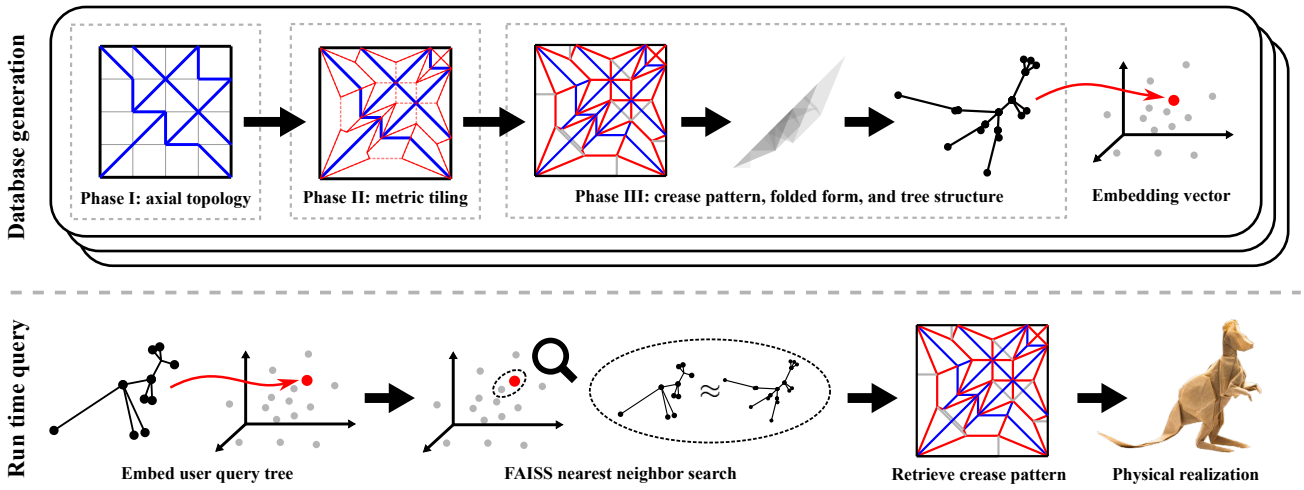
Graphical Abstract

SEARCH 22.5: Spectral Embedding Archive for Rational Crease Pattern Hyperspace

Anonymized for review



Highlights

SEARCH 22.5: Spectral Embedding Archive for Rational Crease Pattern Hyperspace

Anonymized for review

- Database precomputation → online lookup as inverted design paradigm for origami
- Straight skeleton simplification formulated as mixed-integer optimization
- Rational 4D basis for exact geometric computation in $\left[\mathbb{Q}(\sqrt{2})\right]^2$
- Spectral embedding techniques utilized to compare origami crease pattern

SEARCH 22.5: Spectral Embedding Archive for Rational Crease Pattern Hyperspace

Anonymized for review

ARTICLE INFO

Keywords:
origami design
spectral embedding
computational geometry

ABSTRACT

Systematic computational design of 22.5° uniaxial origami remains a complex challenge in geometry processing. While traditional origami CAD tools utilize forward-mapping optimization formulated as a circle/polygon packing problem, enforcing strict 22.5° angular quantization transforms the design space into a highly non-convex, mixed-integer nonlinear programming (MINLP) problem. Furthermore, intersections within this angle system generate coordinates in the quadratic field extension $\mathbb{Q}(\sqrt{2})$, rendering conventional floating-point solvers difficult to utilize. To bypass these runtime intractabilities, we present an inverted computational paradigm. Rather than forward-mapping a target tree to a crease pattern constructed at runtime, we shift the combinatorial burden to an offline precomputation phase. We extensively enumerate millions of valid 45° axial topologies using Boolean Satisfiability (SAT), realize flat-foldable geometric tilings utilizing a Mixed-Integer Linear Programming (MILP) constraint solver, and construct the interior straight skeletons using a custom 4D rational basis to handle coordinates in $\mathbb{Q}(\sqrt{2})$ with exact precision. To map user intent to this massive precomputed dataset, we utilize the Cumulative Wave Kernel Signature (CWKS) to embed the tree structure of each generated crease pattern into a continuous embedding space. This enables a robust nearest-neighbor search to retrieve strictly foldable 22.5° geometries that best approximate the user's intent.

1. Introduction

One of the biggest unsolved problems in origami design is the ability to systematically design models in the 22.5° angle system, in which creases meet at angles of multiples of 22.5°. This angle system is used in the oldest known traditional origami designs, such as the crane, and is still used in some of the most popular modern-day designs. The geometry of a 22.5° crease pattern is unique from other design methods for its subtle symmetry that consistently makes for satisfying folding experiences and elegant structures quintessential to origami. Despite this, there remains no known algorithm to systematically design new models in the 22.5° angle system. Instead, most origami designers design 22.5° models almost entirely through a combination of trial and error and an experienced artist's intuition.

Most representational origami design algorithms focus on uniaxial designs, which are structures that can be represented by a tree graph. Existing approaches, such as circle packing [1] utilize systematic forward-design procedures. By modelling each leaf node of the tree as a non-overlapping circle or polygon in the square, the design problem reduces to a packing optimization problem. It is theoretically possible to adapt these algorithms for 22.5° designs by injecting rigid integer constraints into the nonlinear solver to enforce the angle constraints. However, even if this massive mixed-integer nonlinear optimization problem is solved, there is no known way to convert the packing into a flat foldable crease pattern due to a phenomenon known as "dense bouncing", arising from the fact that 22.5° angles generate lengths in the quadratic field extension $\mathbb{Q}(\sqrt{2})$[2]. Consequently, because these irrational vertex coordinates are incommensurate with any rational grid, they cannot be resolved by any finite geometric subdivision [3], nor does the design space accomodate algorithms [1] that generate creases at arbitrary non-22.5° angles. Therefore, the problem of 22.5° origami design is not only a problem of optimization, but also a problem of combinatorial geometry.

In this paper, we present an inverted approach to 22.5° origami design inspired by the elementary design method of modifying known traditional bases into new subjects [3]. Instead of solving the intractible problem of tree → packing → crease pattern at runtime, we shift the computational burden to an extensive precomputation phase. We first blindly precompute a massive database of valid 22.5° crease patterns, embed the crease patterns into a high-dimensional vector space based on their extracted tree structures, then execute a nearest neighbor search at runtime to approximate the user's desired tree with the closest precomputed crease pattern. Empirical results demonstrate the effectiveness of this approach for reasonably complex user inputs.

The paper is structured as follows. Prior work is reviewed in Section 2. The three phases of the database generation process are detailed in Section 3, Section 4, and Section 5. The embedding scheme and nearest neighbor search are described in Section 6. Finally, the software implementation and empirical results thereof are discussed in Section 7. The paper concludes with a discussion of limitations and future work.

2. Related Work

2.1. Algorithmic Origami Design

Flat-foldability is an important property for "well behaved" crease patterns and has been extensively studied.

ORCID(s):

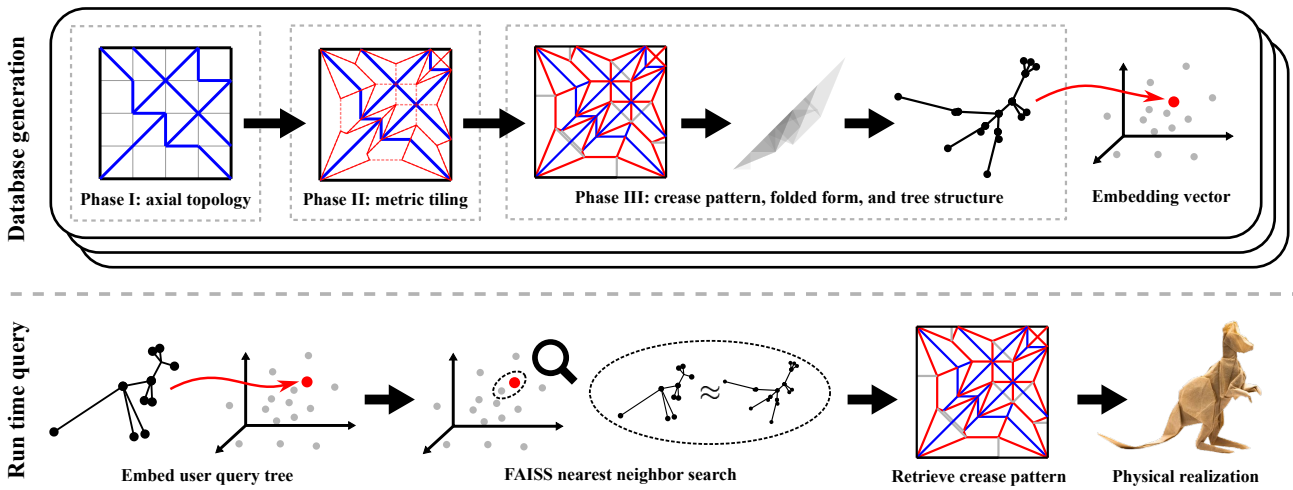


Figure 1: Graphical abstract of the SEARCH 22.5° pipeline. First, a database of valid 22.5° crease patterns is precomputed and embedded in a continuous vector space. At run time, a nearest neighbor embedding search of the user's input tree retrieves the crease pattern that best approximates the user's desired tree structure.

Kawasaki's theorem constrains the crease angles for the pattern to fold flat kinematically [4], while Maekawa's theorem and other mountain-valley assignment constraints prevent self intersection of the paper [5]. Additionally, algorithms and design tools have been produced for specific cases of geometric origami design, including polyhedra [6], rotationally symmetric revolutes [7], and tessellated surfaces [8].

For conventional representational origami design, the best-understood formalization was first established by Lang as a circle packing algorithm [1]. Lang demonstrated that any uniaxial tree graph (which can represent most animals or similar subjects) could be realized as a flat-foldable crease pattern with two steps: first, solve a nonlinear optimization problem to pack non-overlapping circles into a square domain, forming a network of **axial creases** connecting the centers of tangent circles. Then, generate **ridge creases** within each axial polygon to bring the axial creases to lie collinearly along a common folded axis using a "universal molecule" construction (a generalization of the straight skeleton [9]). Refer to Figure 2 for an illustration of this process.

For the sake of easier foldability, this algorithm was later modified to pack polygons instead of circles, leading to box pleating on a square grid [10, 11] and hex pleating on a triangular isometric grid [12]. The former is used in one of the most prominent modern origami design softwares, Box Pleating Studio [13]. In contrast, the 22.5° angle system has no known systematic design algorithm, though some geometric properties [14, 2] have been characterized, and many artists have successfully designed 22.5° models by hand. One elementary "manual" origami design method is to take a known base, such as the base of a traditional model, and adapt it into a new subject with a similar structure [3]. However, this method is limited by the number of known public domain bases, and is generally limited to low-complexity structures.

2.2. Topological Shape Retrieval

When a database of geometries is established, querying it requires mapping structural user intent to topological data. Conventional CAD databases often rely on exact subgraph isomorphism to retrieve 3D boundary representations [15]. However, because origami trees inherently contain topological noise (i.e., unintentional short flaps), isomorphic matching yields high rejection rates for physically viable bases.

To mitigate this, modern geometry processing often uses spectral graph embeddings derived from the Laplacian matrix [15]. One embedding scheme is the Wave Kernel Signature [16], which characterizes shapes by modeling the energy dispersion of quantum particles. Finally, once an embedding scheme is established, the open-source FAISS library [17] enables efficient nearest-neighbor search across millions of high-dimensional vectors.

3. Database Generation Phase I: Topology Enumeration

The conventional circle packing algorithm utilized by TreeMaker [1] generates crease patterns with a forward mapping process involving a continuous nonlinear optimization problem at runtime. However, attempting to enforce strict 22.5° angular quantization onto this continuous packing algorithm transforms the formulation into a highly intractable non-convex mixed-integer nonlinear optimization problem.

