Geometry*, *Pattern*, *Distribution*, *Optimization*, and *Rationalization*. For each one, it provides $\mathbb{R} \to \mathbb{R}$, $\mathbb{R} \to \mathbb{R}^2$, $\mathbb{R}^2 \to \mathbb{R}^2$, and $\mathbb{R}^2 \to \mathbb{R}^3$ algorithms that can be then combined with one another through function composition.

In general, the framework handles all surface-related functions $S(u, v)$ within the domain $0 \leq u \leq 1, 0 \leq v \leq 1$ and provides operators that can be arbitrarily combined, namely the one-dimensional linear variation function $linear(a, b) = \lambda(t).\, a + (b - a)t$ and the (paradoxical) constant "variation" function $constant(c) = \lambda(t \ldots).\, c$. Here, we employ the λ-calculus notation for an anonymous function with parameter t [17]: the result of the function $constant$ is therefore an anonymous function that can be combined with functions of any number of arguments, adapting its number of arguments according to those of the combined function.

The next sections explain the implementation of the framework's categories.

4.1 Geometry

This category contains algorithms to define the facade geometry. As the latter's domain is often two-dimensional, we need to extend the one-dimensional variations' domain into $\mathbb{R}^2$. We use *higher-order functions* (HOFs) [17] to define two functions, $dim_u(f) = \lambda(u, v).\, f(u)$ and $dim_v(f) = \lambda(u, v).\, f(v)$, that make a function f vary only in one dimension, i.e., u or v accordingly. To generalize function composition operations, we provide the operator

$$\circ\,(f, g_1, \cdots, g_n) = \lambda(x_1, \cdots, x_m).\, f\big(g_1(x_1, \cdots, x_m), \cdots, g_n(x_1, \cdots, x_m)\big) \tag{1}$$

and, to simplify the notation used, we define $u \otimes lim = dim_u\big(linear(0, lim)\big)$ and $v \otimes lim = dim_v\big(linear(0, lim)\big)$, wherein lim is the domain's upper limit. We also treat all numbers n in a function context as $constant(n)$ and we represent any first-order function f that receives functional arguments $g_1, \cdots, g_n$ as $\circ\,(f, g_1, \cdots, g_n)$, thus being $f \times g$ the same as $\circ\,(\times, f, g)$.

With HOFs we can move from the numeric space into the functional space and combine functions using functional operators rather than simply combining numbers using numeric operators. In a functional space, the algorithm $straight(w,h) = \lambda(u,v).XYZ(u \times w, 0, v \times h)$, which represents a $w \times h$ planar parametric surface, has the equivalent representation $straight(w,h) = XYZ(u \otimes w, 0, v \otimes h)$. The other predefined surface geometries follow the same logic, which is further detailed in [13].

Having the surface's algorithmic description, we can now explore its geometric pattern by using the algorithms of both *Pattern* and *Distribution* categories, which we explain in the next sections: the former create the shape(s) composing the pattern and the latter distribute those shapes on the surface domain.

4.2 Distribution

This category contains algorithms to distribute elements on a surface. These algorithms receive a matrix of the surface points on which the distribution will be made, as provided by the *Geometry* algorithms, returning another matrix with the same points rearranged in different distribution configurations. As an example, the algorithm $grid_{squares}$ rearranges the points in sets of four points describing a squared area on the surface. Regarding the algorithm $grid_{hexagons}$, it reorganizes the points in sets of six points representing hexagonal areas. Fig. 1 illustrates some of the predefined distributions.

4.3 Pattern

This category contains algorithms to create different geometric patterns. Generally, a geometric pattern results from repeating an element either along one or both dimensions of a two-dimensional domain. In the first case, the result is a pattern continuous in one of the domain's directions but discrete in the other, whereas in the second, the pattern is discrete in both directions. We name the former as *Continuous* and the latter as *Discrete* pattern. In both cases, the repeated element can be kept unchanged along the facade domain or can suffer some transformations regarding its shape, size, etc. Therefore, this category provides algorithms to generate different geometric shapes, as well as to apply different geometric transformations to them, which are organized in two groups: *Shape* and *Transformation*. Each group contains algorithms handling both *Discrete* and *Continuous* patterns but, for the scope of this paper, we focus on the former ones. The latter ones are further detailed in [18].

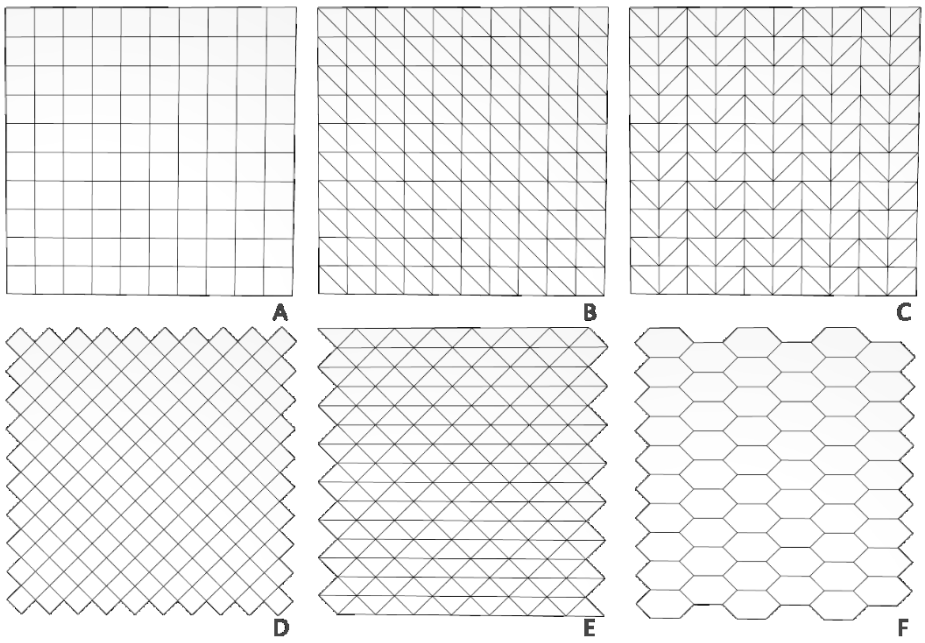

Fig. 1. Some *Distribution* algorithms: A. $\text{grid}_{\text{square}}$; B. $\text{grid}_{\text{triangles}}$; C. $\text{grid}_{\text{triangles2}}$; D. $\text{grid}_{\text{rhombus}}$; E. $\text{grid}_{\text{triangles3}}$; F. $\text{grid}_{\text{hexagons}}$.

Shape. This group contains algorithms to create different 2D and 3D geometric shapes, including polygonal, ellipsoidal, and spherical ones, among others. The available algorithms all receive the set of points where to centre the geometric shape; an information provided by the *Distribution* algorithms. The remaining parameters, in turn, depend on the characteristics of each geometric shape. Fig. 2 illustrates some *Shape* algorithms and their corresponding parameters.

