Declarative Visual Programming with Invariant, Pre- and Post-conditions for Lattice Approximation of 3D Models

Oscar Ruiz-Salguero[1](✉), Carolina Builes-Roldan[1], Juan Lalinde-Pulido[2], and Carlos Echeverri-Cartagena[3]

[1] CAD CAM CAE Laboratory, U. EAFIT, Medellín, Colombia
`{oruiz,cbuilesr}@eafit.edu.co`

[2] High Performance Computing Facility APOLO, U. EAFIT, Medellín, Colombia
`jlalinde@eafit.edu.co`

[3] Machine Tool Laboratory, U. EAFIT, Medellín, Colombia
`cechever@eafit.edu.co`
`http://www1.eafit.edu.co/cadcamcae`

Abstract. In the context of Visual Programing for Product Design, the endowment of the Designer with programing tools to boost productivity is central. However, Product (and Architectural) Design are usually taught without programing courses. This manuscript reports the results of Lattice DesignVisual Programming by a Product Designer with no previous exposure to programing but provided with the intuitive concepts of Pre-, Post-condition and Invariant logical first-order predicates for imperative programing. The scenario of application is the population of 3D domains (i.e. solid models) with lattice individuals of the type zero-curvature Truss (colloquially called 1.5D and 2.5D) structural elements. Result show that, although Pre-, Post-condition and Invariant are devised for imperative programing, they provide a solid and successful structure for visual programming (e.g. Grasshopper) for Designers with no mathematical or programming background. Regarding the specific Additive Manufacturing scope, the manuscript depicts the population of the target domain with lattice individuals which, in this case, undergo a rigid transformation before docked in the target domain. The lattice design presented allows for the grading of the lattice geometry. Future work addresses the programing of non-rigid transformations (non-affine, non-conformal, etc.) which dock the lattice individual into the target solid domain. Regarding the endowment of non-programmer Product Designer with visual programing and pre-, post- and invariant conditions, the performance results are very positive. However, as with any work team, experts must be recruited to help with highly specialized topics (e.g. computational mechanics, differential geometry, discrete mathematics, etc.).

Keywords: visual programming · predicate-based programming · lattice families · truss · frame

Supported by U. EAFIT and Carolina Builes-Roldan.

Glossary

2-manifold	a surface point set locally isomorphic to a flat disk (i.e. without self-intersections)
1-manifold	a curve point set locally isomorphic to a straight wire (i.e. without self-intersections)
BODY Ω	a solid, or subset of $\mathbb{R}^3$, which is compact (bounded and containing its boundary)
$\partial\Omega$	Boundary Representation (B-Rep or skin $\partial\Omega$) of solid Ω
M	the (2-manifold) triangular B-Rep of a solid Ω. Also called a closed triangular mesh
$S(u,w)$	a 2-manifold parametric surface in $\mathbb{R}^3$. That is, $S:[0,1]\times[0,1]\rightarrow\mathbb{R}^3$
$C(u)$	a 1-manifold parametric curve in $\mathbb{R}^3$. That is, $C:[0,1]\rightarrow\mathbb{R}^3$
FACE F	a connected subset of a parametric Surface (FACE $F\subset S$)
EDGE E	a connected subset of a parametric Curve (EDGE $E\subset C$)
VERTEX v	an element of the border of an EDGE, or an endpoint of an EDGE E: $\partial E=\{v_0,v_f\}$
[FACE + thickness]	Finite Element Analysis approximation (for the sake of computing savings) of a 3D thin solid plate as a 2D FACE added with thickness information, colloquially called *2.5D element*
[EDGE + area]	Finite Element Analysis approximation (for the sake of computing savings) of a 3D slender solid rod as a 1D EDGE added with area information, colloquially called *1.5D element*
VoXel	Volumetric piXel or 3D rectangular prism whose dimensions correspond to the resolution of a 3D Scanner (e.g. computer tomograph, magnetic resonator, ultrasound probe, etc.)
Lattice	in Additive Manufacturing, the smallest topological structure that is repeated (possibly with geometrical gradients) to fill a solid domain Ω. Scale-wise, size(VoXel) $\ll$ size(Lattice)
I, O, NIO	qualifiers of a lattice with respect to a solid domain Ω (inside, outside, neither inside nor outside)
pre-condition	1-st order logic predicate describing the status of program execution *before* an instruction sequence is executed
post-condition	1-st order logic predicate describing the status of program execution *after* an instruction sequence is executed
invariant	1-st order logic predicate describing the status of program execution *before any and each iteration* in a (`for` or `while`) loop is executed
CBP	Contract-based Programming, in which formal logic predicates are used to specify the input, output, and intermediate checkpoints of a software piece, with machine-driven checks assessing the compliance of the code with the specified ontracts
EUP	End User Programming
VPL	Visual Programming Language

1 Introduction

Research Target. This manuscript presents a development executed on Visual programming tools (i.e. Grasshopper^{TM}) which (a) defines lattice individuals based on the limbs of types 1.5D [EDGE + area] and 2.5D [FACE + thickness] due to their lower computing consumption, (b) executes the approximation of a BODY (i.e. 3D region) by an enumeration of the given lattice individuals, tuned by using the usual thresholds in Exhaustive Enumerations. This development is executed by a Designer who does not have previous training in programming but is equipped with intuitive knowledge of the Pre-condition, Invariant and post-condition formalisms for programming [7].

Although it is not within our target, it is worth to remark the potential of Declarative, Dataflow or Flow-based Programming environments (e.g. Grasshopper^{TM}) in parallelization, fault-tollerance and scalability. Ref. [15] is a soft introduction to these topics.

Programming for Non-programmers. Visual Programming is built as a tool for practitioners and experts in topics other than from the programming aspect. This tool which would allow them to build/assemble applications based on icons of pre-defined pre-compiled functions. Visual programming and in particular Grasshopper^{TM} is a good starting tool for non-programmers to create scenario-driven, generative designs. However, it presents limitations in more complex scenarios. One of these scenarios is *parametric* design, which Grasshopper underlying kernel (Rhinoceros^{TM}) does not support.

Computer-Aided Design and Manufacturing. Lattice-based objects present advantages in lower material spending and weight and their realization by Additive Manufacturing (AM) offers internal cavities which traditional Removal Manufacturing does not offer. On the other hand, usual 3D solid modeling Boundary Representation (B-Rep) of lattices represents explosive amounts of data, and the same is valid for 3D Finite Element Analyses (FEA). However, if a lattice design has slender limbs (e.g. plates or rods), the more economical representation by Trusses or Frames is possible. In this case, Kinematic or Position constraints are enforced among the pieces. As a result, the number of topological, geometrical and finite elements is lowered.