To bypass this computational bottleneck, we invert the design procedure. Circle packing first generates a **packing** (an allocation of the paper into specific flaps of the input tree), then extracts the **tiling** (network of axial creases that define molecule boundaries) based on tangent circles of the packing. We reverse this causality, as illustrated in Figure 2. We begin by partitioning the paper into a valid planar graph of polygons to establish the tiling, then fill in the crease pattern and extract the tree as a product thereof. This replaces

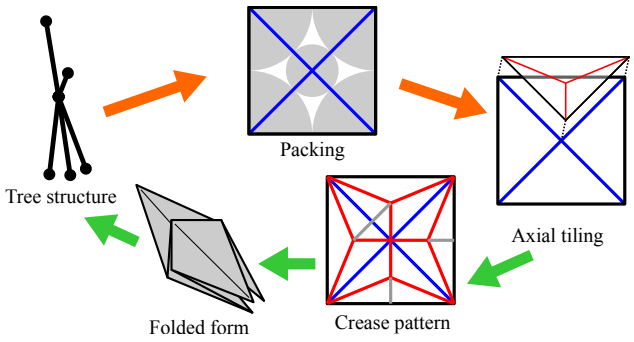


Figure 2: The inverted pipeline. The conventional TreeMaker algorithm (orange arrows) forward-maps an input tree structure into an axial tiling. In our proposed database generation procedure (green arrows), we begin with an axial tiling and extract the resulting tree structure.

the runtime forward solve with an enumerative precomputation process.

Converting the tiling into a crease pattern entails filling each polygon with ridge creases such that all axial creases lie collinearly along a common axis when folded. We achieve this using the straight skeleton [9] (detailed in Section 4), in which ridge creases are formed along angle bisectors of the polygon boundaries. Because our system strictly requires all final creases to be multiples of 22.5°, this bisection property requires all axial creases to run at multiples of 45°.

Before generating a fully defined metric tiling, we must generate the underlying **topology**—that is, the angles and connective relationships of the axial network, disregarding crease lengths. Because the network consists exclusively of orthogonal and 45° diagonal lines, we can exhaustively enumerate them by embedding the problem onto an $N \times N$ integer grid.

We model this enumeration as solutions to a Boolean Satisfiability (SAT) problem and solve it using the open-source SMT solver Z3 [18]. The boundary of the paper is fixed, comprising $4N$ predefined boundary segments. Inside the grid, every possible orthogonal and diagonal segment connecting adjacent integer coordinates is treated as a potential axial crease. We assign a Boolean decision variable $e \in \{0, 1\}$ to every internal edge.

The total number of internal Boolean variables for an $N \times N$ grid is precisely $4N^2 - 2N$. The SMT solver's task is to sort through the 2^{4N^2-2N} possible assignments to find unique topologies that satisfy the following rules.

3.1. Local Z3 Geometric Constraints

To prune the inherently exponential search space, we restrict the SMT solver using a set of core local constraints:

3.1.1. Planarity

An axial topology must be planar. The two diagonals within any single 1×1 cell would cross if both were present. Therefore, they are mutually exclusive. For the two diagonal Boolean variables d^1 and d^2 in every cell (x, y), we assert the single CNF clause:

$$(\neg d^1_{x,y} \vee \neg d^2_{x,y}) \tag{1}$$

3.1.2. No Dangling Edges

An axial crease must not terminate abruptly in the middle of the paper. We define the *degree* of a vertex as the number of active edges incident to it. Then, every internal vertex in the grid must have a degree of either 0 or > 1. For every vertex v and its set of incident edges E_v, we apply a pseudo-Boolean cardinality constraint to ensure the degree is never exactly 1:

$$\sum_{e \in E_v} e \neq 1 \tag{2}$$

3.1.3. No T-Junctions

To prevent dense bouncing, focus the design space, and simplify the subsequent crease pattern generation process, we enforce a T-junction rule. This constraint is conservative and not strictly necessary for valid axial topologies. For every internal vertex $(x, y) \in [1, N-1]^2$, we evaluate its 8 potential neighbors[1], grouped into four opposite pairs: $P_1 = (R, L)$, $P_2 = (U, D)$, $P_3 = (UR, DL)$, and $P_4 = (UL, DR)$. Let the two Boolean variables representing the edges of pair k be a_k and b_k.

If *any* pair i is continuously active, then *all* intersecting pairs j must share identical Boolean states. Logically, this implication is $(a_i \wedge b_i) \implies (a_j \iff b_j)$. To express this in CNF for the solver, for every pair $i \in \{1..4\}$ and every pair $j \in \{1..4\}$, we assert:

$$(\neg a_i \vee \neg b_i \vee a_j \vee \neg b_j) \wedge (\neg a_i \vee \neg b_i \vee \neg a_j \vee b_j) \tag{3}$$

3.1.4. Optional Symmetry Constraint

We can optionally enforce global symmetry by identifying symmetric edges via a coordinate transformation T. For every internal edge e and its counterpart e_{sym} across the line of symmetry, we enforce equivalence $(e \iff e_{sym})$. In CNF, this expands to two clauses:

$$(e \vee \neg e_{sym}) \wedge (\neg e \vee e_{sym}) \tag{4}$$

3.2. Global Topological Constraints

Once the SAT solver yields a satisfying assignment based on the local geometric constraints, we extract the resulting graph $G = (V, E)$ and pass it through a global topology filter before accepting it into the database.

3.2.1. Connectivity Constraint

First, we require the graph to be a single connected component. Disconnected components represent isolated polygons floating within other polygons. This is efficiently filtered using standard graph traversal algorithms.

[1] where R, L, U, D, UR, DL, UL, and DR denote the right, left, up, down, upper right, lower left, upper left, and lower right neighbors respectively

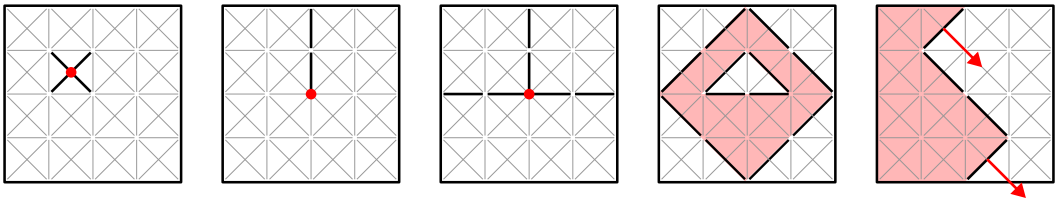


Figure 3: Illustration of the possible local and global topology constraints violations (not including symmetry). From left to right: (1) Two crossing diagonals violate planarity; (2) A dangling degree-1 vertex; (3) A T-junction; (4) A disconnected graph forming a floating polygon; (5) A polygon with non-unique side normals.

3.2.2. Unique Normals Constraint

Second, we strictly enforce that all topological faces must consist of sides with unique directions, evaluating sequential collinear segments as a single entity. While this permits concave polygons, it intrinsically bounds the maximum number of sides to 8 and prevents complex or tightly zigzagging polygonal boundaries that would complicate subsequent crease pattern generation.

We utilize a Doubly-Connected Edge List (DCEL) half-edge traversal to extract all faces of the graph [19]. For every vertex v, its incident neighbors are sorted angularly using `arctan2(dy,dx)`. Walking counter-clockwise around the leftmost turns systematically extracts the closed loops that define the internal polygonal faces.

For each extracted face, we calculate the simplified directional vector of every boundary segment:

$$\vec{d} = \left(\frac{\Delta x}{\gcd(\Delta x, \Delta y)}, \frac{\Delta y}{\gcd(\Delta x, \Delta y)} \right) \tag{5}$$

and enforce that all directional vectors within the same face are unique:

$$\vec{d}_i \neq \vec{d}_j \quad \forall i, j \in \{1, 2, \ldots, k\},\ i \neq j \tag{6}$$

where k is the total number of non-collinear boundary segments defining the face.

With these local Boolean constraints and global topology filters, the enumeration pipeline can produce a theoretically unlimited supply of valid 22.5° axial topologies as N increases. In practice we constrain $N \leq 5$, which alone is estimated to generate at least an order of 10^8 unique crease patterns (see Table 1)

4. Database Generation Phase II: Tiling Realization

The output of the topology enumeration phase is an unweighted planar graph with vertices initialized on an integer grid. We now seek to adjust the lengths of the topology (that is, move the vertices) into a fully defined metric **tiling**.

If a topology contains n vertices, we theoretically possess $2n$ degrees of freedom in the Euclidean plane (moving each vertex in x and y). Our objective is to apply constraints until all degrees of freedom are removed, and the tiling is fully locked into a single crease pattern with fully defined references.

4.1. Basic Geometric Constraints

We begin by establishing the absolute requirements of the tiling.

4.1.1. Angle preservation constraints

Every edge in the initial grid topology is aligned to a multiple of 45°. As the vertices move to form the exact tiling, these edges must remain parallel to their original orientations. Let $\vec{v}_i$ and $\vec{v}_j$ be the positions of two vertices connected by an edge, and $\hat{n} = [n_x, n_y]^T$ be the fixed normal vector of that edge. We enforce the following angular constraint for every edge:

$$(\vec{v}_j - \vec{v}_i) \perp \hat{n} \implies \hat{n} \cdot (\vec{v}_j - \vec{v}_i) = 0 \tag{7}$$

$$n_x x_j - n_x x_i + n_y y_j - n_y y_i = 0 \tag{8}$$

Note that this constraint is linear and homogeneous.

4.1.2. Boundary conditions

To prevent rigid-body translation, rotation, and scaling, the bottom-left vertex and top-right vertex are constrained to $(0, 0)$ and (N, N) respectively. Note that the latter provides an inhomogeneous constraint, which prevents the solver from collapsing all vertex coordinates to 0.

4.1.3. Symmetry constraints

If the topology was generated with a symmetry constraint, we apply the corresponding symmetry constraints to the tiling realization. For book symmetry across the vertical axis $x = N/2$, every symmetric pair of vertices v_i, v_j must satisfy:

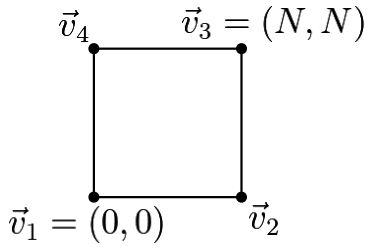
$$x_i + x_j = N \quad \text{and if } i \neq j, \quad y_i - y_j = 0 \tag{9}$$

For diagonal symmetry across $y = x$, every pair of symmetric vertices v_i, v_j must satisfy:

$$x_i - y_j = 0 \quad \text{and} \quad y_i - x_j = 0 \tag{10}$$

We can then combine these linear constraints into a single matrix form $A\vec{x} = \vec{b}$, where $\vec{x} = [x_1, y_1, x_2, y_2, \ldots, x_n, y_n]^T$ is a vector containing all vertex coordinates, each row of A corresponds to one of the aforementioned linear constraints, and $\vec{b}$ is a vector of constants (mostly zeros except for the few inhomogeneous conditions).

As a minimal example, consider the constraints imposed on the empty square (including optional diagonal symmetry



$$\begin{array}{l} \text{Edge angle constraints} \\ \\ \\ \\ \text{Boundary pinning} \\ \\ \\ \\ \text{Diagonal symmetry} \\ \\ \\ \end{array} \underbrace{\left[\begin{array}{cccccccc} 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 \\ 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ \hline 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ \hline 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 1 & 0 & 0 & -1 & 0 \end{array}\right]}_{A} \underbrace{\left[\begin{array}{c} x_1 \\ y_1 \\ x_2 \\ y_2 \\ x_3 \\ y_3 \\ x_4 \\ y_4 \end{array}\right]}_{\vec{x}} = \underbrace{\left[\begin{array}{c} 0 \\ 0 \\ 0 \\ 0 \\ \hline 0 \\ 0 \\ N \\ N \\ \hline 0 \\ 0 \\ 0 \\ 0 \end{array}\right]}_{\vec{b}} \tag{11}$$

Figure 4: The basic geometric constraints on the empty square tiling. The left figure shows the tiling with the 4 vertices and 4 edges. The right figure shows the corresponding linear system $A\vec{x} = \vec{b}$, where each row of A corresponds to a constraint (edge angle, boundary pinning, or symmetry).