As an example, consider the function that creates star-polygons: besides the set of points (pts), it receives the number of vertices ($\text{n}_{\text{vertices}}$), the inner and outer radii (r_{inner} and r_{outer}), and an angle (α):

$$shapeStar(pts, n_{vertices}, r_{inner}, r_{outer}, \alpha) \tag{2}$$

To create a geometric pattern based on star-polygons, we need to define (1) the surface on which to apply the pattern and (2) the type of elements' distribution. We choose a straight facade and a rhombus distribution, which requires combining three algorithms:

1. one shaping the straight surface where to create the pattern – $straight$;
2. another creating a rhombus grid of points – $grid_{rhombus}$;
3. a last one producing the star-polygons – $shapeStar$.

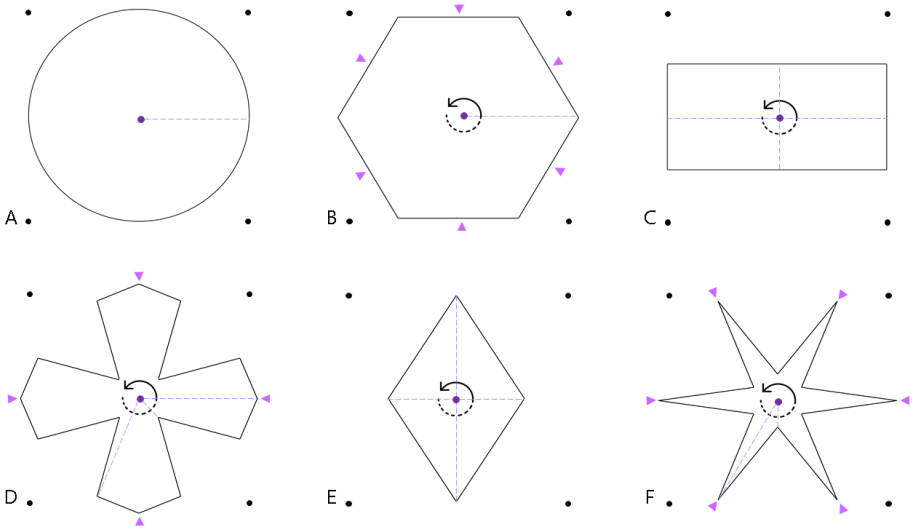

Fig. 2. Some *Shape* algorithms and their inputs: in addition to the set of points represented by the black dots, the circle algorithm (A) receives a radius; the regular-polygon algorithm (B) receives a radius, an angle, and a number of sides; the rectangle algorithm (C) receives a length, a width, and an angle; the rosette algorithm (D) receives three radii, an angle, and a number of vertices; the rhombus algorithm (E) receives two diagonals and an angle; and the star-polygon algorithm (F) receives two radii, an angle, and a number of vertices.

Fig. 3.A-C illustrate the result of this composition using different inputs.

We can simplify the composition of the first two algorithms by writing $grid_{rhombus}\big(straight(w,h)\big)$ and, to facilitate the mathematical representation of algorithms dealing with matrices, we can take advantage of *broadcasting*, i.e., the application of a function f to an array of elements, even if the latter has a different number of dimensions from the other received arguments. *Broadcasting* is represented by the *dot syntax* $f.(args\ ...)$ and it can be applied in single or nested calls $f.\big(g.(args\ ...)\big)$. This means we can simplify

$$shapeStar\begin{pmatrix}\circ\,(grid_{rhombus}, straight)(w,h),\\ constant(n_{vertices}),\\ constant(r_{inner}),\\ constant(r_{outter}),\\ constant(\alpha)\end{pmatrix} \tag{3}$$

into $shapeStar.\big(grid_{rhombus}\big(straight(w,h)\big), n_{vertices}, r_{inner}, r_{outer}, \alpha\big)$. This is valid to all *Shape* algorithms, being some of them illustrated in Fig. 3.

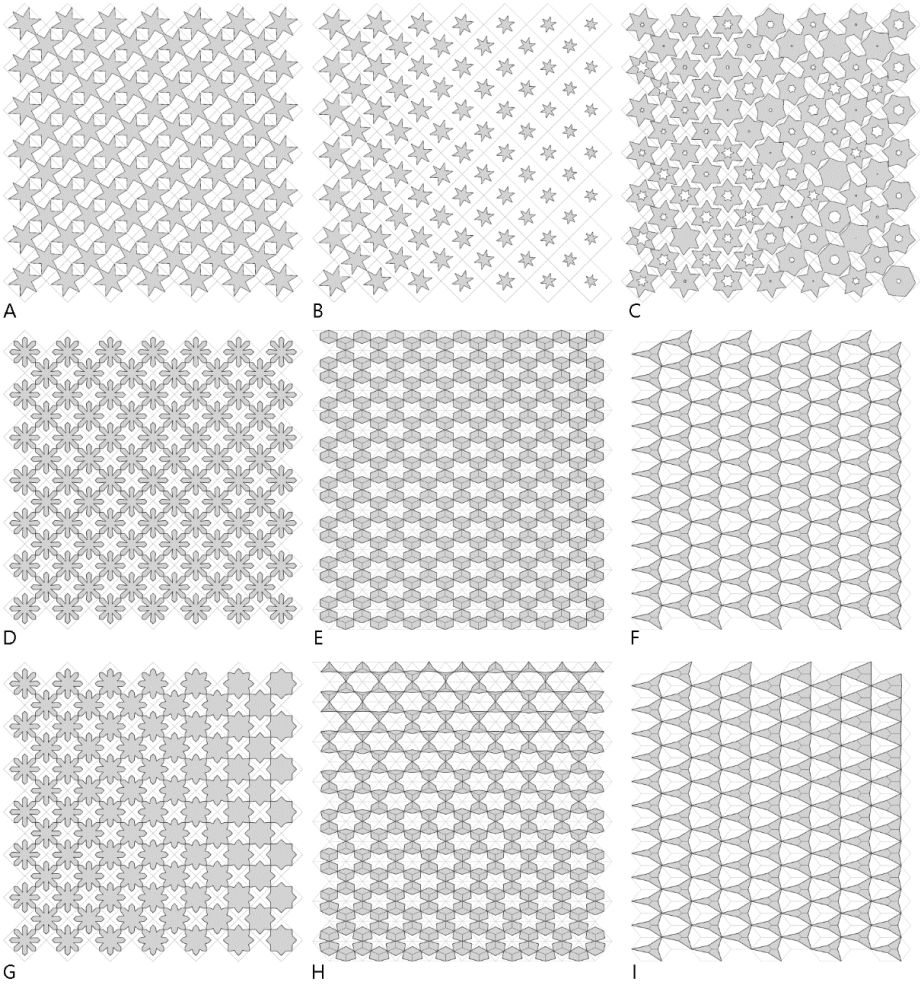

Fig. 3. Examples of *Shape* algorithms combined with different *Geometry* and *Distribution* algorithms.