Geometry Scope. The present work addresses lattice limbs whose medial axes or skeletons have zero curvature. The plates have constant thickness. The rods, however, may have changing cross section. This manuscript is concerned neither with supporting structures *during* the additive manufacture nor with the computational mechanics of the lattices.

Section 1 reviews the state of art of Topologic and Geometric Modeling for lattices. Due to the special subject of this manuscript (Visual programming for Lattice Modeling and Object approximation), Sect. 3 discusses the Methodology used and simultaneously presents the results of the modeling. Section 4 concludes the manuscript and indicates aspects for future work.

2 Literature Review

2.1 Lattice Families and Properties

Refs. [2] and [14] evaluate strategies (solid, intersected, graded, scaled, uniform) to map Solid Isotropic Material Penalisation (SIMP) material densities for a cantilever load case into lattice structures. 3D Solid Finite Element simulations are carried out with a wall (i.e. Schwarz) lattice model and not with truss or frame structures. AM *degradee* patterns are employed according to the support structures, surface areas, processing times, and other criteria. The softwares used are Magics (Materialise Magics. Materialise N.V., Leuven, Belgium, 2014.), Autofab (Autofab Software, Marcam Engineering, 2011), MSC Nastran, Grasshopper - Rhino for complex designs of lattices (no library names specified). No test is given to measure the discrepancy, resiliency, processing effort, and in-process support requirements.

Ref. [2] generates a MATLAB VoXel Representation of a lattice set that approximates the interior of a 3D body. The lattices have their own internal design. It presents the union of the body interior lattice structure with a thick version of the 3D body skin (i.e. B-Rep), discussing the grading of the lattice individuals. This manuscript produces a double discretization of the 3D Body lattice representation by VoXels: (i) a VoxXel set builds a lattice and (ii) a lattice set builds a 3D Body. No Finite Element computation is presented in this manuscript.

Ref. [8] proposes a Programed Periodic Lattice Editor (PLE), based on a set of parameters and lattice topologies, to fill a given 3D solid region $\Omega \subset \mathbb{R}^3$. This user-driven SW addresses honeycomb, octahedral, octet, and other lattice individual types. A given lattice is modeled as a set of spherical nodes and rods (i.e. truss representation). It considers limited, "steady" gradients in the 3 axis. To reduce data size, it establishes that the output of PLE is a Constructive Solid Geometry (CSG) instruction sequence to build the full lattice domain $\tilde{\Omega}$ and not the B-Rep of it. However, this decision only postpones the expense of Structural Analysis, which requires large number of finite elements. The PLE supports an $\mathbb{R}^3$ warping library, supporting cylindrical-, curve- and surface-driven gradings for the lattice geometries. Readers interested in the industrial application of Grasshooper for 3D printing may want to seek Ref. [6].

IntraLattice [10] is a software running on Grasshopper, whose functionality is the generation of graded lattice sets filling up a solid region $\Omega \subset \mathbb{R}^3$, with and without the boundary or skin of Ω, $\partial\Omega$. *IntraLattice* is able to (a) apply non-affine deformations upon the domain Ω, along its constitutive lattices to achieve smooth shapes, (b) generate a triangular B-rep of the lattice-made Ω domain and (c) post-process the B-Rep for actual Additive Manufacturing.

As application of the [FACE+thickness] and [EDGE+area] abstractions in the Computational Mechanics of Lattices area, Ref. [18] presents a process sequence for additive manufacturing of deformed lattices with links of circular cross sections, and gradient-driven geometry along the structure. Ref. [18] informs the articulation of a non-linear Cosserat stress-deformation iso-geometric solver [19] onto *IntraLattice* [10] to estimate the mechanical behavior of the $\tilde{\Omega}$ lattice set. Refs. [11] and [19] model the lattice set as a frame or truss structure, with torque being transmitted at the limb junctions. Ref. [11] executes a linear finite element analysis, while Ref. [19] performs a non-linear one.

2.2 Text vs. Visual Programming. Grasshopper

This section gives the reader a context to appraise the contributions of this manuscript from the point of view of computer programming education.

Ref. [9] compares 3 systems for Visual Computing applied to CAD (e.g. Grasshopper, used also in our implementation). The paradigms present in Visual Computing are *node-based* and *list-based*. This Ref. concludes that *Node-based* Visual programming (e.g. Houdini) presents theses advantages: (a)- combines iteration and encapsulation, (b)- supports both forward- and reverse- order modeling methods, (c)- has implicit iterative process, and, (d)- allows to define more complex processes. On the other hand, *List-based* Visual Programming (e.g. Grasshopper - Rhino, Generative Components CG -Microstation) presents higher difficulty, specially originated in : (1)- non-available or limited encapsulation, (2)- non-available iteration.

Ref. [3] compares the capacity of architects to program Computer Aided Architecture Design tools with script vs. visual programming. Both *Scripting* (e.g. Visual Basic AutoCAD -VBA) and *Visual* (e.g. Grasshopper - Rhino) tools use dialects internal to the host CAD software which are interpreted (as opposed to *compiled*). Beginner designers performed better with Visual Programming. These designers were able to streamline many repetitive and cumbersome tasks of the CAD user tasks. Both groups encountered problems (larger with Visual Programming) when devising complex programs in which fundamental concepts of algorithmic were needed.

Ref. [12] discusses common characteristics of *flow-based programming* FBP languages: (a) Input/Output black-box components in graph-like topology, (b) a manager of the information flow in the graph, (c) Graphic User Interface, (d) documentation and tutorials, (e) hands-on education. In FBP, the program flow is dictated by the characteristics of the instantaneous data, and is it not hard-wired (i.e. written) as a programming code. The clarification is relevant because this is the type of programming that was used for the present experiment.

Ref. [1] discusses End-user Programming (EUP) in which non-programmer users create applications for their particular work domains. Visual Programming Languages (VPLs) are one variety of EUPs. VPLs sub-divide into form-, flow- and rule-based programming. This particular reference addresses Robot Programming, by using a Flow-based Programming in which: (a) box icons represent functions or actions, and connectors represent data. (b) no flow control tokens (`while, for, repeat`) are available. (c) boxes are SIMO (Single Input Multi Output) functions, (d) semantics are located in the boxes and not in the connectors.