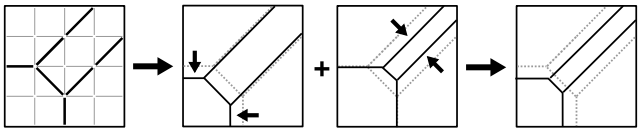


Figure 5: The unconstrained degrees of freedom of a topology "slide" edges around without changing edge angles or connectivity. Left: the initial topology initialized on grid positions. Middle: the topology has two independent degrees of freedom under the basic constraints established in Section 4.1, each corresponding to a column of null(A). Right: superimposing displacements along the two degrees of freedom turns the topology into a specific tiling.

constraints). The global system $A\vec{x} = \vec{b}$ with the basic geometric constraints takes the form of Equation 11.

The metric tiling is fully defined if and only if the matrix A has full rank. If the matrix is rank-deficient, then there are still degrees of freedom for vertices in the tiling to move without violating constraints. An example of such motions is shown in Figure 5. These motions are described by columns of null(A). To fully constrain the axial tiling (that is, ensure A has full column rank rank(A) = $2n$), we need to inject the following additional constraints.

4.2. Quadruplet equidistance constraints

While only the basic geometric constraints are absolutely necessary, we can "spend" the remaining degrees of freedom to constrain the tiling in a way that will minimize the complexity of the crease pattern for the sake of human foldability. A good heuristic is to minimize the number of internal vertices generated by the straight skeletons of the axial polygons.

To physically fold these polygons flat along the axis, their interior **ridge creases** must be constructed. While Lang uses a universal molecule [1] to match the resulting crease pattern to the input tree, we generate crease patterns without respect to any predetermined tree. Therefore, we can utilize the standard straight skeleton [9] as the simplest valid construction of ridge creases.

Internal vertices in a straight skeleton appear where 3 or more edges are equidistant from a common point. We can "collapse" internal vertices together by forcing > 3 sides to be equidistant from a common point. Therefore, we define a **quadruplet** as a set of 4 edges (not necessarily contiguous) on a polygon, which we can constrain to be equidistant from a shared incircle.

Let $\hat{n}_i$ be the inward-facing unit normal vector of edge i and let $\vec{v}_i$ be a point on edge i. Let $\vec{p} = (p_x, p_y)$ and r be the unknown position and radius of the shared incircle.

For the incircle to be tangent to all four edges simultaneously, the perpendicular distance from the center $\vec{p}$ to each edge i must be exactly equal to r. Using vector projection, we can express the distance to a line as:

$$\hat{n}_i \cdot \vec{p} - r = \hat{n}_i \cdot \vec{v}_i \quad \text{for } i \in \{1, 2, 3, 4\} \tag{12}$$

This gives us a system of 4 linear equations. We can represent this system in matrix form as $Q\vec{\xi} = \vec{w}$, where $\vec{\xi}$ contains our unknowns:

$$\underbrace{\begin{bmatrix} n_{1x} & n_{1y} & -1 \\ n_{2x} & n_{2y} & -1 \\ n_{3x} & n_{3y} & -1 \\ n_{4x} & n_{4y} & -1 \end{bmatrix}}_{Q} \underbrace{\begin{bmatrix} p_x \\ p_y \\ r \end{bmatrix}}_{\vec{\xi}} = \underbrace{\begin{bmatrix} \hat{n}_1 \cdot \vec{v}_1 \\ \hat{n}_2 \cdot \vec{v}_2 \\ \hat{n}_3 \cdot \vec{v}_3 \\ \hat{n}_4 \cdot \vec{v}_4 \end{bmatrix}}_{\vec{w}} \tag{13}$$

Notice that Q is a 4×3 matrix. We have 4 equations but only 3 unknowns (p_x, p_y, and r). For an overdetermined system to have a valid solution, the right-hand side vector $\vec{w}$ must lie within the column space of Q. We do not actually need to solve for $\vec{\xi}$ as long as some solution exists. If we can eliminate $\vec{\xi}$ entirely, we are left with a relationship that the vertices $\vec{v}_i$ must satisfy. Recall that a vector $\vec{w}$ is in the column space of Q if and only if it is orthogonal to the left null space of Q. Let $\vec{q}^T = \begin{bmatrix} q_1 & q_2 & q_3 & q_4 \end{bmatrix}$ be a basis

vector for the left null space of Q, such that:

$$\vec{q}^T Q = 0 \tag{14}$$

Because the normals $\hat{n}_i$ are strictly fixed constants, the matrix Q is constant. Therefore, $\vec{q}$ is also constant and can be computed in terms of the normals.

For the incircle to exist, we must satisfy:

$$\vec{q}^T Q \vec{\xi} = \vec{q}^T \vec{w} \tag{15}$$

$$0 = \sum_{i=1}^{4} q_i \hat{n}_i \cdot \vec{v}_i \tag{16}$$

$$0 = \sum_{i=1}^{4} (q_i n_{ix}) x_i + (q_i n_{iy}) y_i \tag{17}$$

Note that this is a strictly linear, homogeneous constraint acting entirely on the vertex positions x_i and y_i, which can be expressed as an additional row in the constraint matrix A without ever solving for the position or radius of the incircle.

In general, constraining an unconstrained quadruplet (that contains at least one unconstrained edge) will remove one degree of freedom. However, it is possible for a quadruplet constraint to be satisfied without actually reducing the number of vertices in the straight skeleton. This is because a vertex appears in the straight skeleton if and only if it is strictly closer to its constituent edges than to any other edge in the polygon. If another edge "invades" the incircle, the quadruplet constraint is effectively useless. Even worse, it is possible for a quadruplet constraint to cause an edge to collapse to 0 or negative length, violating the planarity of the tiling. Or, the mouth of a concave polygon might "clip" through the other end, forming an invalid "bowtie" shape. All of these effects are defined by inequalities. Therefore, we must resort to a Mixed-Integer Linear Programming (MILP) formulation to strategically select the optimal set of quadruplets that maximize the number of collapsed straight skeleton vertices while ensuring the physical foldability of the tiling.

4.3. Optimal Quadruplet Selection

To evaluate the physical consequences of activating potential quadruplet constraints, the solver must be able to solve the linear system from Equation (11) for any possible combination of quadruplets, and evaluate the validity of the resulting geometry. Let the state vector $\vec{X}$ be an augmented vector of the vertex coordinate vector $\vec{x}$:

$$\vec{X} = \left[\vec{x},\ \vec{z}_{quads},\ \vec{p}_{incircles},\ \vec{r}_{incircles},\ \vec{b}_{bowtie} \right]^T \tag{18}$$

Where we concatenate the following vectors:

- $\vec{x}$: A concatenation of continuous 2D coordinates $\vec{v}_i = (x_i, y_i)$ for every vertex in the graph
- $\vec{z}_{quads}$: Boolean decision variables $z_k \in \{0, 1\}$ denoting whether quadruplet candidate k is constrained.
- $\vec{p}_{incircles}$: Continuous 2D coordinates $\vec{p}_k = (p_{k_x}, p_{k_y})$ representing the prospective incircle center for quadruplet candidate k.
- $\vec{r}_{incircles}$: Continuous variables $r_k \geq 0$ representing the prospective incircle radius for quadruplet candidate k.
- $\vec{b}_{bowtie}$: Boolean variables used to check topological validity of concave polygons.

Now recall that the standard form of a MILP problem is defined as [20]:

$$\max_{\vec{X}} \quad \vec{c}^T \vec{X} \tag{19}$$

$$\text{subject to} \quad M\vec{X} \leq \vec{B} \tag{20}$$

where $\vec{c}$ is the objective function as reward weights on the elements of $\vec{X}$, M is the constraint matrix that encodes all of the linear equalities and inequalities, and $\vec{B}$ stores the constants on the right-hand side of each constraint.

Our objective is to maximize the number of quadruplets that can be activated without violating geometric constraints. Therefore, $\vec{c}$ contains non-zero weights only for the decision variables $\vec{z}_{quads}$. We heavily reward resolving concave (reflex) vertices because they create disproportionately messier crease patterns if not collapsed into a clean internal vertex. Accordingly, quadruplets involving reflex vertices are assigned a priority weight ($c_k = 10.0$), while standard convex quadruplets receive a base weight ($c_k = 1.0$)[2].

4.3.1. Formulating the Constraint Rows of M

1. Basic geometric constraints First, the same basic geometric constraints described in Section 4.1 are applied to the vertex coordinates. These constraints only depend on the vertex coordinates $\vec{x}$ and are the only equality constraints in this problem formulation.

2. Edge Integrity: To prevent edges from collapsing to zero or negative length, we enforce a minimum feature size, $\epsilon > 0$. Let $\hat{t}$ be the unit tangent vector of an edge. For any directed edge from vertex $\vec{v}_1$ to vertex $\vec{v}_2$, the projected distance must remain at least ϵ:

$$\hat{t} \cdot (\vec{v}_2 - \vec{v}_1) \geq \epsilon \quad \Longrightarrow \quad -\hat{t} \cdot \vec{v}_2 + \hat{t} \cdot \vec{v}_1 \leq -\epsilon \tag{21}$$

Note that ϵ is not for numerical stability, but another metric for human foldability. If ϵ is too small, the crease pattern may result in tiny creases that are difficult to fold. If ϵ is too large, the solver may not be able to find any valid solutions. We empirically set $\epsilon = 1/4N$, where N is the grid size of the original topology.[3]

[2] The weights are tuned empirically to act as a lexicographical prioritization scheme, where the solver always prioritizes resolving reflex vertices first, and only then maximizes the total number of convex quadruplets. The solver is not sensitive to the exact values of these weights.

[3] This corresponds to a feature size of ≈1cm on an ordinary 15cm square of origami paper when $N \leq 5$.

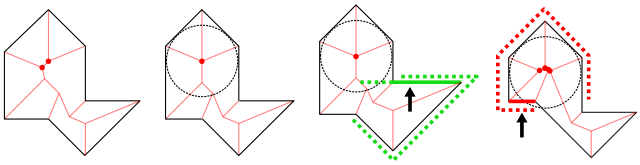


Figure 6: Incircle invasions will break the straight skeleton. From left to right: (1) An unconstrained polygon and its messy straight skeleton. (2) Applying an incircle constraint to a quadruplet collapses two interior points into one. (3) An edge from a different convex chain (separated by two concave vertices) will never break the straight skeleton even if its projection invades the incircle. (4) If an edge from the same convex chain invades the incircle, the constraint is useless and the straight skeleton is broken.

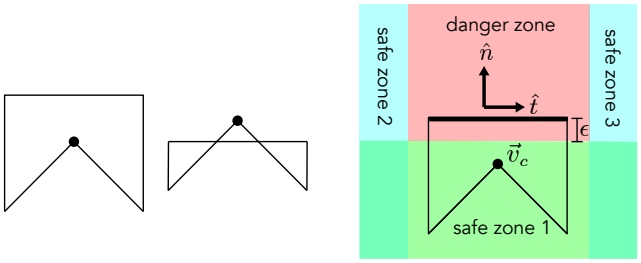


Figure 7: Anti-bowtie clearance. Left: an unconstrained polygon. Center: the polygon with a "bowtie" violation. Right: the three safe zones defined by b_1, b_2, b_3 and the danger zone (red) where the constraint is violated, defined by the edge's vectors $\hat{n}, \hat{t}$ and minimum clearance size ϵ.