Transformation. This group provides geometric transformations algorithms, which are organized into *affine* and *rule-based* transformations.

The former includes *scaling* (T_{scale}), to change distances between points according to a constant factor *k*, compressing or stretching shapes if $k < 1$ or $k > 1$, correspondingly; *reflection* (T_{mirror}), to invert shapes with respect to an axis or point; *rotation* (T_{rotate}), to rotate shapes around an axis; *shearing* (T_{shear}), to distort shapes parallel to an axis or plane; and *translation* ($T_{translate}$), to move shapes according to a displacement vector.

The latter contains transformations resulting from the application of rules inspired by real examples (Fig. 4), including *shape substitution* (T_{shape}), to replace shapes

with new ones; *color application* (T_{color}), to apply different colors to the shapes; *protrusion creation* ($T_{protrude}$), to move the shapes perpendicularly to the surface; *recursive subdivision* ($T_{subdivide}$), to subdivide shapes by recursively applying a geometric rule; and *edge deformation* (T_{edge}), to bend or fold the shapes' edges. Combining these algorithms with the *Shape* ones allows generating a wider variety of more dynamic geometric patterns.

Fig. 4. Examples of building facade patterns (Photos © The Authors).

Affine Transformations. These algorithms receive a surface ($ptss$), a specific position on the surface (pt), and a factor controlling the transformation effect intensity (k), being 0 and 1 the null and maximum effects, correspondingly. The remaining arguments depend on the transformation to apply.

As an example, the algorithm T_{scale} allows scaling a shape according to different criteria, including (1) its position, (2) its distance to one or more *attractor* points/curves, (3) being or not contained in certain surface areas, and (4) random rule(s). For each case, T_{scale} receives the information needed to perform the transformation, e.g., the direction of the effect to produce (option 1); a set of *attractor* points or curves (option 2); a set of surface areas (option 3); and random values (option 4). We describe this algorithm as $T_{scale}(ptss, pt, k, args \ldots)$, being $args$ ... the supported optional arguments. This logic applies to all *affine* transformations.

To illustrate the practical application of these algorithms, we start with a pattern resulting from horizontally and vertically aligned squares. We select the algorithms $shapePolygon$, to create the squares, and $grid_{squares}$, to distribute them in a squared grid (Fig. 5.A). To increase the squares' rotation in the u direction, we combine both algorithms with the algorithm T_{rotate}, which, in this case, returns a factor that increases with the surface length. The latter, in turn, affects the $shapePolygon$'s

parameter angle (α), producing squares whose angle increases with the u dimension (Fig. 5.B):

$$shapePolygon.\left(grid_{squares}(ptss), n_{sides}, r_{outer}, \alpha \times T_{rotate}\left(ptss, pt, k, \underset{v_u}{\rightarrow}\right)\right) \quad (4)$$

wherein $\underset{v_u}{\rightarrow}$ is the transformation effect direction.

In this composition, $grid_{squares}$ and T_{rotate} inform the two parameters pts and α of $shapePolygon$, correspondingly: the former provides the position of each polygon and the latter changes its angle according to a factor. In turn, both algorithms are informed by the algorithm $straight$ but with a small difference: while $grid_{squares}$ takes all surface points at once, T_{rotate} receives a surface point at a time, corresponding to the position of the element to rotate. The latter case can therefore benefit from *broadcasting*:

$$T_{rotate}.\left(straight(w,h), grid_{squares}\big(straight(w,h)\big), k, \underset{v_u}{\rightarrow}\right) \quad (5)$$

The resulting composition therefore applies *broadcasting* techniques in nested function calls:

$$shapePolygon.\begin{pmatrix} grid_{squares}\big(straight(w,h)\big), \\ n_{sides}, \\ r_{outer}, \\ \alpha \times T_{rotate}.\left(straight(w,h), grid_{squares}\big(straight(w,h)\big), k, \underset{v_u}{\rightarrow}\right) \end{pmatrix} \quad (6)$$

Following this logic, we can apply multiple transformations to this same pattern by combining more algorithms in sequential function compositions. For instance, to randomly change both the squares' radius size (Fig. 5.C) and center position (Fig. 5.D), we select the algorithms T_{scale}, to control the parameter radius (r_{outer}), and $T_{translate}$, to change the pts position.

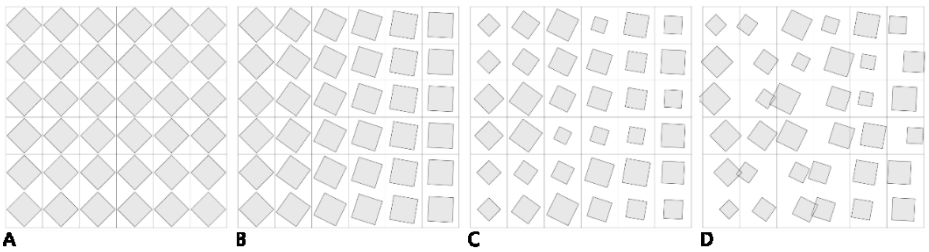

Fig. 5. A pattern of squares distributed in a squared grid (A), whose angle varies horizontally (B) and whose radius size (C) and center position (D) changes randomly.

Fig. 3 illustrates more examples combining T_{scale} with different *Shape* algorithms: in pattern B, T_{scale} makes both $shapeStar$'s parameters r_{inner} and r_{outer} uniformly decrease along the u dimension; in pattern C, T_{scale} controls only the $shapeStar$'s

parameter r_{inner} to randomly increase along the u dimension; and, in pattern G, T_{scale} controls the $shapeRosette$'s parameter r_{inner} to uniformly increase along the u dimension.

Rule-based Transformations. Rule-based transformations are more complex than *affine* ones. Therefore, their mode of combination differs from what we have seen so far. Before explaining their application, we introduce the concept of *matrix of functions* (MF), i.e., a matrix containing different functions. We use MFs to group the transformation algorithm(s) to apply, while defining their pattern of application. We represent a MF as

$$MF = \begin{bmatrix} f1 & f2 \\ g1 & g2 \end{bmatrix}, \text{ where } \begin{vmatrix} f_1 \in \mathcal{L}(U,U), f_2 \in \mathcal{L}(U,V), \\ g_1 \in \mathcal{L}(V,U), g_2 \in \mathcal{L}(V,V). \end{vmatrix} \quad (7)$$

MFs are useful to iterate along two-dimensional domains, which is the case of our surface-related algorithms: *Geometry* and *Distribution* algorithms produce two-dimensional *matrices of points* (MP). To deal with possible size differences between matrices, *rule-based* algorithms map the smallest size matrices along the largest size ones, iteratively applying the former to a submatrix of the latter of the same size. For instance, consider Fig. 6: when mapped along the MP below, the 2×2 matrix A affects 2×2 submatrices of the MP at a time, producing pattern 1; the same happens with the examples B and C. Note that, the framework supports MFs of any size, including non-squared matrices, row matrices, columns matrices, unit matrices, or even matrices covering the entire surface at once. Having this knowledge, we can now focus on these algorithms' application.