2.3 Pre-condition, Post-condition, Invariant. Contract-Based Programming

Ref. [13] discusses (a) operational, (b) denotational, and (c) axiomatic semantics. This reference addresses the semantics of an existing piece of code (*code* $\rightarrow$ *semantics*), in contrast with the intuitive or automated generation of a code to satisfy a semantical specification. In our work we informally use *axiomatic semantics*. Axiomatic semantics maps the variable states into logical predicates and the instructions into premises. Premises transform the states as proven by predicate logics. We work in the direction *semantics* $\rightarrow$ *code*. We do not apply formal logic predicate calculus, due to the assumption that the practitioner is neither a programmer nor a mathematician.

Ref. [7] dissects Pre-condition, Post-condition and Invariant of a Loop:

```
1 Pre: <initial program variables status>
2 WHILE <C: boolean condition for loop execution>
3 Inv: <invariant status of variables during Loop>
4 . . . keep the Invariant True while progress towards termination
5 . . .
6 ENDWHILE
7 Post:  <final program variables status>
```

They are predicates before, after or during a loop execution (Pre-, Post- and Invariant respectively). These are 1st-order logic predicates (**not instructions**). They are True when the program execution hits the corresponding lines (1,3,7). The Pre-condition (line 1) and Post-condition (line 7) describe the values of the *relevant* variables before and after the loop, respectively. The Invariant (line 3) pictures a typical un-finished state of variables when the loop is executing. One value of the invariant is that `Inv` $\wedge(\neg C)$ = `Post`. Therefore, the instructions in the loop (lines 4,5) must work towards $(\neg C)$ while keeping `Inv` being true. This last consideration dictates the instruction of the loop (lines 4,5).

Pre-, Post-conditions and Invariant are to be sketched, written or added by either the programmer or a program-by-contract editor. An informal process allows for the sketching of them and the programmer endeavoring to implement the sketch. An automated process includes typing the logic predicates and using an automated code-producer that adheres to them

Ref. [17] discusses the capabilities of ADA-2012 for Contract-baset Programming (CBP). In CPB, software piece specification is conducted by evaluating logic predicates about the computer status before and after the execution of the piece. CBP is supported by Pre- and Post-conditions to ensure Input/Output compliance with specifications. Code Contracts are independent of language and compiler. However, by applying ADA-2021, it is possible to also assess the compliance of object types and static/dynamic predicates.

Ref. [4] reports *Clousot*, a tool for Code Contract Checking. *Clousot* does: (1) check Pre- and Post-condition, (2) check common runtime errors, (3) infer Invariants., (4) allow for tuning of checking cost vs. precision, (5) allow domain refinement (pre-condition), (6) back-propagate conditions to determine software pieces, (7) infer precondition and postcondition inference from a given software piece, among other correctness tests and/or syntheses.

Ref. [16] addresses the absence of programming loops in Grasshopper and possible (cumbersome) repairs for this deficiency. For the present purposes, the important point is that, in absence of loop instructions, the application of Pre-condition, Post-condition and Invariant is apparently impossible. Yet, our project shows that the non-programmer Product Designer was able to enforce those logical predicates in Grasshopper and to obtain the correct voxel enumeration for the 3D region (i.e. solid).

2.4 Conclusions of the Literature Review

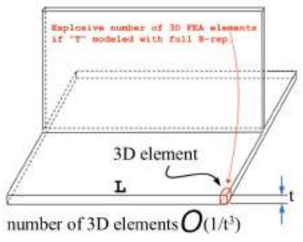
The examined literature shows the existence of commercial or quasi-commercial visual-computing generative software for lattice-based object (a) design and (b) mechanical simulation.

Regarding (a) above, the goal of this manuscript is to report the performance of a non-programmer product designer in programming generative geometry visual programming applications. This non-programmer designer is only equipped with intuitive notions of pre-conditions, post-conditions and invariants. It is well known that these notions are the mathematical kernel [7] of *imperative programming* and automated *imperative* code generation. Yet, the test reported in this manuscript pertains to the domain of visual programming (i.e. Grasshopper) and the capacity of this non-programmer product designer to compose a Grasshopper program for the approximation of a 3D object by lattice individuals of diverse topology and geometry.

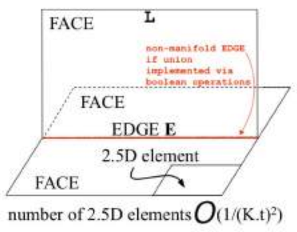
Regarding (b) above, the current state-of-art is the following: computational mechanics computations based on B-Reps of lattice structures is simply intractable, due to the massive size of the geometric or finite element models. As alternative, it is true that the truss/frame (also called [FACE+thickness] or [EDGE + area]) simplification of lattice structures indeed lowers the computational burden of stress - strain predictions for the lattice-based objects. This geometrical simplification allows the computations to finish (which is a considerable milestone). However, the mechanical soundness of this computation is at this time open for much clarification and improvement. These computational mechanics predictions are not, in any case, the goal of this manuscript.

To the best of our knowledge, no previous manuscripts report this special set of circumstances: (a) no programming experience, (b) use of imperative programming Pre-, Post-conditions and Invariant mathematical formulation, and (c) visual programming. We show that mathematical foundations of *imperative* programming are quite effective in helping a non-programmer designer to write a plug-in even if using visual programming Grasshopper.

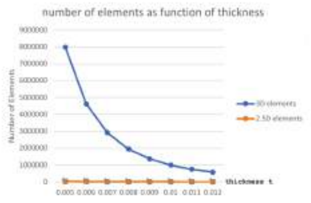
The particular domain of application for the above features is the population of a given domain (or region) $\Omega \subset \mathbb{R}^3$ with lattice individuals of diverse topologies and graded geometries. For the sake of computational resource savings in subsequent computational mechanics simulations, the [EDGE + area] (1.5D) and [FACE + thickness] (2.5D) truss formalism will be used.


Explosive number of 3D FEA elements
if "T" modeled with full B-rep
3D element
L
t
number of 3D elements $O(1/t^3)$

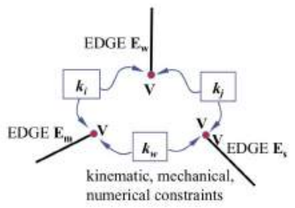
(a) Solid Modeling.



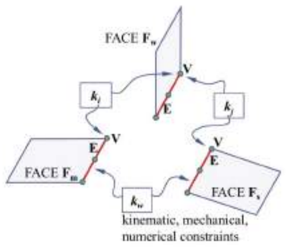
(b) FACE Modeling.