3. Quadruplet Activation (Big-M Method): If $z_k = 1$, quadruplet k must be constrained such that the distance from the incircle center $\vec{p}_k$ to the boundary edge originating at vertex $\vec{v}$ with inward normal $\hat{n}$ must exactly equal the radius r_k. However, if $z_k = 0$, the constraint must be relaxed. To encode this conditional logic as linear rows in M, we use the standard Big-M method [20].We choose a large constant $C \approx 3N$ based on the largest possible feature size $N\sqrt{2}$ with a reasonable safety factor and apply the following inequalities to a vertex $\vec{v}$ on each edge in the quadruplet:

$$\hat{n} \cdot \vec{p}_k - \hat{n} \cdot \vec{v} - r_k + z_k C \leq C \tag{22}$$

$$-\hat{n} \cdot \vec{p}_k + \hat{n} \cdot \vec{v} + r_k + z_k C \leq C \tag{23}$$

4. Incircle invasion: Recall that a vertex only appears in the straight skeleton if and only if it is strictly closer to its constituent edges than to any other edge in the polygon, so if another edge "invades" the incircle, the quadruplet constraint is effectively useless. To prevent this, we must ensure that any invader edge i maintains a normal projection distance from the incircle center $\vec{p}_k$ that is at least r_k:

$$-\hat{n}_i \cdot \vec{p}_k + \hat{n}_i \cdot \vec{v}_i + r_k + C z_k \leq C \tag{24}$$

However, this equation treats every side of the polygon as an infinite line, while incircle invasion only occurs if the actual line segment invades the circle. Formulating this stronger constraint would introduce a quadratic term. Instead, we realize that edges that do not share a convex chain with any edge in the quadruplet are incapable of actually invading the incircle, so as a conservative heuristic, only apply the above constraint to edges that do share a convex chain with the quadruplet. This is illustrated in Figure 6.

5. Bowtie Clearance: In some concave polygons, it is possible for a concave "mouth" vertex to clip through an edge on the opposite side of the polygon, as illustrated in Figure 7. To prevent this, we can enforce that every reflex vertex must reside in at least one of the three "safe zones" defined by the edge's normal and tangent vectors.

For every reflex vertex $\vec{v}_c$ and non-adjacent edge $(\vec{v}_1, \vec{v}_2)$ with normal $\hat{n}$ and tangent $\hat{t}$, we create three Boolean variables based on the vertex's position relative to the edge:

- $b_1 = 1$ if $\hat{n} \cdot (\vec{v}_c - \vec{v}_1) < 0$ (safe zone 1: the vertex is behind the edge)
- $b_2 = 1$ if $\hat{t} \cdot (\vec{v}_c - \vec{v}_1) > 0$ (safe zone 2: the vertex is to the left of the edge)
- $b_3 = 1$ if $\hat{t} \cdot (\vec{v}_c - \vec{v}_1) < 0$ (safe zone 3: the vertex is to the right of the edge)

We can enforce that the reflex vertex must reside in at least one safe sector by applying the constraint:

$$b_1 + b_2 + b_3 \geq 1$$

The MILP solver gets to choose which of b_1, b_2, b_3 is true. Similar to the quadruplet toggles z_k, we can use big-M constraints to link these Boolean variables to toggle which which safe zone the vertex is constrained to. For example, the corresponding constraint for b_1 would be:

$$\hat{n} \cdot (\vec{v}_c - \vec{v}_1) \geq \epsilon - C(1 - b_1) \tag{25}$$

$$-\hat{n} \cdot \vec{v}_c + \hat{n} \cdot \vec{v}_1 + C b_1 \leq C - \epsilon \tag{26}$$

In total, four inequality constraints are added to the system for every pair of reflex vertex and non-adjacent edge.

4.3.2. The Compound Matrix Inequality

By concatenating these varied constraints, we construct the master MILP matrix equation. The inequality system $M\vec{X} \leq \vec{B}$ takes the following block matrix structure[4] shown in Equation 27, where:

- M_{edges} applies the tangent vector projections ($\hat{t}^T$) mapping adjacent vertices to their required edge clearance.
- N_{quad}, $N_{invader}$, and N_{bowtie} apply the corresponding inward normal and tangent projections ($\hat{n}^T, \hat{t}^T$) mapping vertices to their respective distance inequalities.
- $N_{incircle}$ projects the prospective incircle centers ($\vec{P}_k$) against those same normal vectors.

[4]For the sake of visual compactness, we include the basic equality constraints A as part of M rather than as a separate matrix. In practice, all constraints are handed off to an open-source CBC solver [21] via the PuLP interface [22].

$$
\begin{array}{l}
\text{Basic constraints} \\
\text{Edge Integrity} \\
\text{Quadruplets} \\
\\
\text{Incircle Clearance} \\
\\
\text{Anti-bowtie}
\end{array}
\underbrace{\left[\begin{array}{ccccc}
A & 0 & 0 & 0 & 0 \\
\hdashline
M_{edges} & 0 & 0 & 0 & 0 \\
\hdashline
-N_{quad} & C \cdot I_{quad} & N_{incircle} & -I_{quad} & 0 \\
N_{quad} & C \cdot I_{quad} & -N_{incircle} & I_{quad} & 0 \\
\hdashline
N_{invader} & C \cdot I_{inv} & -N_{incircle} & -I_{inv} & 0 \\
\hdashline
0 & 0 & 0 & 0 & -S_{sector} \\
\hdashline
N_{bowtie} & 0 & 0 & 0 & C \cdot I_{bowtie}
\end{array}\right]}_{M}
\underbrace{\left[\begin{array}{c}
\vec{x} \\
\vec{z}_{quads} \\
\vec{P}_{incircles} \\
\vec{r}_{incircles} \\
\vec{b}_{bowtie}
\end{array}\right]}_{\vec{X}}
\leq
\underbrace{\left[\begin{array}{c}
\mathbf{0} \\
-\epsilon \\
\hdashline
\mathbf{C} \\
\mathbf{C} \\
\mathbf{C} \\
\hdashline
-\mathbf{1} \\
\mathbf{C} - \epsilon
\end{array}\right]}_{\vec{B}}
\tag{27}
$$

- S_{sector} is a block summation matrix that groups and adds the three Boolean sector variables ($b_1 + b_2 + b_3$) for each reflex-edge pair.
- I represents identity matrices of the appropriate dimensions mapping the scalar variables ($\vec{r}_{incircles}$ and the Boolean toggles).
- $\mathbf{C}$, ϵ, and $\mathbf{1}$ represent column vectors populated dynamically with the respective constants.

In total, M and $\vec{B}$ are composed of pre-computable constants for each topology. The problem in this form can be passed into an open-source MILP solver [21, 22] to constrain the maximum number of quadruplets while maintaining the geometric validity of the tiling.

4.3.3. Diversity Threshold for Multiple Solutions

A single topology can often be tiled in several geometrically distinct ways. To populate the database with diverse layouts from the same root topology, we loop the MILP solver. Upon finding an optimal set of active quadruplets $\mathcal{A}$, we log the configuration and inject a strict integer cut back into the solver:

$$\sum_{i \in \mathcal{A}} z_i \leq |\mathcal{A}| - d_{threshold} \tag{28}$$

where $d_{threshold}$ is a hyperparameter defining the minimum diversity difference between solutions. This forces the MILP to swap at least $d_{threshold}$ quadruplets from the previous solution, nudging it to discover alternative geometric variations. However, it is not guaranteed that the resulting tilings will be different, even if the selected sets of quadruplets are different.

4.3.4. Quadruplet Harvesting

The solution to the MILP comes with a set of vertex positions that is guaranteed to be physically realizable (satisfying the constraints applied by M). However, it cannot guarantee that the tiling is fully locked. This is because it is computationally prohibitive to feed every possible quadruplet into the MILP (consider the fact that an n-gon contains $\binom{n}{4}$ possible quadruplets), so we initialize it with a heuristic "harvest" of promising quadruplets. This includes quadruplets that are either contiguous or are the nearest edges to a concave pair of edges. As a result, the MILP might not have had access to a key quadruplet needed to fully constrain the tiling.

We can quickly check the rank of A with the current selection of quadruplets by performing Gaussian elimination over a finite field $\mathbb{Z}_p$ with a large prime [23]. If the matrix is rank-deficient, the $\mathbb{Z}_p$ elimination identifies exactly which columns (variables) lack pivots. We map these free variables back to their physical vertices. The missing quadruplets will likely involve these unconstrained vertices. We then initialize an exhaustive search of unused quadruplet candidates, specifically filtering for quadruplets that involve the identified sliding vertices. It tentatively applies each quadruplet constraint and keeps ones that increase the rank until full rank is reached[5]. Only then are the constraints actually applied and solved for the exact tiling geometry.

4.4. Rational 4D basis

Before solving the linear system to compute the final tiling geometry, we must address the numerical representation of vertices in the 22.5° system. In conventional crease pattern software, points are represented as Cartesian coordinates stored as pairs of floating-point numbers. However, 22.5° triangles inherently generate lengths in $\mathbb{Q}(\sqrt{2})$. Relying on floating-point arithmetic for these irrational intersections will inevitably cause precision drift and numerical instability.

We first observe that all Cartesian coordinates in the 22.5° angle system can be represented in the form $(R_x + S_x\sqrt{2}, R_y + S_y\sqrt{2})$, where R_x, S_x, R_y, S_y are rational numbers [2]. Therefore points can be represented as a tuple of four fractions, each of which can be stored as a pair of integers for exact rational arithmetic.

To map these 4D coordinates to the 2D physical plane, we project the four basis vectors such that they are uniform in length and spaced 45° apart. This mapping is geometrically identical to the standard 2D projection of a 4D hypercube, as illustrated in Figure 8, and was proposed for modeling 22.5° crease patterns by Lang et al. [24].

Let the vertex $\vec{v}_i = (x_i, y_i)$ be represented in 4D as $\vec{v}^{(4D)} = (a_i, b_i, c_i, d_i)$. The projection matrix mapping this state back to 2D Cartesian coordinates is defined as:

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} = \begin{bmatrix} 1 & \frac{\sqrt{2}}{2} & 0 & -\frac{\sqrt{2}}{2} \\ 0 & \frac{\sqrt{2}}{2} & 1 & \frac{\sqrt{2}}{2} \end{bmatrix} \begin{bmatrix} a_i \\ b_i \\ c_i \\ d_i \end{bmatrix} \tag{29}$$

[5] On the rare occasion that exhausting quadruplets is still insufficient, the topology is discarded. Refer to Table 1 for error rates and MILP solver timeout rates.

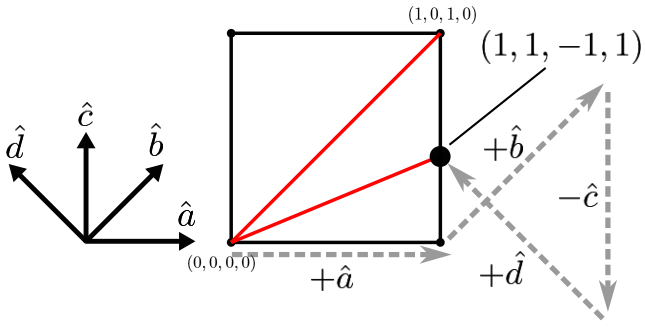


Figure 8: An example vertex of a simple crease pattern expressed in the hypercube 4D basis projection. The vertex $(1, 1, -1, 1)$ can be understood intuitively as the sum of the basis vectors $\hat{a}+\hat{b}-\hat{c}+\hat{d}$. Unlike its 2d counterpart $(1, \sqrt{2}-1)$, this representation is fully rational.

As established in this unpublished work [24], these rational 4D coordinates project uniquely to the 2D plane, and crucially, 2D 22.5° vertices "lift" bijectively back into 4D. By utilizing this hypercube projection, we can perform all spatial reasoning in a strictly rational domain.