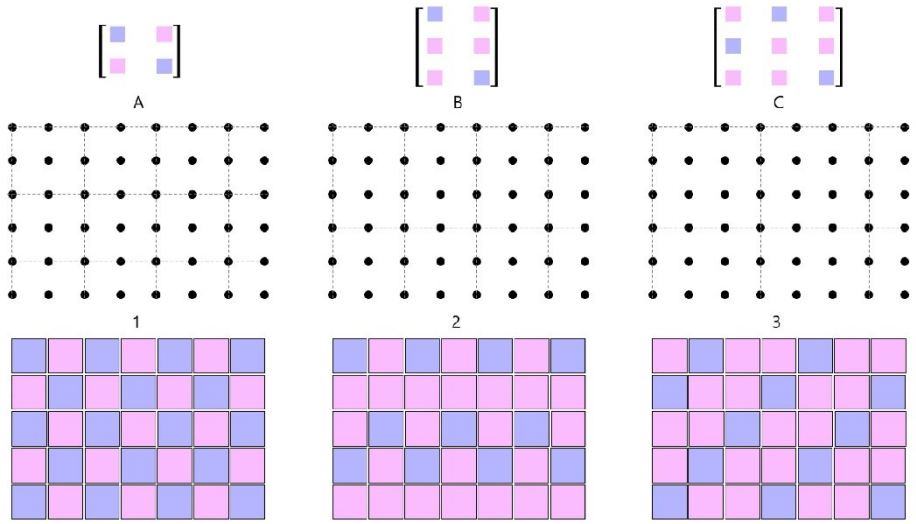

Fig. 6. Examples of *matrix of functions* of different sizes applied to the same *matrix of points*.

Rule-based algorithms all receive tree matrices: one with the surface points ($ptss$), another one with one or more algorithms to apply (M_{rules}), and a last one describing their order of application ($M_{pattern}$):

$$T_{ruleBased}(ptss, M_{rules}, M_{pattern}) \tag{8}$$

Mathematically, these matrices are different, being $ptss$ a MP; M_{rules} a MF; and $M_{pattern}$ an integer matrix. The result is a new matrix merging both M_{rules} and $M_{pattern}$ information, which organizes the algorithms of M_{rules} according to the sequence set by $M_{pattern}$: the latter's integers identify which algorithm of M_{rules} to apply at each position. For instance, consider both matrices $M_{rules} = [f\ g]$ and $M_{pattern} = \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}$. As the latter's integers correspond to the former's algorithms indexes, integer 1 matches the same index algorithm, i.e., f, and integer 2 the algorithm of index 2, i.e., g, being the result $\begin{bmatrix} f & g \\ g & f \end{bmatrix}$. As this new matrix' size often differs from that of $ptss$, it is then resized to be a matrix of the same size as $ptss$, containing the algorithms of M_{rules} arranged according to the pattern set by $M_{pattern}$.

We illustrate their application with some examples combining *rule-based* algorithms with those previously presented. For example, to replace the squares in Fig. 5.D with circles at every four elements (Fig. 7.A), we select the *shape substitution* algorithm T_{shape} and we provide it with three matrices:

1. one containing the surface points: $ptss$.
2. one identifying the shapes composing the pattern: M_{shapes}.
3. a last one describing their mode of application: $M_{pattern}$.

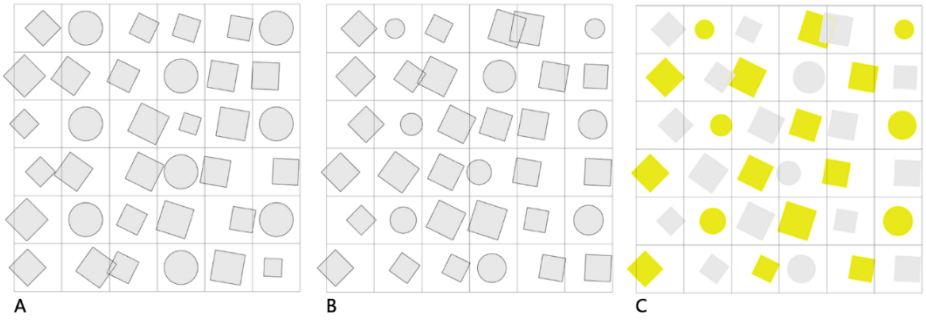

Fig. 7. The pattern in Fig. 5.D combined with *rule-based* algorithms to create a circle for every four squares with (A) a constant size and position or (B) a size and position that varies in the same way as the squares, and to (C) color both squares and circles in a yellow and white chess pattern.

In this case, $ptss$ are the same as in Fig. 5.D, M_{shapes} is a matrix containing both algorithms $shapePolygon$ and $shapeCircle$, and $M_{pattern}$ a matrix describing their order of application:

$$T_{shape}(grid_{squares}(straight(w,h)),[shapePolygon \quad shapeCircle],[1\ 1\ 1\ 2]) \tag{9}$$

As visible in Fig. 7.A, we can apply different transformations to each *Shape* algorithm in M_{shapes}: while the squares are affected by both algorithms T_{scale} and $T_{translate}$, the circles are not. The example in Fig. 7.B, in turn, already applies both transformations to both *Shape* algorithms.

We follow the same logic in the remaining *rule-based* algorithms. As an example, to apply different colors to the pattern in Fig. 7.B, we select the *color application* algorithm T_{color}. We provide it with the same $ptss$ plus two new matrices, one with the colors to apply and another with their pattern of application (Fig. 7.C):

$$T_{color}\left(ptss,[\ \ ],\begin{bmatrix}2 & 1\\ 1 & 2\end{bmatrix}\right) \tag{10}$$

In this composition, T_{shape} and T_{color} originate a new matrix of the same size as $ptss$, merging the information of the following matrices $[shapePolygon \quad shapeCircle]$, [1 1 1 2], $[\ \ ]$, and $\begin{bmatrix}2 & 1\\ 1 & 2\end{bmatrix}$: as the first two produce $[shapePolygon\ shapePolygon\ shapePolygon\ shapeCircle]$ and the last ones return $\begin{bmatrix} & \\ & \end{bmatrix}$, the resulting matrix therefore is

$$\begin{bmatrix}shapePolygon & shapePolygon & shapePolygon & shapeCircle\\ shapePolygon & shapePolygon & shapePolygon & shapeCircle\end{bmatrix} \tag{11}$$

To move the shapes perpendicularly to the surface and create different three-dimensional effects like the one in Fig. 4.C, we use the algorithm $T_{protrude}$. Besides the surface points $ptss$, it receives the protrusion movements to apply, which we represent with vector functions, and their order of application:

$$T_{protrude}(ptss, M_{vectors}, M_{pattern}) \tag{12}$$

Like the previous *rule-based* algorithms, $T_{protrude}$ maps the protrusion movements along the surface's shapes as set in $M_{pattern}$.