(c) Number of 3D vs 2.5D (FACE + thickness) elements in slender T shape.



(d) Kinematic Constraints on VERTEXs.



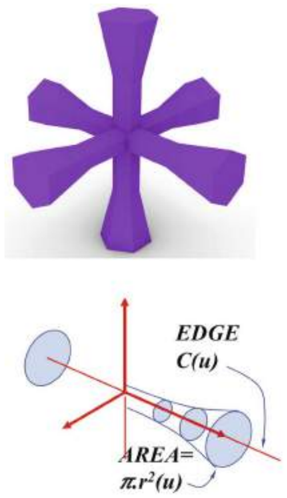
(e) Kinematic Constraints on EDGEs.

Fig. 1. Slender Shape Modeling. Order $O(n)$ of number of Finite Elements for the cases: Solid (3D) vs. FACE (2.5D) modeling. Savings by using 2.5D elements when $t \rightarrow 0$. Kinematic (or other) constraints as alternatives to Full B-Reps, while avoiding non-manifold topology.

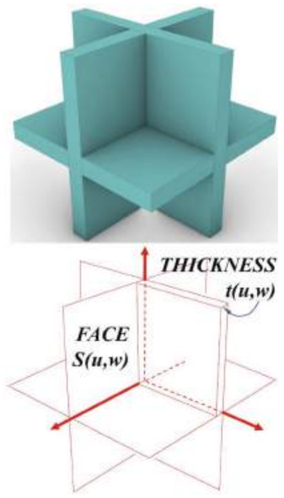
3 Methodology

3.1 Modeling of Slender Members

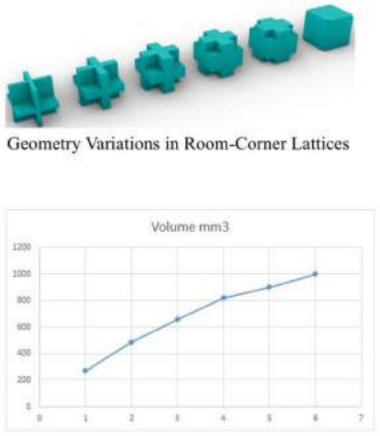
Construction of [EDGE + area] and [FACE + thickness] Limbs in Lattices. Slender members have a thin dimension (thickness, radius, or other) which is much smaller than the other member dimensions L. Full 3D B-Rep models decompose in a Topology Hierarchy such as BODYs, LUPMs, SHELLs, FACEs, LOOPs, EDGEs, VERTEXes, embedded in Geometries (e.g. Curves $C(u)$, Surfaces $S(u, w)$) in $\mathbb{R}^3$. The fact that $t/L \ll 1$ (Fig. 1(a)) causes the



(a) (EDGE + area) used to represent 3D Conic Cross Lattice.



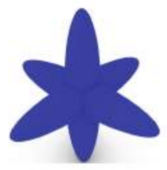
(b) (FACE + thickness) used to represent Room Corner Lattice.

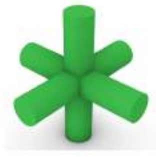


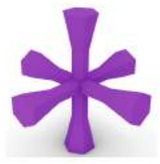
(c) Room-corner Lattice Family and it occupancy ratio.

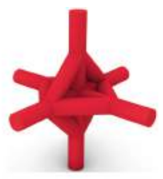
Fig. 2. Representation of 3D Conic Cross and Room Corner lattice individuals by using [EDGE + area] and [FACE + thickness] limbs, respectively.

number of finite elements to be very large ($O(1/t^3) \rightarrow \infty$ as $t \rightarrow 0$, Fig. 1(c)). When members are slender, their medial axes (a.k.a. skeletons) are 1-dimensional (set of curves) or 2-dimensional (set of surfaces). This manuscript addresses such cases. We do not address cases in which the medial axis is a mixture of 1- and 2-dimensional sets.


(a) Ellipsoidal 3D Cross


(b) Cylindrical 3D Cross


(c) Conical 3D Cross


(d) Diamond Kernel 3D Cross

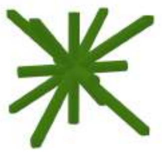

(e) 3D Asterisk

Fig. 3. [EDGE + area] (i.e. 1.5D) limb based Lattice Individuals.

[FACE + thickness] or 2.5D Limbs. The case in Fig. 1(b) shows subdivision in larger (and fewer) 2D tiles equipped with a thickness t. They are colloquially called *2.5D Finite* or [FACE + thickness] Elements. The number of 2.5D elements is in the order $O(1/t^2)$ (red trend, Fig. 1(c)), thus requiring less elements than full B-Rep 3D models, which are in the order $O(1/t^3)$ (both measures when $t \rightarrow 0$).

[EDGE + area] or 1.5D Limbs. Similarly, slender members whose medial axis is 1-dimensional look like wires (Fig. 1(d)), equipped with a cross section or area. They are colloquially called 1.5D or [EDGE + area] elements. Examples of the modeled [EDGE + area] limb-based lattice individuals appear in Fig. 3.

Manifold Enforcement in 1.5D and 2.5D Elements. [FACE + thickness] or [EDGE + area] finite elements do not admit boolean operations since the result might be non-manifold (Fig. 1(b)). Since manifold conditions are *sine qua non* ones, kinematic constraints K_i are enforced among 1.5D and 2.5D elements (Figs. 1(d), 1(e)), thus replacing the non-manifold prone boolean operations. In this manner, the solidarity among structural members in trusses or frames is ensured, without resorting to define a boolean union of their struts or plates.

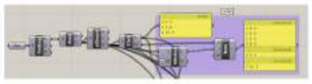
Figure 2(a) shows lattice individuals whose medial axis are 1-dimensional, showing the particular case of a 3D cross built by 6 conic rods. The EDGE is the medial axis $C(u)$ of the rod. The local rod cross section or area $\pi.r^2(u)$ is dependent on the parameter u parameterizing the rod medial axis.

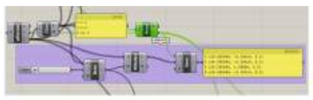
Figure 2(b) represents analogous situation, this time applied to lattices built with thin walls. The medial axis of the Room Corner lattice individual is a 2-dimensional set. Therefore, the simplification applicable here is the [FACE + thickness] modeling. The Room Corner lattice individual admits representation by 12 [FACE + thickness] elements. Each element contains a FACE $S(u, w)$ and a thickness map $t(u, w)$. Figure 2(c) shows Room-corner lattice individuals. It also displays diverse lattice individuals of this type, achieved by varying wall thickness and their proportion of occupied 3D space.