4.5. Exact Precision Linear Solve

During the MILP solve, the tiling constraints were formulated using floating-point arithmetic for fast linear algebra. This is sufficient for quickly checking that potential quadruplets do not cause degeneracy. However, the final tiling must be solved with rational arithmetic to guarantee exact precision downstream. We now transition the system $A\vec{x} = \vec{b}$ (Sections 4.1 and 4.2) into the 4D hypercube basis (Section 4.4).[6]

For example, recall the angular constraint from Equation (7), requiring the edge between vertices $\vec{v}_i$ and $\vec{v}_j$ to remain orthogonal to the edge's fixed normal vector $\hat{n} = [n_x, n_y]^T$:

$$n_x(x_j - x_i) + n_y(y_j - y_i) = 0 \tag{30}$$

We use the projection matrix from Equation (29) to substitute the 2D coordinates (x, y) with their 4D rational representations (a, b, c, d). Let us decompose the coordinate differences into rational and irrational parts as R and S respectively:

$$x_j - x_i := R_x + S_x\sqrt{2} \tag{31}$$

$$= (a_j - a_i) + \frac{1}{2}\big((b_j - b_i) - (d_j - d_i)\big)\sqrt{2} \tag{32}$$

$$y_j - y_i := R_y + S_y\sqrt{2} \tag{33}$$

$$= (c_j - c_i) + \frac{1}{2}\big((b_j - b_i) + (d_j - d_i)\big)\sqrt{2} \tag{34}$$

Similarly, because the normal vector $\hat{n}$ is derived from 22.5° geometry, its components can also be decomposed into rational and irrational parts. Let us define these components of n_x, n_y:

$$n_x := r_x + s_x\sqrt{2} \tag{35}$$

$$n_y := r_y + s_y\sqrt{2} \tag{36}$$

Substituting these components into the original equation (7) yields a polynomial expansion:

$$(r_x+s_x\sqrt{2})(R_x+S_x\sqrt{2})+(r_y+s_y\sqrt{2})(R_y+S_y\sqrt{2}) = 0 \tag{37}$$

which can be separated into rational and irrational terms:

$$C_{rational} + C_{irrational}\sqrt{2} = 0 \tag{38}$$

For the entire expression to be exactly zero, both coefficients $C_{rational}$ and $C_{irrational}$ must independently evaluate to zero. This transforms our single 2D continuous constraint into two independent linear equations in terms of the constant edge normal and the rational components of the vertex coordinates:

$$C_{rational} = R_x r_x + 2S_x s_x + R_y r_y + 2S_y s_y \quad = 0 \tag{39}$$

$$C_{irrational} = R_x s_x + S_x r_x + R_y s_y + S_y r_y \quad = 0 \tag{40}$$

By applying this transformation to every row in the system, we can convert the entire problem into a rational linear system with twice as many rows and twice as many unknowns (four unknowns per vertex). This overdetermined system is then solved using Gauss-Jordan elimination. The solver operates over a custom C++ `Fraction` class that normalizes the denominators after each arithmetic step. Because no realistic crease pattern will ever require a denominator larger than 2^{63}, native 64-bit integers can be used instead of arbitrary-precision integers.

5. Database Generation Phase III: Crease Patterns and Trees

The exact tiling defines only the axial creases of the crease pattern. Next, the ridge creases must be constructed by filling in the axial polygons with straight skeletons. The crease pattern then needs to satisfy Kawasaki flat foldability, which we will achieve by introducing hinge creases. We neglect mountain-valley assignment and layer self-intersection at this stage. Finally, the folded state of the crease pattern must be computed in order to extract its tree structure. For an illustration of the relationship between axial, ridge, and hinge creases, refer to Figure 9.[7]

[6] Unlike Sections 4.1 and 4.2, the system from this point onward defines the square as a normalized $[0, 1]^2$ rather than $[0, N]^2$.

[7] The crease patterns in this paper are colored by crease type, not mountain-valley assignment. However, we use a similar color scheme because axial creases are always valley (blue, by convention) and ridge creases are mostly mountain (red, by convention).

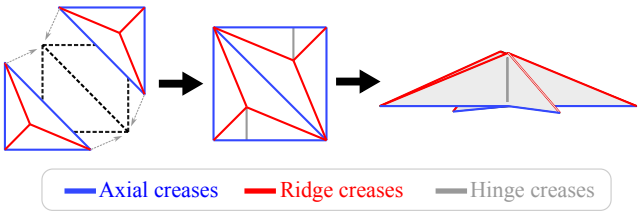


Figure 9: A dissection of a simple crease pattern. Left: the axial tiling (black) consists of two triangles, each of which is composed of a straight skeleton "tile". Center: crease pattern, made Kawasaki foldable by the addition of two hinge creases. Right: the resulting folded form. The axial creases (blue) lie collinear along the common axis, the hinge creases (grey) are perpendicular to the axis, and the ridge creases (red) are at 22.5° angles from the axis.

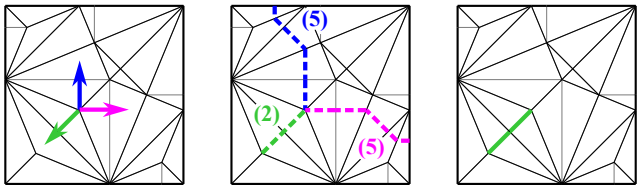


Figure 10: The greedy procedure to enforce flat-foldability with the least number of new creases introduced. Creases are left uncolored for visual clarity. Left: a degree-3 vertex has 3 new crease directions that could make it flat foldable. Middle: each direction is tentatively raycasted outwards, reflecting over creases until it collides with the paper border or another vertex. Right: the direction that created the fewest new creases (2) is committed.

5.1. Straight Skeleton Generation

The interior creases of each polygon primarily consist of the polygon's straight skeleton [9], which in origami, is what places the axial creases (the border of the polygon) onto a collinear axis [25]. Rather than implementing a custom straight skeleton algorithm that operates directly on the exact rational coordinates established in Section 4.5, we take a hybrid approach to leverage existing libraries built on floating-point arithmetic [26] while still ensuring the final output is perfectly rational.

First, we project the exact 4D vertices of a face into 2D Cartesian floats. However, standard geometric algorithms—such as the sweep-line methods commonly used for straight skeletons—are notoriously fragile when handling floating-point arithmetic. Because these algorithms rely on strict spatial sorting, tiny imprecisions can cause the algorithm to make inconsistent topological decisions, leading to program failures [27].

To bypass these well-documented robustness issues, we quantize the floats by truncating to 3 decimal places. Because the MILP constraint in Equation (21) guarantees a minimum feature size of $\epsilon = 1/4N \gg 10^{-3}$, this forces near-degenerate points to snap to a rigid grid without risking topological collapse.

We feed this sanitized (but no longer accurate) floating-point polygon into an open-source library [26] to extract the topological connectivity of the interior creases. To reclaim the exact precision of these internal vertices, we begin with the polygon boundary whose exact 4D coordinates are solved in Section 4.5. For any interior vertex connected to at least 2 boundary vertices, we can snap those two interior edges to their nearest 22.5° angle and compute the intersection of creases using Equation (50) to find the exact 4D coordinates of this vertex, which can now be used to solve for any other interior vertices connected to it. By recursively traversing inwards, we can recover the exact 4D coordinates of every interior vertex in the straight skeleton.

5.2. Kawasaki Flat Foldability

The resulting straight skeletons are generally not flat-foldable on their own. Kawasaki's Theorem states that for an interior vertex to fold flat, the alternating sum of sector angles between incident creases must equal 180° [4]. Resolving even-degree vertices that violate Kawasaki's theorem necessitates a minimum of two simultaneous crease modifications, introducing a branching factor of up to $\binom{8}{2}$ potential direction pairs. In contrast, any odd-degree interior vertex can be resolved by the addition or removal of a single crease, of which there are only 8 possible directions.[8]

We employ a greedy Kawasaki-fixing algorithm that resolves odd-degree vertices first. For a target vertex, the algorithm checks each of the 8 directions for a potential route that would satisfy Kawasaki's theorem if a hinge crease were added. For each valid direction, the algorithm raycasts a tentative crease from the vertex. If this tentative crease collides with an existing crease, it terminates and forms a degree-3 vertex. To satisfy Kawasaki's theorem at this new vertex, the tentative crease is reflected over the existing crease and propagated further. The process repeated recursively until the tentative crease fully terminates at the paper boundary or another vertex. To minimize the structural complexity of the model, the algorithm commits the hinge direction that satisfies Kawasaki's theorem while introducing the fewest number of new creases. This procedure is illustrated in Figure 10. The algorithm then iterates through the remaining vertices until all vertices satisfy Kawasaki's theorem.

Due to the greedy nature of this heuristic, there are occasional edge cases where the algorithm may fail to converge (refer to Table 1). In these instances, the crease pattern is simply discarded from the database. Because the objective of this Phase is to populate a sufficiently large database rather than exhaustively enumerate every theoretically possible crease pattern, it is acceptable to discard the occasional edge case.

5.3. The Stacked Folded State Architecture

To calculate the 2D folded coordinates, we construct a spanning tree of the faces[9] of the crease pattern via a

[8] Recall that hinge creases are by definition perpendicular to axial creases, which are restricted to multiples of 45°.

[9] The crease pattern, which is a planar graph, explicitly tracks topological faces using a standard Doubly-Connected Edge List (DCEL) algorithm [19], sorting half-edges counter-clockwise and discarding the infinite outer face using signed-area calculations.

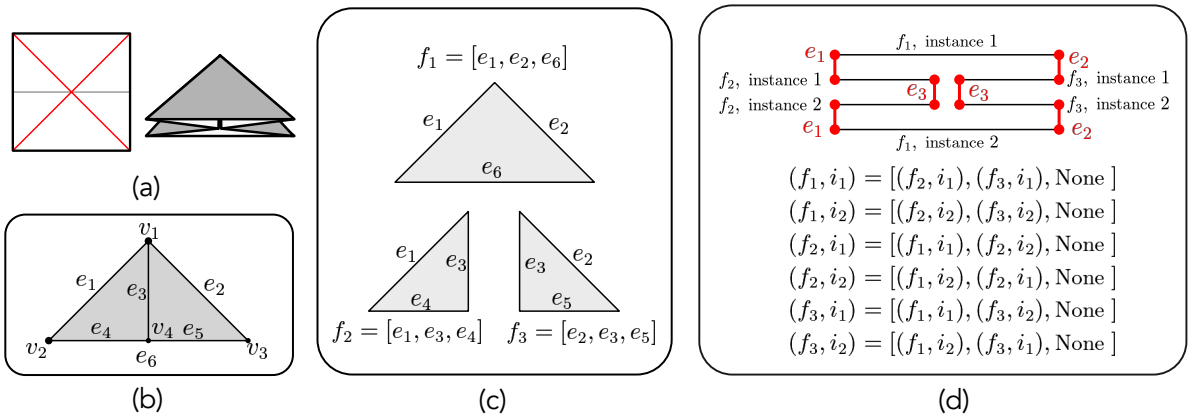


Figure 11: A visualization of the folded state data structure for a waterbomb base. (a) The unfolded crease pattern and folded state. (b) The coordinate geometry defining unique edges and vertices of the folded state. (d) The three unique faces defined by edges. (e) The instances of those faces and their connectivity. In this specific case, each face has 2 instances, and all instances have a connection to the paper border `[None]`.

Kinematics Breadth-First Search (BFS). Starting from an arbitrary root face, we span the adjacency graph, recursively reflecting the exact 4D vertices of child faces across their connecting hinge edges [28]. The reflection operator is closed within the 4D rational basis because all creases angles are at multiples of 22.5°.

To keep track of connectivity (i.e., two faces connected across a crease) in the folded configuration, we define a specialized data structure (illustrated in Figure 11) that separates the planar geometry from the physical paper layers (*instances*):

- `vertices`: A list of unique exact 4D coordinates present in the folded state.
- `edges`: A list of unique line segments, defined as pairs of vertex indices.
- `faces`: A list of unique face polygons, defined as an undirected cycle of edge indices.
- `instances`: The physical paper layers. Each unique topological face f corresponds to a stack of one or more instances. An instance is represented as a list of connection pointers, where the k-th pointer corresponds to instance connected across the k-th edge of face f. Each pointer stores the face and instance index (f_{other}, i_{other}) of the neighboring instance across that specific edge, or `None` if the edge is a paper boundary.

The data structure does not track mountain-valley assignment or Z-ordering of the layer instances.