As another example, to recursively subdivide a triangular tiling [19] using the geometric rule in Fig. 8.A, we select the algorithm $T_{subdivide}$ and provide it with the subdivision rule to apply ($M_{subdivide}$). As subdivision rules are recursively applied to shapes, we have to prevent them from being endlessly executed by controlling the number of iterations. Therefore, $T_{subdivide}$ receives an additional input, the subdivision rules' level of recursion ($l_{recursion}$):

$$T_{subdivide}(ptss, M_{subdivide}, M_{pattern}, l_{recursion}) \tag{13}$$

As an example, in Fig. 8.B, rule A is applied twice, corresponding to a $l_{recursion} = 2$, whereas in Fig. 8.C and 8.D it is applied three and four times, respectively. Fig. 8.E illustrates rule A applied to a triangular tiling with different levels of recursion. Fig. 8.F and 8.G result from the same rule and a $l_{recursion} = 3$ on different tilings.

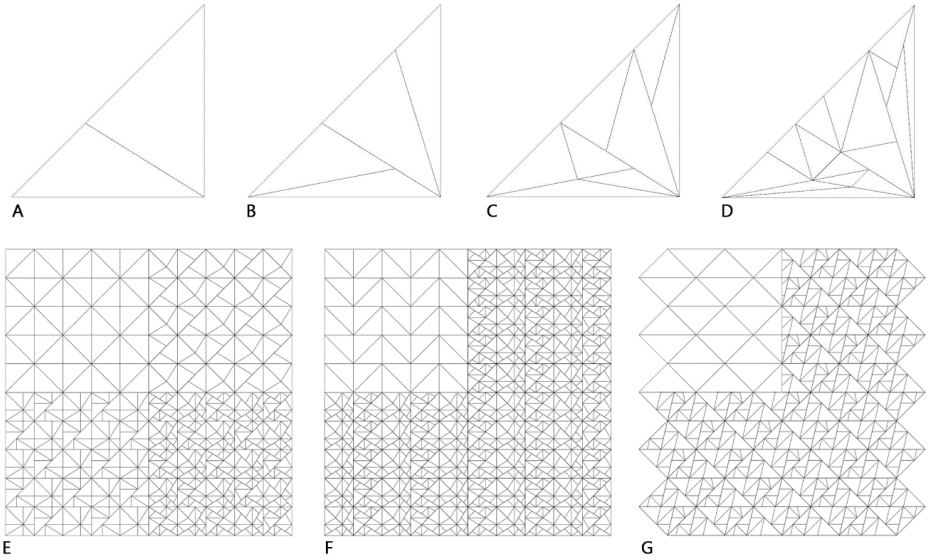

Fig. 8. A. the subdivision rule; B-D: the rule applied twice, thrice, and four times; E-G: Rule A applied in three triangular tilings (the latter are illustrated in the top-left of each example).

We can also add some randomness to the subdivision rules, as illustrated in Fig. 9 (I.C and II.C). This is triggered by the additional input $random$, which receives a *Boolean* value: when true, it adds some randomness to the subdivision rule's level of recursion.

Finally, to deform the edges of a square tiling (Fig. 10.A), we select the algorithm T_{edge}. Mathematically, we represent the edge deformation movement with a single vector function v, being its amplitude the vector's length and its direction the vector's sign (Fig. 10.1). We can apply multiple deformation movements to the same edge and obtain deformations with different amplitude and directions: Fig. 10.2 illustrates the result of two deformation movements of opposite directions. To deal with multiple deformations, T_{edge} needs to be informed about the edge's subdomain to which each deformation movement will be applied, which we represent with different t factors (Fig. 10.2). We can also control the maximum curvature position of each deformation movement with different k factors: in Fig. 10.3, the left deformation has a $k = 0.5$ (the middle position), whereas the right one has a $k = 0.3$.

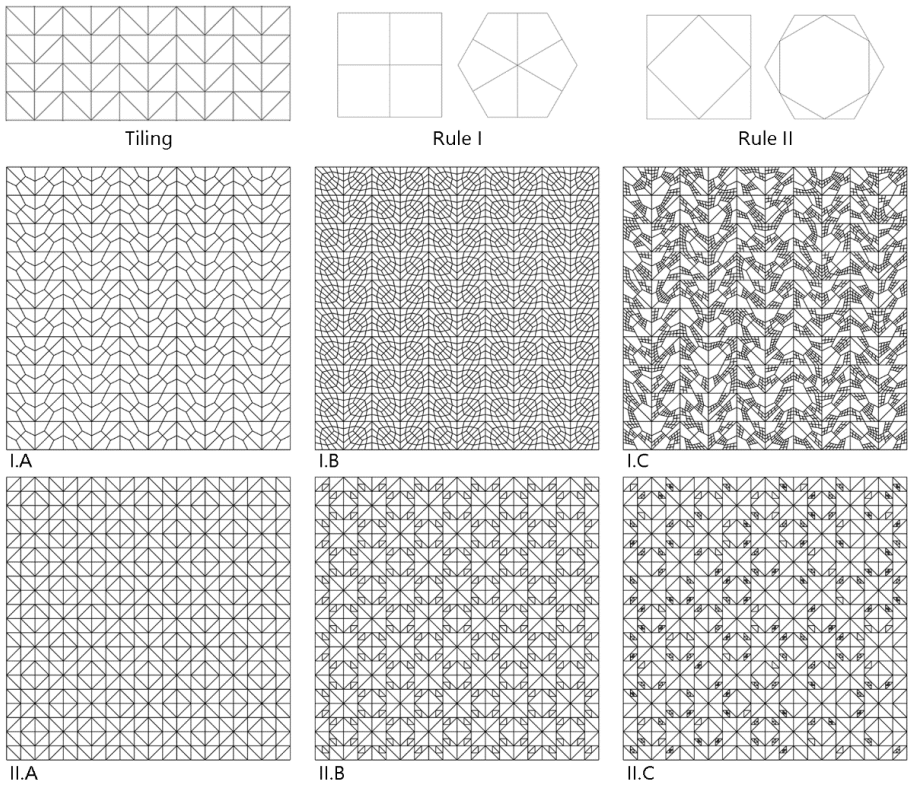

Fig. 9. Top: the triangular tiling and the applied subdivision rules Rule I and Rule II; Middle: Rule I applied with a level of recursion of one (I.A), two (I.B), and three with randomness (I.C); Bottom: Rule II applied with a level of recursion of one (II.A), two (II.B), and three with randomness (II.C).