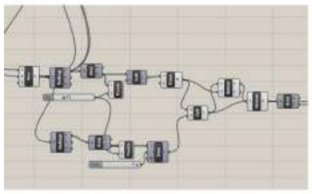
3.2 Lattice Family Creation via Visual-Programming

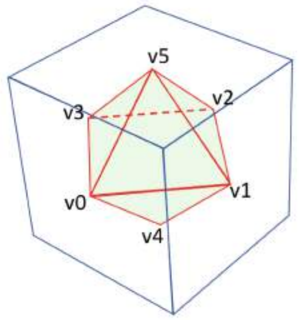
The visual code appearing in this manuscript is not polished or optimized. It appears exactly as the designer produced it, counting with no previous programming experience.

This section will discuss as example the Diamond lattice family. Figure 4 displays a partial view of the Grasshopper circuit for their construction. These lattice individuals *are not* full B-Reps, but instead a collection of 8 faces. The basic strategy is:

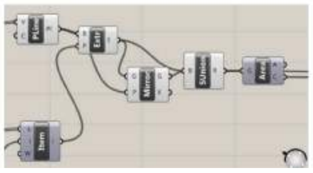

(a) Lattice Coordinates Order.

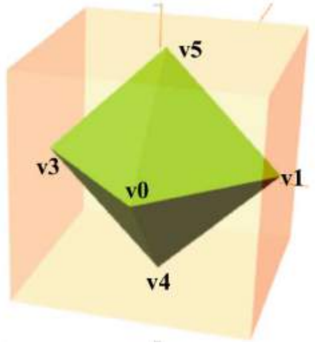

(b) Lattice Coordinates Retrieval.


(c) FACE creation and Pyramid Reflection.



(d) FACEs v0-v1-v5 and v0-v5-v3 in upper pyramid.


(e) Circuit for Reflection and FACE Boolean Union.



(f) Diamond Lattice Individual.

Fig. 4. Grasshopper Circuits for Diamond Family Lattice.

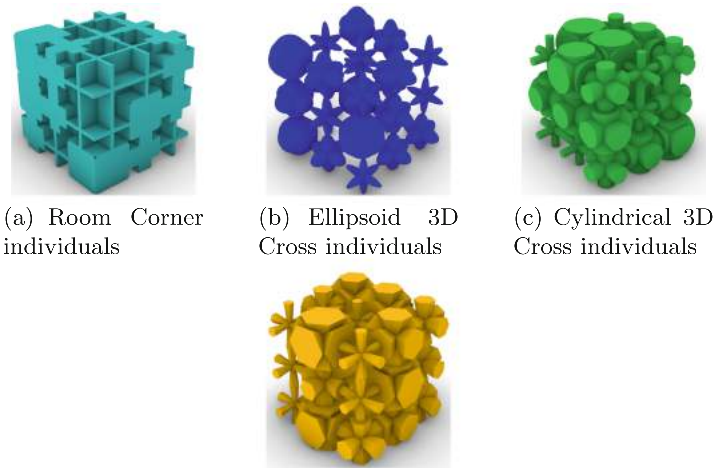

(a) Room Corner individuals (b) Ellipsoid 3D Cross individuals (c) Cylindrical 3D Cross individuals (d) Conical 3D Cross individuals

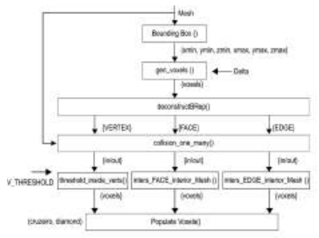
Fig. 5. Rectangular prismatic 3D region filled with $N_x \times N_y \times N_z$ lattices. Individuals with Constant Topology and Diverse Geometry.

(1) Create a Cube (B-Rep) which will circunscribe the diamond. (2) Explode the Cube into FACEs, EDGEs, VERTEXes. (3) Interrogate the FACEs for their geometric center. (4) Classify the FACE centroids as per z-coordinate, obtaining 3 bins with 1, 1 and 4 vertices ({v5},{v4},{v3,v0,v1,v2}, respectively (Figs. 4(a) and 4(f)). (5) Choose the bin with 4 FACE centroids ({v0,v1,v2,v3} in Figs. 4(d) and 4(b). Construct segments from these vertices to the cusp vertex, v5, of the pyramid (i.e. bin with 1 vertex, whose z coordinate is maximal). (6) Create (by extrusion, Fig. 4(c)) the triangular FACEs of the upper hemisphere by using the geometric centers of cube FACEs (e.g. FACE [v0,v1,v5] in Fig. 4(d)). (7) Reflect the upper pyramid (Figs. 4(c) and 4(d)) with respect to plane z=1/2.H to obtain the lower pyramid. The finished FACE-based Diamond lattice individual is displayed in Fig. 4(f).

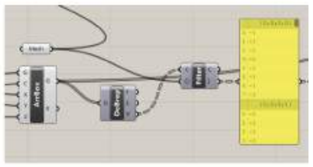
Pattern Grading. Figure 5 presents a case in which the *topology* of the lattice individual is kept along the domain, while their *geometry* is modified. This effect may be gradual or drastic (as shown in the figure for the sake of illustration). In any event, it is clear that the Visual Computing paradigm makes this form of Generative Design available to the Product Designer.

3.3 3D Region Lattice Occupancy

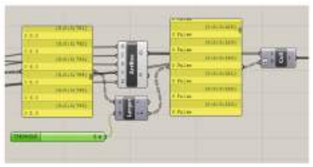
Figure 6(a) displays the generic workflow for the construction of the Lattice enumeration that fills a given B 3D domain representing a solid, with boundary representation M (2-manifold mesh).



(a) Workflow to determine Lattice occupancy in a 3D domain.



(b) Grading of VERTEXes of a given lattice as Inside (I) or Outside (O) a given mesh M.



(c) Grading of Lattices according to the threshold number of I VERTEXes (vertices Inside Mesh M).

Fig. 6. VERTEX-based Threshold for Lattice Inclusion in a 3D Region bounded by a manifold mesh **M**.

The following adjectives are used [5,20] for a given lattice: **I**=lattice is inside M, **O**=lattice is outside M, **NIO**=lattice is neither inside or outside M.