5.4. Orthogonal Slicing and Tree Extraction

With the paper successfully folded into a layered stack of 2D polygons, we must extract its uniaxial stick figure tree structure. We execute an orthogonal slicing algorithm to reveal its cross-sectional connectivity. This algorithm is illustrated in Figure 12.

Recall that all axial creases by definition lie along a common axis when the model is folded. Let $\vec{a}$ be this axis. We project all vertices of the folded state onto $\vec{a}$ to identify potential hinge locations along the axis of the model.

We mathematically sweep a 1D slice line $\mathcal{L}(h)$ perpendicular to $\vec{a}$, executing a slice at every potential hinge location. When a slice intersects the folded state, it splits a single face into two faces joined at an unfolded crease. The topological bifurcation procedure performs the following operations:

1. **Edge Splitting:** Intersected edges are split at the intersection point, registering new sub-segments into a global edge map to prevent duplicates.
2. **Face Bifurcation:** The vertex loop of each face is analyzed. If a face contains vertices strictly on both sides of the slice line, it is bifurcated into a left chain and a right chain. A new internal crease edge is generated to seal the two new topological faces.
3. **Pointer Resolution:** The engine duplicates the old physical instances into the new bifurcated faces. It maps the newly created internal creases to point to each other, seamlessly sewing the severed halves together. For external connections, it checks the split geometry of the neighboring instance to ensure the pointer points to the correct geometric half of the neighbor.

With the model completely sliced at every critical hinge location, we extract the tree by analyzing the connectivity of the face instances. We rebuild an adjacency graph of all instances, except we explicitly ignore connections across hinge-parallel (that is, axis-perpendicular) edges, such as the newly sliced non-crease edges. Conceptually, these orthogonal slices act as physical cuts that sever the paper into disconnected components, each of which corresponds to a distinct edge of the tree structure.

The topological tree is then constructed by analyzing how these components interact at the hinge locations:

1. **Edges:** Every connected component spans between a minimum and maximum hinge value (h_{start} and h_{end}).

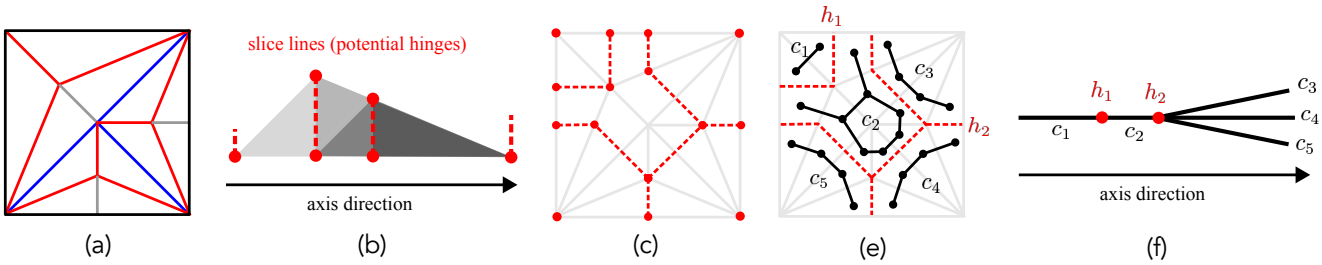


Figure 12: The tree extraction slicing process. (a) The crease pattern. (b) The folded state with axis and orthogonal slices through vertex projections. (c) The sliced crease pattern (i.e., the packing). (d) Disconnected components of the crease pattern as a result of slicing. (e) The final extracted tree based on the connectivity between hinge sets and the face components. Not shown: degree-2 vertices are removed from the tree.

Each component forms an **Edge** in the tree structure (not to be confused with the edges of the folded state or the crease pattern) with its metric weight derived from the axial length $h_{end} - h_{start}$.

2. **Nodes:** At each hinge location h, we must determine how the incident components converge to form structural nodes. We construct a local adjacency graph: two components are connected if and only if they share a physical paper connection across a hinge-parallel edge located at h. Each connected cluster within this local graph becomes a distinct **Node** in the tree. This rigorous clustering ensures that independent branches overlapping in the 2D projection remain topologically separate.
3. **Simplification:** Finally, the raw tree is simplified by collapsing and merging intermediate degree-2 nodes.

This distills the low-level complexity of the sliced layers into an abstracted tree representing the structural appendages (i.e., flaps) of the origami model. This tree has defined edge lengths but undefined node positions. This crease pattern → tree pipeline is the inverse of Lang's TreeMaker algorithm [1].

6. Spectral Shape Embedding and Indexing

With the offline precomputation phase having successfully generated and sliced millions of valid 22.5° crease patterns, the final challenge is rendering this massive dataset searchable. When a user draws a target uniaxial tree graph at runtime, the system must instantaneously retrieve the crease pattern with the closest matching tree structure.

Conventional CAD database retrieval often relies on exact topological graph matching or subgraph isomorphism algorithms [15]. However, in the context of origami crease patterns, exact topological matching is inherently brittle. Both the geometric extraction process described in Section 5 as well as user-drawn trees frequently yield topological noise, such as structurally irrelevant degree-2 nodes or negligible "stub" appendages.

Strict isomorphism checks would reject highly similar structural bases due to these trivial discrepancies. Instead, we require a continuous mathematical descriptor where metric and topological similarities translate smoothly into Euclidean distance within a vector space. We achieve this through a spectral graph embedding of the extracted tree structure for each crease pattern.

In practice, origami models exhibit significant morphological plasticity; the final tree structure of a model is influenced but not strictly fixed by the base crease pattern, but changes as shaping creases are introduced. For instance, the traditional crane and traditional dragon originate from an identical base crease pattern——forming two wings, a head, and a tail——yet manifest divergent tree structures post-shaping, as the dragon's crease pattern produces an additional four small appendages (Figure 13).

This observation demonstrates that our retrieval system need not satisfy a rigid isomorphic match to be effective. Instead, spectral embeddings provide a robust, fault-tolerant metric where approximately matching trees can function as relevant design starting points, accommodating the inherently fluid nature of the human creative process.

6.1. Discretization and the Graph Laplacian Eigenvalues

Let the tree structure extracted from a crease pattern be defined as an undirected metric graph, where each edge possesses a physical length corresponding to the axial length of the folded flap. Prior to embedding, we normalize all edge lengths by the total sum of the tree's edge lengths for scale invariance. Additionally, to approximate smooth functions over the graph, we must mitigate the discretization error caused by long, undivided edges. We apply a uniform mesh resampling step, subdividing edges such that no single segment exceeds a maximum normalized length resolution (e.g., 0.02). This produces a finely discretized metric tree G.

We then construct the discrete Laplace-Beltrami operator for G. Even with discretization, we weight differences in edge element lengths with the generalized eigenvalue problem:

$$L\mathbf{x} = \lambda M\mathbf{x} \tag{41}$$

where L is the standard stiffness matrix (with edge conductance defined as $1/l_e$ where l_e is the normalized edge length) and M is the lumped diagonal mass matrix (where

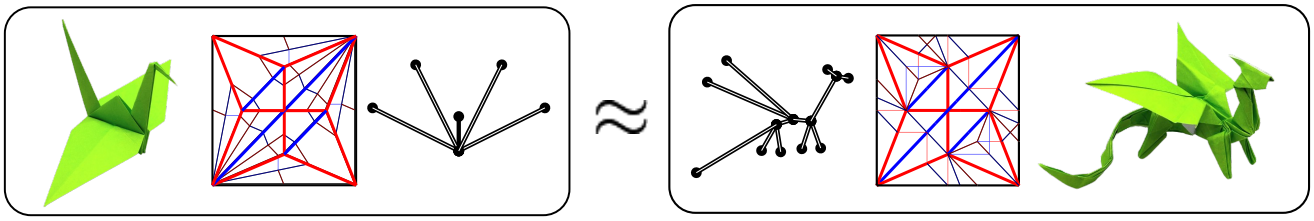


Figure 13: Comparison of the traditional origami crane (left) and dragon (right), each containing the end tree structure, crease pattern, and physical model. The underlying base crease patterns (bolded creases) are identical, while the divergent shaping creases (faded creases) reflect the refinement process. Despite significant differences in the final tree structures—such as the dragon's additional leg appendages—the shared base demonstrates that even distantly related spectral embeddings can be sufficient for meaningful structural retrieval.

$M_{i,i}$ is half the sum of incident segment lengths) [29, 30]. This formulation is equivalent to the Finite Element Method (FEM) discretization of the 1D Laplace operator. Solving this system yields a set of non-negative eigenvalues $0 = \lambda_0 \leq \lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_k$. Because the tree is a single connected component, the first eigenvalue λ_0 is trivially zero and is discarded.

6.2. The Cumulative Wave Kernel Signature (CWKS)

To construct a permutation-invariant feature vector from the raw eigenvalues, we utilize the Wave Kernel Signature (WKS) [16]. We evaluate the signature across a set of D logarithmically spaced energy bands $e_j \in [e_{\min}, e_{\max}]$. The raw signature profile takes the form:

$$W_j = \sum_{i=1}^{k} \exp\left(-\frac{(e_j - \log \lambda_i)^2}{2\sigma^2}\right) \tag{42}$$

While the standard WKS provides a robust baseline Probability Density Function (PDF), evaluating structural similarity using the L_2 distance between two WKS profiles can still be overly rigid. If a slight geometric perturbation shifts a tree's eigenvalue from one energy band e_j into an adjacent band e_{j+1}, the raw WKS heavily penalizes this shift as two orthogonal errors (a missing peak in band j and a false peak in band $j+1$). To bridge this sensitivity gap, we transform the WKS into a Cumulative Wave Kernel Signature (CWKS) by taking the cumulative sum across the energy bands. This formulation behaves analogously to the 1D Wasserstein (Earth Mover's) metric [31]: the cumulative sum natively smooths the discrepancy of shifted eigenvalues across adjacent frequency bands, correctly registering them as structurally proximate. Finally, we normalize the CWKS vector by its total spectral mass (forcing the final bucket to 1.0), ensuring the signature strictly isolates the proportional geometry and ignores the raw node count of the discretized mesh.

To empirically validate the robustness of the spectral embedding, we conducted an ablation study measuring retrieval accuracy against increasing levels of geometric noise (ϵ). As illustrated in Figure 14, we track the Top-5 Recall rate, which directly mirrors the user experience of selecting

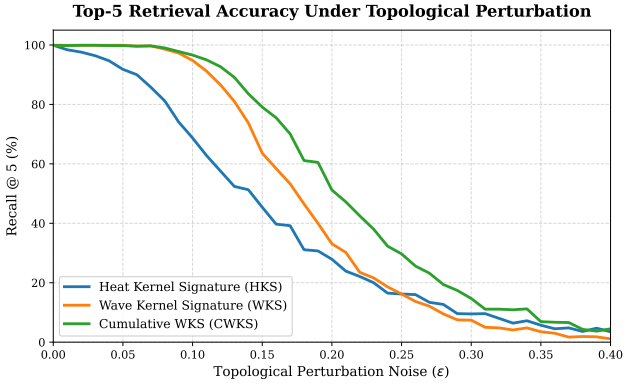


Figure 14: Ablation study of retrieval accuracy under varying amplitudes of perturbation for different embedding schemes. The cumulative wave kernel signature (CWKS) consistently outperforms the heat kernel signature (HKS) and wave kernel signature (WKS) across all noise levels.

from a gallery of top search results. Although all embedding schemes decay in accuracy with increasing noise, the CWKS scheme consistently performs better than HKS [32] and WKS across all noise levels.

6.3. Hyperparameter robustness and Database Indexing

The spectral embedding relies on a set of fixed parameters derived from the physical bounds of realistic crease patterns. We extract only the first 32 non-zero eigenvalues ($k = 32$) of the discretized tree, which then embed into $D = 64$ frequency bands. Experimental tuning reveals insensitivity once these parameters are sufficiently large—high-frequency noise or higher-resolution frequency discretization have a negligible effect on retrieval accuracy. This is demonstrated by the fact that Principal Component Analysis (PCA) shows that over 95% of the variance is captured by the first 10 components. Additionally, we observe that the corresponding log-eigenvalue spectrum reliably falls within the bounds of $E_{\min} = 3.0$ and $E_{\max} = 10.0$. Consequently, these hyperparameters remain fixed across the database, requiring no user-facing tuning or experimental optimization.