Considering this, T_{edge} receives two additional inputs: a matrix containing the t factors and another one with the k factors. In sum, T_{edge} receives (1) the surface points ($ptss$), (2) the vector functions organized in sets $\{v_1, \dots, v_n\}$ ($M_{vectors}$), (3) the latter's order of application ($M_{pattern}$), (4) the t factors $\{t_1, \dots, t_n\}$ ($M_{subdomain}$), being $t = 0$ the edge's starting point and $t = 1$ its ending point, and (5) the k factors $\{k_1, \dots, k_n\}$ ($M_{curvature}$):

$$T_{edge}\left(ptss, M_{vectors}, M_{pattern}, M_{subdomain}, M_{curvature}\right) \tag{14}$$

In practice, $M_{pattern}$ dictates where each set of vector functions is used over the $ptss$ and, in each position, the applied vector functions receive the respective factors in both matrices $M_{subdomain}$ and $M_{curvature}$.

Fig. 10 illustrates some edge deformations examples, being B, for instance, the result of applying T_{edge} to A with the following inputs:

- $M_{vectors} = [\{v_1, v_1, v_2, v_2\}]$, wherein v_1 and v_2 have the same amplitude but opposite directions;
- $M_{pattern} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$, meaning the set $\{v_1, v_1, v_2, v_2\}$ is applied to all squares;
- $M_{subdomain} = [\{0, 1\}]$, which indicates the deformation occurs in the entire edge;
- $M_{curvature} = [\{0.5\}]$, informing the maximum curvature matches the edge's middle.

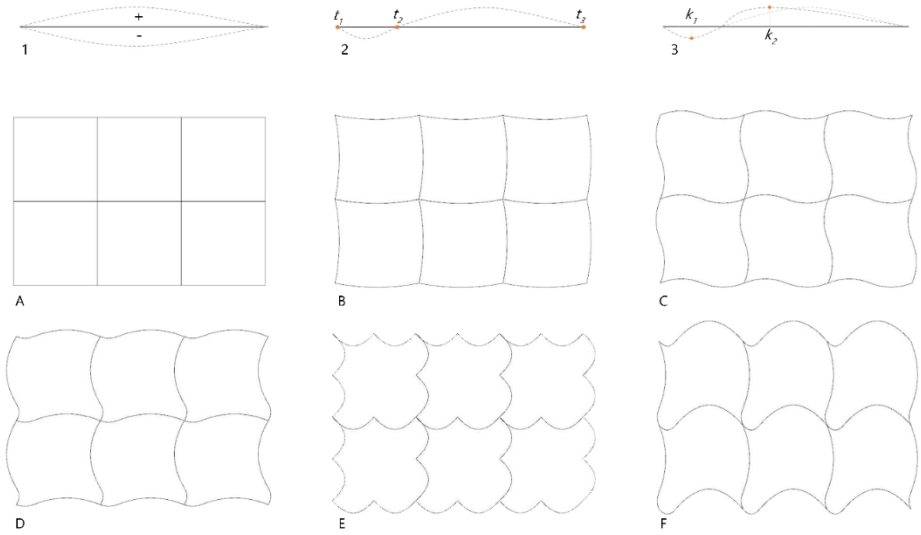

Fig. 10. Top: 1. edge deformation movement sign and amplitude; 2. edge subdomain factors; 3. deformation maximum curvature factors; Bottom: A. a square tilling; B-F the same square tilling after applying the algorithm T_{edge} with different input values.

4.4 Optimization and Rationalization

These categories contain algorithms to analyse, optimize, and rationalize designs, by either driving the changes made to the geometric pattern or controlling the number of different shapes composing the pattern.

For the former case, we can select algorithms like $opt_{daylight}$, to optimize natural daylight illumination, or $opt_{ventilate}$, to improve the natural ventilation, among others. Generally, *optimization* algorithms receive (1) the surface domain $ptss$ and (2) the pattern to optimize $shapes$. The other arguments, in turn, depend on the type of optimization to perform. As an example, $opt_{daylight}$ receives information about the design's context, including building's location and orientation, average hours of sunlight, among others. In turn, $opt_{ventilate}$ receives information that is specific to the type of analysis to execute.

Based on the received information, opt algorithms analyse the design and change it according to the results obtained. Then, they rerun the analysis, repeating the cycle

until reaching a solution with the desired performance requirements. In practice, opt algorithms return one or more values to be then used as input in the final pattern, which may result in shapes of different sizes (if affecting size-related parameters), positions (if altering translation-related values), deformations (if changing deformation-related inputs), or even geometries (if modifying shape-related functions). The result is a new partially improved design. The latter is again used as input in opt algorithms, which in turn return new improved values for the geometric pattern.

As an example, consider a pattern based on rhombus shaped openings whose size needs to be optimized to match certain performance requirements: the more intense the red tone is, the smaller the aperture size should be (Fig. 11). We select the rhombus-shape algorithm

$$shapeRhombus\left(pts, u_{diagonal}, v_{diagonal}, \alpha\right) \tag{15}$$

wherein $u_{diagonal}$ and $v_{diagonal}$ are its horizontal and vertical diagonals and α its rotation. To optimize their size, we select an opt algorithm, which we generically represent as $opt(ptss, shapes, args \dots)$, being $args \dots$ the information related to the performed optimization. We combine both algorithms so that the latter controls the size-related parameters of the former, which, in this case, are $u_{diagonal}$ and $v_{diagonal}$. In case the opt algorithm optimizes the $u_{diagonal}$, the pattern converges towards example A:

$$shapeRhombus\left(pts, opt(ptss, shapes, args \dots) \times u_{diagonal}, v_{diagonal}, \alpha\right) \tag{16}$$

Otherwise, if the opt algorithm improves the $v_{diagonal}$, the pattern converges towards example B. Finally, if both parameters are optimized, the pattern becomes similar to pattern C:

$$shapeRhombus\begin{pmatrix} pts, \\ opt(ptss, shapes, args \dots) \times u_{diagonal}, \\ opt(ptss, shapes, args \dots) \times v_{diagonal}, \\ \alpha \end{pmatrix} \tag{17}$$

To control the number of different shapes composing the pattern, we select the algorithm $discretize$, which is important to minimize the design manufacturing costs, while increasing its construction viability. When applied, $discretize$ iteratively reduces the pattern's diversity of shapes, while making the balance between its design intent and performance. Finally, to identify and count the different existing shapes, we use the algorithm $tallying$, which returns each shape typology locations and quantities; an information that is critical to proceed to the ensuing manufacturing stage.

Mathematically, $discretize$ follows a discretization process that aims at reducing the number of values accepted by a continuous variable, converting a continuous set of values into a finite range grouped into n intervals. This allows us to set the maximum number of intervals of the discretized range, which, in turn, corresponds to the maximum number of different elements in the design ($n_{typologies}$). Since the existence of different elements in a pattern is directly related to the number of *Shape* algorithms used and to the values given to their parameters, it is then important that

$discretize$ constrains these two ranges. In practice, $discretize$ collects the values that differ from element to element, producing a new range of values varying between the maximum and minimum values found and containing the same number of values as $n_{typologies}$. Then, it replaces the parameters' original values with their closest match in the new interval, guaranteeing the final design contains $n_{typologies}$ different shapes.