3.4 Application of Pre-, Post-condition and Invariants in Imperative and Declarative Programming

It is not possible to directly define an axiomatic semantics, as it is based on states and instructions, for declarative programming languages. Because of this reason, we have resorted to a 2-step process to reach a declarative visual program from Pre-, Post-condition and Invariant: (i) from Pre-, Post-condition and Invariant to an Imperative Program, and (ii) from an Imperative Program to a Declarative Visual Program.

It is not intention of this manuscript to discuss grammars, theorem-proving engines or code-generators that automatically execute translations (i) and (ii) above. We do not aspire to such an endeavor, and it would defeat the informal mental processes (i) and (ii) that our non-programmer Product Designer executes.

From Pre, Post, Inv to Imperative Programming. Algorithm 1 exemplifies how the Pre-, Post-condition and Invariant is applied for the population and approximation of a solid region *Omega* by using lattice individuals. Pre-, Post-condition and Invariants (lines 7, 19 and 9) are not instructions. Instead, they are comments that describe the status of the code execution at particular checkpoints. The set C (lines 5, 8, 16 and 17) of already estimated cells measures how advanced the population of $\widetilde{B}$ is. Growing C to equal the grid G (line 16) clearly means completion. However, the invariant must be kept (**IF** decision in line 10).

As expressed before, `Inv`$\wedge\neg(C \neq G) \Rightarrow$ `Post`. Effectively, $(\widetilde{B} \approx (\Omega \cap C)) \wedge (C = G) \Rightarrow (\widetilde{B} \approx (\Omega \cap G))$. In this manner, $\widetilde{B}$ approximates the solid Ω immersed in grid G. Notice that c_{ijk} is an empty cell of the grid, while l_{ijk} is a lattice individual (e.g. in Fig. 3) populating the space c_{ijk}.

Algorithm 1. Generic Pre- and Post-condition and Invariant for Approximation of B-Rep Ω.

```
1:  procedure [$\widetilde{B}$]=PopulateGrid($\Omega$,$G$)
2:              ▷ $c_{ijk}$:grid cell, $l_{ijk}$:lattice individual in cell $c_{ijk}$, $C$:set of cells $c_{ijk}$,
3:              ▷ $G$: axis-aligned grid containing solid $\Omega$, made of cells $c_{ijk}$
4:      $\widetilde{B} = \{\}$
5:      $C = \{\}$
6:      $c_{ijk} = first(G)$
7:                                                          ▷ Pre: solid $\Omega \subseteq$ grid $G$
8:      while ($C \neq G$) do
9:                                                          ▷ Inv: $\widetilde{B} \approx \Omega \cap C$
10:         if ( $(\Omega \cap c_{ijk}) \approx c_{ijk}$ ) then
11:             $l_{ijk} = RigidTransform(BasicLattice)$    ▷ e.g. Lattices in Fig. 3
12:         else
13:             $l_{ijk} = \Phi$
14:         end if
15:         $\widetilde{B} = \widetilde{B} \cup l_{ijk}$
16:         $C = C \cup c_{ijk}$
17:         $c_{ijk} = successor(c_{ijk}, G)$ )
18:     end while
19:                                                         ▷ Post: $\widetilde{B} \approx \Omega \cap G$
20: end procedure
```

From Imperative to Declarative Programming. Figure 7 presents the translation of Algorithm 1 to a generic equivalent of Grasshopper. For the sake of generality, this section uses generic feature names instead of Grasshopper names. In declarative visual programming, no `FOR` or `WHILE` iterations are available. Instead, `iterators` provides access to list elements (in this case, cells c_{ijk} of a grid G). A generic fuzzy lattice - in - solid inclusion diagnose is applied to each cel c_{ijk}. If a certain threshold number of elements FACE, EDGE or VERTEX of the lattice individual are inside Ω, the lattice c_{ijk} is considered INSIDE. This decision is implemented with the romboid icon gate that corresponds to line 10 of Algorithm 1. If the lattice c_{ijk} is considered to be Inside Ω, a rigid transformation is computed and applied to move the basic lattice individual (e.g. Fig. 3) to the position of lattice c_{ijk}, thus producing the lattice individual l_{ijk}. This lattice filling is added to the result set $\tilde{B}$ via a `set-ADD` operation. It must be pointed out that in declarative visual programming the *overall steady state* of the gate network is contains the result of the computation. The *transient states* are by definition unstable. Also, no specific instruction can be used for checkpoint, as in imperative programming, since individual instructions do not exist.

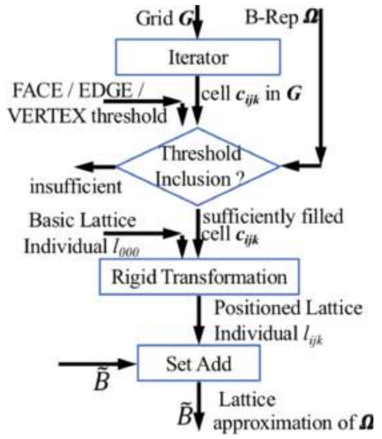
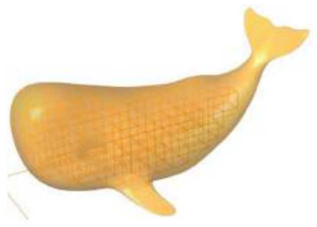

Grid G
B-Rep Ω
Iterator
FACE / EDGE / VERTEX threshold
cell c_{ijk} in G
Threshold Inclusion ?
insufficient
Basic Lattice Individual l_{000}
sufficiently filled cell c_{ijk}
Rigid Transformation
Positioned Lattice Individual l_{ijk}
Set Add
$\tilde{B}$
$\tilde{B}$
Lattice approximation of Ω

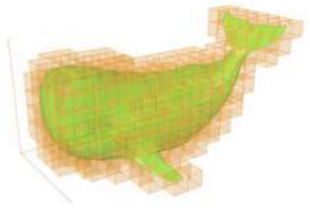
Fig. 7. Generic Declarative Equivalent of Algorithm 1

4 Results

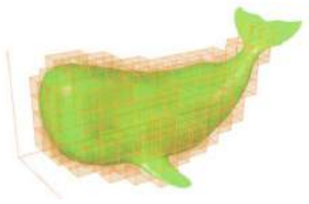
The workflow of Fig. 6(a): (1) identifies a rectangular prismatic Bounding Box enclosing M and orthogonally oriented w.r.t. World axes. (2) builds a 3D regular lattice grid within the Bounding Box, (3) for each cubic lattice, its FACE, EDGE and VERTEX sets are extracted (BRep deconstruction), (4) for the particular topology e.g. VERTEX, the number of VERTEXes inside and outside the mesh M is identified (Fig. 6(b)), (5) a Threshold value is used (Fig. 6(c)) to decide whether the **NIO** lattices are graded as **I** (e.g. a lattice is considered I if more than Threshold=5 of its VERTEXes are inside M), (6) once the lattice is graded as **I**, a rigid transformation is applied on a generic copy of that particular lattice individual (e.g. diamond, 3D cross, room corner, etc.) to populate the region enclosed by mesh M with the lattice individuals graded as **I** (i.e. inside M). Results for the *Whale* dataset of this lattice enumeration process are displayed in Figs. 8 and 9.