The resulting 64-dimensional CWKS vectors for every valid tiling generated during Phase III are serialized and

inserted into a flat L_2 vector index using the FAISS library [17]. At runtime, when a user queries the system, the target tree is identically processed, embedded, and queried against the FAISS index.

7. Implementation

To validate the practicality of the proposed algorithm (SEARCH), we implemented a web-based application, *ExplOri* (a portmanteau of "explore" and "origami"). The system allows users to sketch a target tree structure and retrieve the closest matching crease pattern from the precomputed database. The application can be accessed at [*URL anonymized for review*].

7.1. System Architecture

The architecture is divided into an offline precomputation engine and an online distributed web service.

Offline Precomputation and Storage The offline phase executes the three phases of the database generation pipeline. It exhaustively enumerates topologies via the SAT solver, realizes valid metric tilings via the MILP solver, constructs the interior straight skeleton crease patterns, and extracts the corresponding folded tree structures. The trees are then indexed for FAISS using their CWKS vectors.

The database stores three distinct components: an optimized FAISS L_2 vector index, the serialized tree graph (retained to accommodate changes in embedding scheme without complete database regeneration), and the exact geometry of the axial tiling. For the sake of memory efficiency, the full crease patterns are not stored—instead, they are deterministically reconstructed at runtime from the axial tiling as a form of lossless compression.

Runtime Processing and Retrieval The online phase initiates when a user sketches a target tree in the web application. The interface serializes this user input into a lightweight JSON graph representation and transmits it to the server. Upon receiving the payload, the Python backend executes the spectral translation to generate the embedding vector. The resulting vector is queried against the FAISS index. Latency profiling confirms that the FAISS L_2 lookup accounts for less than 10% of the end-to-end query time, with the majority of latency attributed to the crease pattern reconstruction of the top-k retrieved tilings. The server then returns the top-k results to the client for visualization.

7.2. Database Scaling and Robustness

The database generation process requires running the entire Phase I → Phase III process across a range of grid resolutions ($N = 2, 3, 4, 5$) and symmetry configurations (book, diagonal, none). Some of the exponentially larger databases were only partially generated to sample the full design space. The results are summarized in Table 1.

The empirical data validates the resilience of our hybrid floating-point and exact-rational pipeline. Despite evaluating millions of topologies, the error rate of the geometric processing pipeline—which includes the MILP solve, the Gauss-Jordan exact solve, straight skeleton generation, Kawasaki propagation, and tree extraction—remained strictly below 0.1% for nearly all configurations, peaking at only 0.38% for the $N = 5$ (Book symmetry) dataset.

Conversely, the NP-hard nature of the Mixed-Integer Linear Programming (MILP) formulation becomes immediately apparent as grid resolution scales. While $N = 2$ grids evaluate near-instantaneously with timeout rates under 6%, scaling to $N = 5$ results in a 30-second solver timeout rate approaching 40–50%. This presents a practical limitation on the size of the database that can be precomputed in a reasonable time frame.

Finally, the predicted number of total tilings underscores the sheer magnitude of the 22.5° design space. The $N = 5$ diagonal symmetry space alone is estimated to contain over 18 million unique, flat-foldable geometric states, though in practice we only generated around 2 million. The serialized memory footprint of these states scales predictably with structural complexity, requiring an average of 1.11 KB per tiling at $N = 2$, 1.50 KB at $N = 3$, 2.10 KB at $N = 4$, and 2.75 KB at $N = 5$.

7.3. Front-end Interface

Conventional origami CAD tools, such as TreeMaker and Box Pleating Studio, require the user to explicitly define exact numerical lengths for every edge of the tree in order to seed the underlying packing optimization solvers [1, 13]. In contrast, ExplOri takes only approximate sketches as input. As shown in Figure 15, the tree edge lengths are determined only by visual proportions. Because the CWKS spectral embedding retrieves the closest continuous topological match rather than enforcing exact subgraph isomorphism, the output tree is inherently an approximation of the input. The visual sketching interface, then, intuitively aligns user expectations with the approximate nature of the retrieval process.

The backend is configured to return the top-k nearest neighbors, where a larger k gives the user more options to choose from at the expense of longer latency. Beyond rendering the 2D crease pattern, the database payload also delivers the complete geometric context of the retrieved solution, including the flat folded form, the extracted tree structure, and its corresponding packing. ExplOri presents these results in an interactive gallery for the user to evaluate structural tradeoffs, choosing the geometric variant that best suits their aesthetic intent. Furthermore, users can dynamically filter the retrieval index via search configurations, targeting specific topology resolutions (N) or enforcing specific symmetries to narrow the search space.

Finally, a common bottleneck in 22.5° origami design is the physical realization of a crease pattern. Because 22.5° coordinate intersections lie in the $\mathbb{Q}(\sqrt{2})$ field, determining the sequence of initial folds required to construct the

Grid size N	Symmetry	Total Tilings	Success	MILP timeout	Error
2	Book	18	94.7%	5.3%	0.00%
2	Diag	27	96.4%	3.6%	0.00%
2	None	81	98.8%	1.2%	0.00%
3	Book	147	85.4%	14.6%	0.00%
3	Diag	1,417	90.8%	9.2%	0.09%
3	None	174,784	97.3%	2.6%	0.07%
4	Book	78,226	92.2%	7.8%	0.03%
4	Diag	206,644	90.5%	9.5%	0.05%
4	None	~7,890,000	76.0%	24.0%	0.03%
5	Book	~7,240,000	58.0%	41.6%	0.38%
5	Diag	~18,300,000	71.7%	28.3%	0.05%

Table 1
Algorithmic yield rates and database scale across varying grid resolutions (N) and symmetries. Total tilings for incomplete database configurations are estimated (~) by projecting the average tilings per successful topology across the estimated total topologies. Timeout rate indicates cases where MILP takes longer than 30s. Error rate indicates any failure in the pipeline during the MILP solve, exact Gauss-Jordan solve, straight skeleton generation, Kawasaki propagation, or tree extraction.

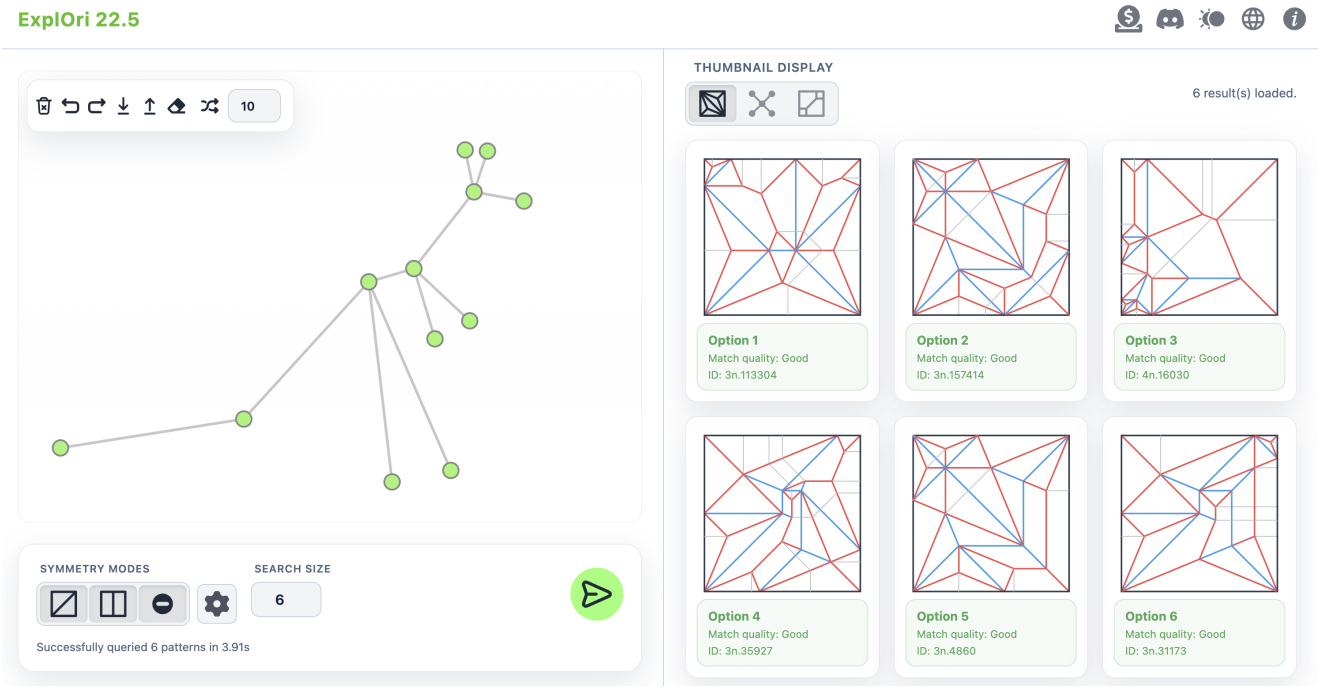


Figure 15: A screenshot of the ExplOri web interface. On the left, the user sketches a target tree structure, such as a kangaroo. On the bottom left, the user configures the search parameters. The results are displayed as an interactive gallery on the right, sorted by tree similarity.

geometry—known as "reference finding"—is generally unintuitive for human folders. Therefore, ExplOri includes an automated reference-finding module based on work by Nolan [14]. The system utilizes the exact rational coordinates of the reconstructed crease pattern to calculate the initial step-by-step geometric construction folds. This allows the user to fold the crease pattern without any measuring tools or manual calculations.

7.4. Community reception

Beyond satisfying any mathematical constraints, the ultimate metric of success for a design tool is its practical utility and adoption by the community. To evaluate this, we frame our results not as a controlled psychometric user study, but as an unconstrained "in-the-wild" deployment and qualitative observation of human-algorithm co-creation.

The ExplOri interface was hosted publicly and shared with the global community of origami designers. Within the first week of deployment, the server processed thousands of unique query trees from users in over 70 countries. Figure 16