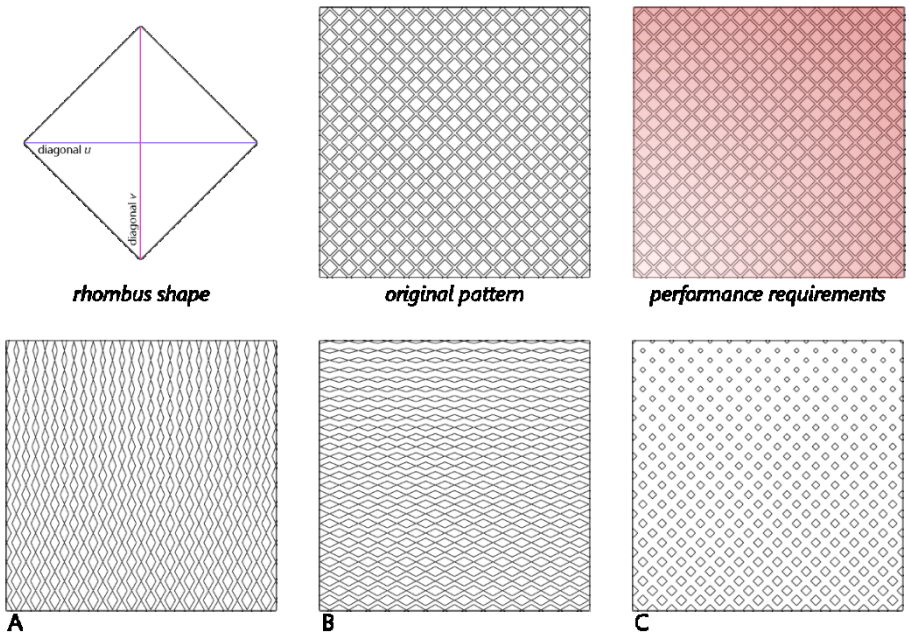

Fig. 11. Top: the rhombus shape's diagonal parameters (left); the rhombus-based pattern to optimize (middle); a conceptual representation of the optimization requirements (right); Bottom: the original pattern with their rhombuses' size optimized regarding their (A) diagonal *u*,(B) diagonal *v*, and (C) both diagonals.

Regarding the algorithm $tallying$, it extracts/stores the design information that is important to manage its construction process on site, including the existing element typologies' quantities and positions. Mathematically, $tallying$ receives a pattern, returning a list with its shapes organized by type, with their quantities, geometric information, and spatial locations. It also enables the visualization of this information through the 3D model in the design tool, applying different colours to each shape typology. Further details on this algorithmic category can be found in [20].

5 Evaluation

To evaluate the framework's application in an AD context, we selected the AD tool *Khepri* [21], a textual programming tool tailored for architectural design, based on the

Julia PL [22]. Although we evaluate the framework on a particular tool using a specific PL, it can be applied in any other AD tool using its own PL because the proposed theory is described in mathematical terms.

As the framework follows an AD approach, it therefore benefits from the latter's advantages. First, it allows the transition from a manual design process to a fully automated one. Moreover, the resulting process is both flexible and parametric, facilitating the integration of design changes, supporting the generation of more complex designs, enabling the exploration of wider design spaces, and allowing for the automation of the *generation-analysis-regeneration* cycle typical of optimization processes. Contrastingly, manually using design tools requires considerable time and effort to change the designs, which not only limits the complexity of the obtained solutions, but also narrows the design space explored. Finally, the proposed approach is also highly configurable due to allowing the available algorithms to receive functions as input.

Moreover, the framework embraces all designs stages in a single and continuous workflow: the resulting function compositions integrate algorithms addressing different stages, meaning that modifications made to one algorithm automatically propagate to the other algorithms. Also, its implementation in the AD tool *Khepri* allowed the framework to benefit from some of its capabilities, namely the algorithms portability among different CAD, BIM, and analysis tools. In practice, the same algorithmic description generates identical models in the different supported tools by adapting the embedded information according to the tool: while in CAD tools it produces a simple geometric model, in BIM tools it enriches the model with building semantics and, in analysis tools, it generates a simplified version of the model with only the information needed for each specific analysis.

There are already some tools that provide functionalities similar to those available in the framework. These include *ParaCloud Gem* [23], a 3D pattern modeler, and some plug-ins for *Grasshopper* and *Dynamo*, including:

1. *PanelingTools*, that contains surface paneling functionalities supporting grid manipulation and the morphing of patterns, and rationalization techniques for analysis and fabrication [24].
2. *LunchBox*, that provides functionalities to explore mathematical shapes, surface paneling, and wire structures [25].
3. *Weaverbird*, that includes mesh subdivision and transformation operators and functionalities to help the preparation of meshes for fabrication [26].
4. *Parakeet*, that has functionalities to generate algorithmic patterns resulting from tilings, geometric shapes and grids subdivisions, edge deformations, among others [27].
5. *SkinDesigner*, that provides functionalities to generate facade geometries from building massing surfaces repeating panels [28].

Despite addressing the same problems, these tools are limited by the available predefined operators [29], not allowing the latter's configuration so as to respond to more specific design contexts or intents. Also, they require the user to directly interact with the design tool, hindering the automation of the design process. This results in itera-

tive design exploration processes that are tiresome and error prone. Finally, most tools resort to visual PLs, e.g., *Grasshopper* and *Dynamo*, that are hard to apply to complex problems [30, 31]. Our framework, in turn, overcomes these shortcomings by (1) structuring and providing algorithmic strategies that adapt to diverse design scenarios and requirements and to different design tools and workflows; (2) promoting an entirely algorithmic use of the functionalities available, despite also allowing user interaction with the design tool, e.g., to take advantage of visual inputs [21]; and (3) being designed for textual PLs implementations and, thus, benefiting from their expressive power [30, 32, 33]. Nevertheless, in its current state, the framework requires more programming experience than the previous tools and is less visually attractive and intuitive. Future work will focus on improving these shortcomings.

6 Conclusions

Algorithmic Design (AD) is a powerful design approach that supports the exploration of more complex designs and wider design spaces, as well as the search for better-performing solutions. Unfortunately, architects still face several limitations when adopting AD techniques: (1) the transition from a purely visual to a more complex and abstract design process and (2) the need for specialized knowledge, including programming experience. The challenge, then, is to make AD more accessible and attractive to the architectural community.

Based on previous studies proving that the provision of predefined algorithms and strategies facilitates the development of algorithmic solutions, we proposed a mathematics-based framework designed to tame the complexity of AD techniques. As domain of application, we focused on building facades due to their major role in building design, presenting a mathematical theory for them. In the paper, we described the proposed mathematics-based theory and its implementation in an algorithmic framework. We explained its structure and available algorithms and we illustrated its application through a sequence of conceptual examples. In the end, we demonstrated the ability of the framework to be generalized and to be applied in different design contexts and workflows.