(a) Lattice Enumeration for (demanding) Vertex Threshold=8.



(b) Lattice Enumeration for FACE Threshold=1.



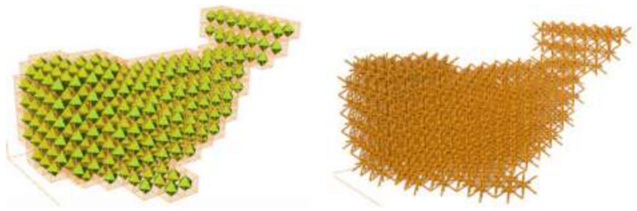
(c) Lattice Enumeration for FACE Threshold=5.

Fig. 8. Whale data set. Threshold number of Topologies (EDGE or FACE) needed to declare a lattice as **I** (Inside)

4.1 Lattice Enumeration

Figure 9 shows diverse lattice populations of the Whale dataset. This particular figure displays actual FACE sets building the lattice individuales of all cells. No [FACE + thickness] or [EDGE + area] simplifications are executed.

Figure 10 depicts results of the visual computing circuits applied to approximate the Cat dataset with lattice individuals of the type Asterisk. Figure 10(a) shows the 2-manifold Triangular Boundary Representation (i.e. mesh M), or skin, enclosing a solid body B. Figure 10(b) displays the immersion of the B-Rep M into an orthogonal cubic grid. Figure 10(c) shows an approximation of the Cat M with [EDGE + area] type limbs (type 3D Asterisk). An over-estimation of B is achieved if a lattice receives an I (Inside) grade when one of its six VERTEXes is Inside mesh M. A more sensible grading of a lattice as **I** occurs if a large portion its VERTEXes are inside M.


(a) Diamond. (b) 3D Asterisk.

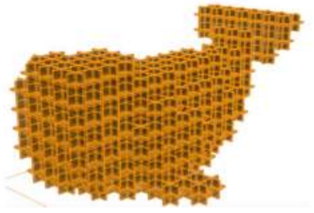

(c) Room Corner.

Fig. 9. Results of assorted Lattice individuals occupancy of the Whale 3D model.

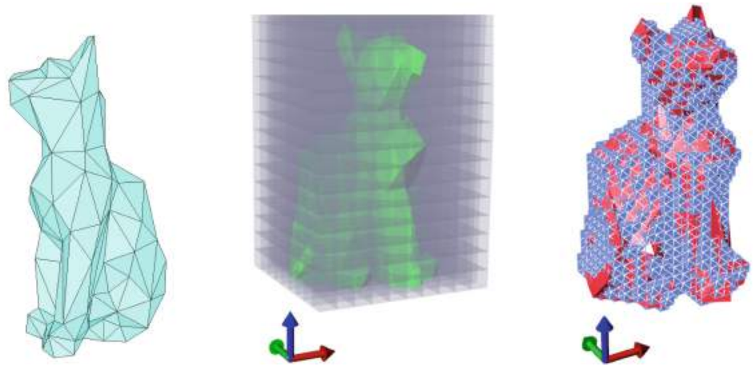
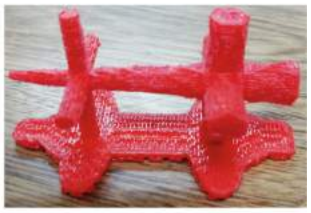

(a) 3D Model. Cat data set. (b) Grid immersion of model. (c) Trusses within Latices.

Fig. 10. Cat data set. Results of visual programming processing: immersion of model in grid, lattice approximation and inner lattice trusses.

4.2 Material Realization

Figure 11 presents an Additive Manufacturing realization of two individuals of the family Conical 3D Cross (Fig. 3(c)). In this material incarnation, the section radius is decreasing along the limb axis curve. In Fig. 3(c), the section radius grows along the limb axis curve. They obviously belong to the same lattice family. However, it must be emphasized that the *kernel of this manuscript is not the material realization of the lattice family*, but instead the successful application of Contract-Based Programming (Pre-condition, Post-condition, Invariants) in Flow-based Programming by a non-programmer Designer and the 1.5D and 2.5D truss - frame modeling strategies.

(a) View 1


(b) View 2

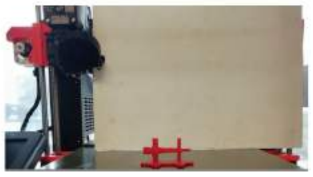

(c) Set up in a Coordinate Measurement Machine

Fig. 11. Additive manufacturing realization of two Conical 3D Cross lattice individuals of family in Fig. 3(c).

5 Conclusions and Future Work

Visual Programming for a Non-programmer Designer

The experiment of having a non-programmer Product Designer to program a lattice-based filling of a 3D region using a set of suitable lattice topologies with varying geometry had these circumstances: (a) no previous programming training, (b) independent learning (i.e. absence of programming tutors), (c) use of visual programming tools (i.e. Grasshopper), (d) informal seminars on Pre- and Post-conditions and Invariants [7].

In spite of Pre- and Post-conditions and Invariants being devised for *imperative* languages, the Product Designer used Grasshopper, a *flow-based* programming language, with high proficiency and technically correct results. It must be noticed however, that it was not intention of the present research to compete against professional lattice-based design tools.

Therefore, this work shows the power of the Visual programming paradigm and tools for the particular domain of lattice-based geometric modeling. In particular, the change of paradigm from iteration-based imperative programming towards flow-based programming (e.g. in Grasshopper) does not impede the novice visual programmer to enforce code correctness and pre- and post-condition clauses.

Region Population with Lattice Individuals

Several families of slender-limb lattice individuals have been modeled. They include(a) rods modeled as [EDGE + area] (a.k.a. 1.5D) elements and (b) plates modeled as [FACE + thickness] (a.k.a. 2.5D) elements.

The approximation of domain Ω by lattices is tuned (as usual in enumerations) by considering inclusion thresholds above which a *partially* included lattice is declared to be *full*.

A reduced Additive Manufacture of (two) lattice individuals of the family Conical 3D Cross is presented to illustrate how they spatially and kinematically relate to each other. However, it most be noticed that the manufacture of the approximation to the 3D region Ω via lattices is not the purpose of this manuscript.

We do not seek to equate informal seminars on pre-, post-conditions and invariants for a non-programmer Product Designer with the complex, very long and formal process that formal logic represents for even trained programmers. All what we can aspire is that an informal treatment, allows the non-programmer Product Designer to be proficient vis-a-vis development of declarative visual programs and correction of his/her run results.

Future work is required in the application of non-affine geometric transformations to the lattice individuals before their docking in the target 3D region. This extension would permit lattices whose geometry contains straight-to-curve deformations, dictated by the functionality of the Additive Manufacturing.

Acknowledgments. The authors wish to thank U. EAFIT, Colombia for the funding and academic space for this investigation, channeled through its CAD CAM CAE Laboratory. Special thanks are presented to M.A. Carolina Builes-Roldan for funding an important portion of the project.

Conflicts of Interest. The Authors declare that they have no conflict of interest in the publication of this manuscript.

References

1. Alexandrova, S., Tatlock, Z., Cakmak, M.: RoboFlow: a flow-based visual programming language for mobile manipulation tasks. In: 2015 IEEE International Conference on Robotics and Automation (ICRA), pp. 5537–5544 (2015). https://doi.org/10.1109/ICRA.2015.7139973
2. Aremu, A., et al.: A voxel-based method of constructing and skinning conformal and functionally graded lattice structures suitable for additive manufacturing. Addit. Manuf. **13**, 1–13 (2017)
3. Celani, G., Vaz, C.E.V.: CAD scripting and visual programming languages for implementing computational design concepts: a comparison from a pedagogical point of view. Int. J. Architectural Comput. **10**(1), 121–137 (2012). https://doi.org/10.1260/1478-0771.10.1.121. ISSN 1478-0771
4. Fähndrich, M., Logozzo, F.: Static contract checking with abstract interpretation. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 10–30. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18070-5_2
5. Garcia, M.J., Henao, M.A., Ruiz, O.E.: Fixed grid finite element analysis for 3D structural problems. Int. J. Comput. Methods **02**(04), 569–586 (2005). https://doi.org/10.1142/S0219876205000582

6. Garcia-Cuevas, D., Pugliese, G.: Advanced 3D Printing with Grasshopper: Clay and FDM. Independent (2020). iSBN 979-8635379011
7. Gries, D.: The Science of Programming. Springer, New York (1981). iSBN 978-0-387-90641-6, eISBN 978-1-4612-5983-1. Chapter Developing Loops from Invariants and Bounds
8. Gupta, A., Kurzeja, K., Rossignac, J., Allen, G., Kumar, P.S., Musuvathy, S.: Programmed-lattice editor and accelerated processing of parametric program-representations of steady lattices. Comput. Aided Des. **113**, 35–47 (2019)
9. Janssen, P., Chen, K.: Visual dataflow modelling: a comparison of three systems. In: Design Futures 2011 - Proceedings of the 14th International Conference on Computer Aided Architectural Design Futures, Liee, Belgium, pp. 801–816 (2011)
10. Kurtz, A., Tang, Y., Zhao, F.: Intra lattice (2015). http://intralattice.com, generative Lattice Design with Grasshopper. McGill Additive Design and Manufacturing Laboratory - ADML
11. Montoya-Zapata, D., Cortes, C., Ruiz-Salguero, O.: Fe-simulations with a simplified model for open-cell porous materials: a kelvin cell approach. J. Comput. Methods Sci. Eng. 1–12 (2019, in press). https://doi.org/10.3233/JCM-193669. Published online: 27 May 2019
12. Morrison, J.P.: Flow-Based Programming: A New Approach to Application Development, 2nd edn. CreateSpace Independent Publishing Platform (2010). ISBN-10: 1451542321, ISBN-13: 978–1451542325
13. Nielson, H., Flemming, N.: Semantics with Applications: An Appetizer. Undergraduate Topics in Computer Science (UTiCS) Series. Springer, London (2007). iSBN-13: 978-1-84628-691-9, e-ISBN-13: 978-1-84628-692-6
14. Panesar, A., Abdi, M., Hickman, D., Ashcroft, I.: Strategies for functionally graded lattice structures derived using topology optimisation for additive manufacturing. Addit. Manuf. **19**, 81–94 (2018). https://doi.org/10.1016/j.addma.2017.11.008
15. Schwarzkopf, M.: The remarkable utility of dataflow computing (2020). https://www.sigops.org/2020/the-remarkable-utility-of-dataflow-computing/, ACM - SIGOPS. Special Interest Group in Operating Systems
16. Sebestyen, A.: Loops in grasshopper. In: Bricks are Landing. Algorithmic Design of a Brick Pavilion, pp. 15–24. T.U. Wien (2019). iSBN 978-3-9504464-1-82
17. Wang, B., Gao, H., Cheng, J.: Contract-based programming for future computing with Ada 2012. In: 2016 International Conference on Advanced Cloud and Big Data (CBD), pp. 322–327 (2016). https://doi.org/10.1109/CBD.2016.062
18. Weeger, O., Boddeti, N., Yeung, S.K., Kaijima, S., Dunn, M.: Digital design and nonlinear simulation for additive manufacturing of soft lattice structures. Addit. Manuf. **25**, 39–49 (2019). https://doi.org/10.1016/j.addma.2018.11.003. https://www.sciencedirect.com/science/article/pii/S2214860417303962
19. Weeger, O., Yeung, S.K., Dunn, M.L.: Isogeometric collocation methods for Cosserat rods and rod structures. Comput. Methods Appl. Mech. Eng. **316**, 100–122 (2017). https://doi.org/10.1016/j.cma.2016.05.009. https://www.sciencedirect.com/science/article/pii/S004578251630336X, special Issue on Isogeometric Analysis: Progress and Challenges
20. Xie, Y.M., Steven, G.: A simple evolutionary procedure for structural optimization. Comput. Struct. **49**(5), 885–896 (1993)