7 Acknowledgements

This work was supported by national funds through *Fundação para a Ciência e a Tecnologia* (FCT) with references UIDB/50021/2020 and PTDC/ART-DAQ/31061/2017, and by the PhD grant under contract of FCT with reference SFRH/BD/128628/2017.

References

1. Caetano, I., Santos, L., Leitão, A.: Computational Design in Architecture: Defining Parametric, Generative, and Algorithmic Design. Frontiers of Architectural Research (2020).

2. Aguiar, R., Cardoso, C., Leitão, A.: Algorithmic design and analysis fusing disciplines. In: Disciplines and Disruption - Proceedings of the 37th Annual Conference of the Association for Computer Aided Design in Architecture, pp. 28–37. Cambridge, Massachusetts, USA (2017).
3. Trubiano, F.: Performance Based Envelopes: A Theory of Spatialized Skins and the Emergence of the Integrated Design Professional. Buildings 3, 689–712 (2013).
4. El Sheikh, M. M.: Intelligent building skins: Parametric-based algorithm for kinetic facades design and daylighting performance integration. PhD thesis. University of Southern California. Los Angeles, USA (2011).
5. Al-Kodmany, K., Ali, M. M.: An Overview of Structural and Aesthetic Developments in Tall Buildings Using Exterior Bracing and Diagrid Systems. International Journal of High-Rise Buildings 5(4), 271–291 (2016).
6. Schulz, C. N.: Existence, space & architecture. Praeger. Stamps, New York (1971).
7. Schittich, C.: Building Skins. Birkhäuser (2006).
8. Stojšić, M.: (New) Media Facades: Architecture and/as a Medium in Urban Context. AM Journal 12, 135–148 (2017).
9. Venturi, R., Brown, D. S., Izenour, S.: Learning from Las Vegas. MIT Press (1972).
10. Moussavi, F., Kubo, M.: The Function of Ornament. Actar (2006).
11. Otani, M., Kishimoto, T.: Fluctuating Patterns of Architecture Façade and their Automatic Creation. In: CAADRIA 2008 - Proceedings of the 13th International Conference on Computer Aided Architectural Design Research in Asia, pp. 375–382. Chiang mai, Thailand (2008).
12. Pell, B.: The Articulate Surface: Ornament and Technology in Contemporary Architecture. Birkhäuser GmbH, Germany (2010).
13. Caetano, I., Santos, L., Leitão, A.: From Idea to Shape, From Algorithm to Design: A Framework for the Generation of Contemporary Facades. In: Celani, G., Sperling, D., Franco, J. (eds.) Computer-Aided Architectural Design: The Next City – New Technologies and the Future of the Built Environment 16th International Conference, CAAD Futures 2015, Selected Papers, pp. 527-546. Springer-Verlag Berlin Heidelberg (2015).
14. Woodbury, R., Aish, R., Kilian, A.: Some Patterns for Parametric Modeling. In: Expanding Bodies: Art • Cities• Environment - Proceedings of the 27th Annual Conference of the Association for Computer Aided Design in Architecture, pp. 222–229. Halifax, Nova Scotia (2007).
15. Qian, Z. C.: Design Patterns: Augmenting Design Practice in Parametric CAD Systems. PhD thesis. School of Interactive Arts and Technology: Simon Fraser University. Burnaby, Canada (2009).
16. Chien, S., Su, H., Huang, Y.: PARADE: A pattern-based knowledge repository for parametric designs. In: Emerging Experience in Past, Present and Future of Digital Architecture - Proceedings of the 20th International Conference of the Association for Computer-Aided Architectural Design Research in Asia (CAADRIA), pp. 375-384. Daegu, Korea (2015).
17. Leitão, A.: Improving generative design by combining abstract geometry and higher-order programming. In: Rethinking Comprehensive Design: Speculative Counterculture - Proceedings of the 19th International Conference on Computer- Aided Architectural Design Research in Asia (CAADRIA), pp. 575–584. Kyoto, Japan (2014).
18. Caetano, I., Leitão, A.: Weaving Architectural Façades: Exploring algorithmic stripe-based design patterns. In: Hello, Culture – Proceeding of the 18th International Conference on Computer Aided Architectural Design Futures, pp. 1023–1043. Daejeon, South Korea (2019).

19. Grünbaum G., Shephard, G. C.: Tilings and patterns. New York: W.H. Freeman, New York, USA (1987).
20. Caetano, I., Leitão, A.: Algorithmic Patterns for Facade Design: Merging design exploration, optimization and rationalization. In: Facade Tectonics 2018 World Congress Conference Proceedings, vol. 1, pp. 413–422. Los Angeles, USA (2018).
21. Sammer, M., Leitão, A., Caetano, I.: From Visual Input to Visual Output in Textual Programming. In: Intelligent & Informed - Proceedings of the 24th International Conference of the Association for Computer-Aided Architectural Design Research in Asia, vol. 1, pp. 645–654. Wellington, New Zeland (2019).
22. The Julia Language (by J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah), https://julialang.org/, 2012, last accessed 2020/03/28.
23. ParaCloud GEM, https://paracloud-gem.software.informer.com/, 2011, last accessed 2020/03/29.
24. PanelingTools for Rhino and Grasshopper (by Rajaa Issa), https://www.food4rhino.com/app/panelingtools-rhino-and-grasshopper, 2013, last accessed 2020/03/27.
25. LUNCHBOX (by Nathan Miller), https://www.food4rhino.com/app/lunchbox, 2011, last accessed 2020/03/21.
26. Weaverbird – Topological Mesh Editor, http://www.giuliopiacentino.com/weaverbird/, 2009, last accessed 2020/03/19.
27. PARAKEET (by Esmaeil), https://www.food4rhino.com/app/parakeet, 2019, last accessed 2020/03/23.
28. SKINDESIGNER (by sgaray), https://www.food4rhino.com/app/skindesigner, 2017, last accessed 2020/03/21.
29. Zboinska, M. A.: Hybrid CAD/E platform supporting exploratory architectural design. CAD Computer Aided Design 59, 64–84 (2015).
30. Leitão, A., Santos, L., Lopes, J.: Programming Languages For Generative Design: A Comparative Study. International Journal of Architectural Computing 10(1), 139–162 (2012).
31. Janssen, P.: Visual Dataflow Modelling: Some Thoughts on Complexity. In: Fusion - Proceedings of the 32nd eCAADe Conference, vol. 2, pp. 305–314. Department of Architecture and Built Environment, Faculty of Engineering and Environment, Newcastle upon Tyne (2014).
32. Wortmann, T., Tunçer, B.: Differentiating parametric design: Digital workflows in contemporary architecture and construction. Design Studies 53, 173–197 (2017).
33. Celani, G., Vaz, C.: CAD Scripting and Visual Programming Languages for Implementing Computational Design Concepts: A Comparison from a Pedagogical Point of View. International Journal of Architectural Computing 10(1), 121–138 (2012).