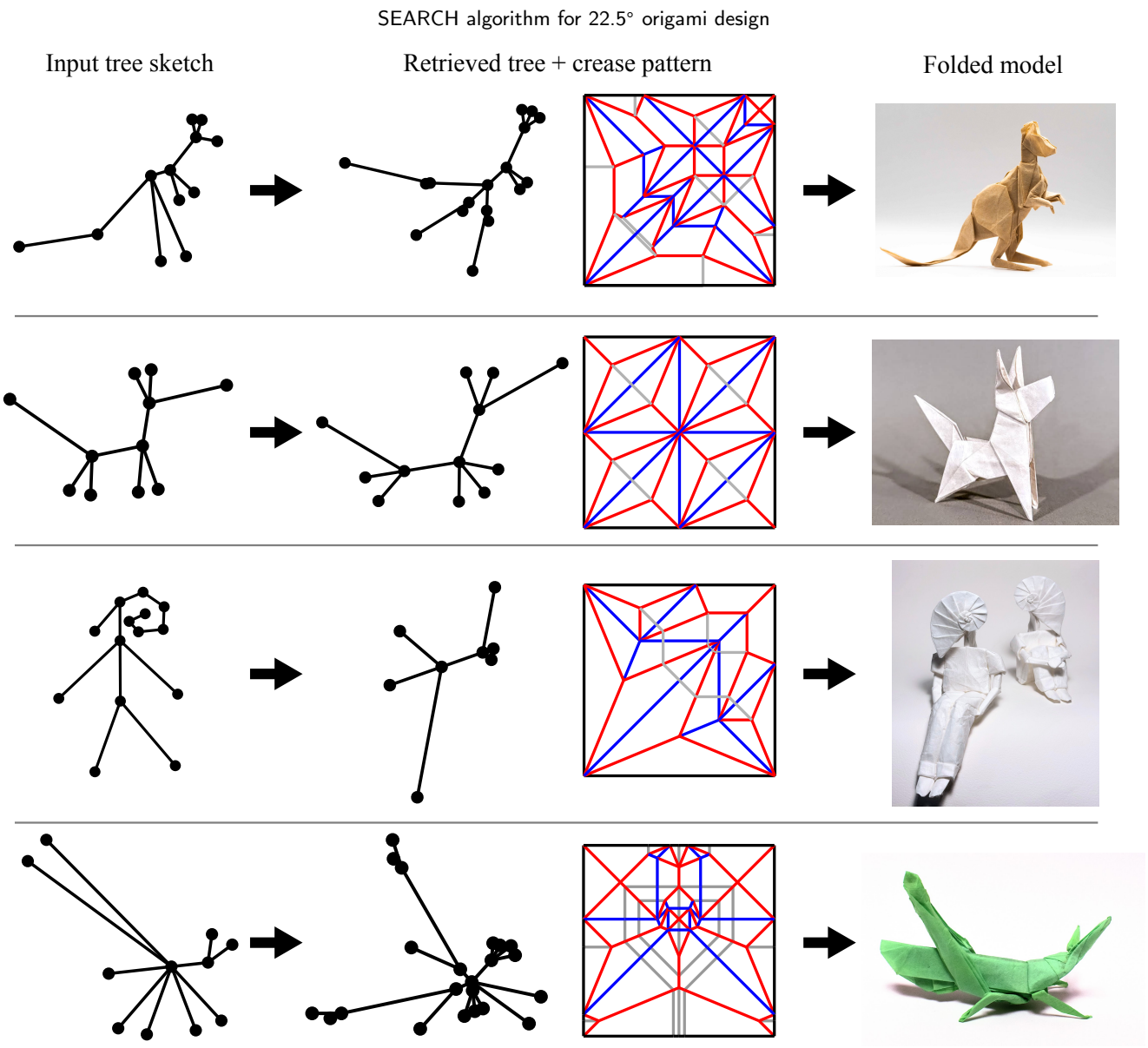


Figure 16: A sample of new 22.5° origami designs created using the ExplOri web interface. For each design, the input tree sketch, a tree + crease pattern retrieved from the database and selected by the user, and a photo of the physical folded model are shown. From top to bottom: *Kangaroo* by the author, *Watchful Dog* by Richard Ellison, *Spiraling* by Ronik Bhaskar, and *Grasshopper* by Alexander Ross.

showcases a selection of new origami designs created by the author and other members of the origami design community using the ExplOri web interface. Each design was generated by sketching a target tree structure, and upon selecting a suitable crease pattern retrieved by the search interface, physically folding and refining the design by hand.

Qualitative feedback from the community highlighted the tool's value as an ideation assistant rather than a rigid, final-state generator. In all of the final designs shown in Figure 16, the final physical structure diverges significantly from the crease pattern originally retrieved from the database. This divergence is an intended process of the workflow, as the remaining artistic component of the design process remains solely in the hands of the human.

8. Conclusion

While SEARCH 22.5° serves as a novel proof of concept, the methodology currently exhibits several limitations. The current archive of 2 million crease patterns represents only a fraction of the infinite 22.5° design space. Relying on brute-force enumeration poses severe scalability limits for finding accurate matches to complex query trees. Furthermore, even in the tractable domain of low-complexity uniaxial origami design, the query results only serve as initial suggestions for the design process—extensive artistic tweaking and iterative physical refinement are always required to arrive at a final origami design.

One direction of future work is in the 15° design domain. Similar to how 22.5° geometry is closed under $\mathbb{Q}(\sqrt{2})$, 15°

geometry is closed under $\mathbb{Q}(\sqrt{3})$ [33]. This suggests that the entire pipeline could potentially be applied to generate 15° crease patterns, which are currently far less common and less understood by origami designers.

Declaration of competing interest

Removed for anonymized review.

Acknowledgments

Removed for anonymized review.

Declaration of generative AI and AI-assisted technologies in the manuscript preparation process

During the preparation of this work, the author used Google Gemini to refine academic prose, structure the LaTeX formatting of the manuscript, and search for relevant literature. Additionally, the tool was utilized during the software development phase to assist with non-critical engineering tasks, including front-end UI design and translating prototype Python algorithms into C++. After using this tool/service, the author thoroughly reviewed, edited, and validated all generated text and code, and takes full responsibility for the content and integrity of the final publication and accompanying software.

Data Availability

The source code is available at: `https://anonymous.4open.science/r/SEARCH225`. The database of generated crease patterns will be made available upon request.

Appendix: Crease Intersections in 4D

Let crease i pass through vertex $\vec{v}_i = (x_i, y_i)$ at an angle θ_i relative to the x-axis, and crease j pass through vertex $\vec{v}_j = (x_j, y_j)$ at an angle $\theta_j \neq \theta_i$. Let the relative displacements be denoted as $\Delta x = x_j - x_i$ and $\Delta y = y_j - y_i$. We seek to find the intersection point $\vec{v}_c$ of these two creases. The two creases form rays originating from these vertices. Expressing these geometric lines in standard point-slope form, the intersection point $\vec{v}_c = (x_c, y_c)$ must satisfy both equations simultaneously:

$$y_c - y_i = \tan\theta_i (x_c - x_i) \tag{43}$$
$$y_c - y_j = \tan\theta_j (x_c - x_j) \tag{44}$$

We parameterize the intersection point relative to the origin of the second crease, $\vec{v}_j$. Let $\delta x = x_c - x_j$ and $\delta y = y_c - y_j$. From Equation (44), we know that the vertical displacement is strictly proportional to the horizontal displacement by the tangent of the ray:

$$\delta y = \delta x \tan\theta_j \tag{45}$$

We substitute $x_c = x_j + \delta x$ and $y_c = y_j + \delta x \tan\theta_j$ into the line equation for the first crease (Equation 43):

$$(y_j + \delta x \tan\theta_j) - y_i = \tan\theta_i((x_j + \delta x) - x_i) \tag{46}$$
$$\Delta y + \delta x \tan\theta_j = \tan\theta_i(\Delta x + \delta x) \tag{47}$$
$$\Delta y - \Delta x \tan\theta_i = \delta x \tan\theta_i - \delta x \tan\theta_j \tag{48}$$

Solving for the horizontal displacement δx yields:

$$\delta x = \frac{\Delta y - \Delta x \tan\theta_i}{\tan\theta_i - \tan\theta_j} \tag{49}$$

Once δx is computed, the corresponding vertical displacement is simply $\delta y = \delta x \tan\theta_j$. All trigonometric functions for 22.5° multiples are precomputed analytically in the exact form of $r + s\sqrt{2}$, so the derived displacements δx and δy are also guaranteed to take this form.

The exact 4D coordinate of the intersection is then recovered by adding this vector displacement to 4D projection of the starting vertex $\vec{v}_j^{(4D)}$:

$$\vec{v}_c^{(4D)} = \vec{v}_j^{(4D)} + \delta x \hat{X}^{(4D)} + \delta y \hat{Y}^{(4D)} \tag{50}$$

where $\hat{X}^{(4D)} = [1,0,0,0]^T$ and $\hat{Y}^{(4D)} = [0,0,1,0]^T$ are the 2D Cartesian axes lifted onto the 4D rational basis with Equation 29.

To ensure the coordinates remain strictly within the rational field during this addition, scalar multiplication by an irrational magnitude $(r + s\sqrt{2})$ must be evaluated as a linear matrix transformation rather than a floating-point distribution. For any 4D rational vector $\vec{v}^{(4D)} = [a,b,c,d]^T$, the multiplication is defined as:

$$(r + s\sqrt{2})\vec{v}^{(4D)} = r\vec{v}^{(4D)} + sM_{\sqrt{2}}\vec{v}^{(4D)} \tag{51}$$

where $M_{\sqrt{2}}$ is the constant scaling matrix that shifts the rational components to account for the $\sqrt{2}$ factor:

$$M_{\sqrt{2}} = \begin{bmatrix} 0 & 1 & 0 & -1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ -1 & 0 & 1 & 0 \end{bmatrix} \tag{52}$$

This closed matrix-vector operation mathematically guarantees that the computed intersection point $\vec{v}_c^{(4D)}$ incurs zero precision loss.

Edge case 1: Crease j is vertical ($\theta_j = 90°$) Because the second line is vertical, the horizontal coordinate of the intersection must match x_j. Therefore, the displacement from $\vec{v}_i^{(4D)}$ is exactly Δx. We compute the intersection relative to $\vec{v}_i^{(4D)}$:

$$\vec{v}_c^{(4D)} = \vec{v}_i^{(4D)} + \Delta x \hat{X}^{(4D)} + (\Delta x \tan\theta_i)\hat{Y}^{(4D)} \tag{53}$$

Edge case 2: Crease i is vertical ($\theta_i = 90°$) Because the first line is vertical, the horizontal coordinate of the intersection must match x_i. The horizontal displacement from $\vec{v}_j^{(4D)}$ to the intersection is therefore $\delta x = -\Delta x$. The vertical position relative to y_j is $(-\Delta x)\tan\theta_j$. Then subsituting $y_j = y_i + \Delta y$:

$$\vec{v}_c^{(4D)} = \vec{v}_i^{(4D)} + (\Delta y - \Delta x \tan\theta_j)\hat{Z}^{(4D)} \tag{54}$$

References

[1] Robert J. Lang. A computational algorithm for origami design. In *Proceedings of the twelfth annual symposium on Computational geometry*, pages 98–105, 1996. doi: 10.1145/237218.237258.

[2] Tomohiro Tachi and Erik D. Demaine. Degenerative coordinates in 22.5° grid system. In *Origami5: Proceedings of the 5th International Conference on Origami in Science, Mathematics and Education (OSME 2010)*, page to appear, Singapore, 2010. A K Peters.

[3] Robert J. Lang. *Origami Design Secrets: Mathematical Methods for an Ancient Art*. CRC Press, 2nd edition, 2011. ISBN 9781568814360.

[4] Toshikazu Kawasaki. On the relation between mountain-creases and valley-creases of a flat origami. In Humiaki Huzita, editor, *Proceedings of the 1st International Meeting on Origami Science and Technology*, pages 229–237, 1989.

[5] Erik D. Demaine and Joseph O'Rourke. *Geometric Folding Algorithms: Linkages, Origami, Polyhedra*. Cambridge University Press, 2007. doi: 10.1017/CBO9780511735172.

[6] Erik D. Demaine and Tomohiro Tachi. Origamizer: A practical algorithm for folding any polyhedron. In *Proceedings of the 33rd International Symposium on Computational Geometry (SoCG)*, pages 34:1–34:16, 2017. doi: 10.4230/LIPIcs.SoCG.2017.34.

[7] Jun Mitani. A design method for 3d origami based on rotational sweep. *Computer-Aided Design and Applications*, 6(1):69–79, 2009. doi: 10.3722/cadaps.2009.69-79.

[8] Tomohiro Tachi. Designing freeform origami tessellations by generalizing resch's patterns. *Journal of Mechanical Design*, 135(11): 111006, 2013. doi: 10.1115/DETC2013-12326.

[9] Oswin Aichholzer, Franz Aurenhammer, David Alberts, and Bernd Gärtner. A novel type of skeleton for polygons. *Journal of Universal Computer Science*, 1(12):752–761, 1995. doi: 10.1007/978-3-642-80362-8_47.

[10] Mu-Tsun Tsai. On the constructions of generalized offset pythagorean stretch patterns. In *Origami 8: Eighth International Meeting of Origami Science, Mathematics, and Education*, volume 4, pages 3–19. Tarquin, 2025.

[11] Robert J Lang and Mu-Tsun Tsai. Generalized offset pythagorean stretches in box-pleated uniaxial bases. In *Origami 7: Seventh International Meeting of Origami Science, Mathematics, and Education*, volume 2, pages 591–606. Tarquin, 2018.

[12] Brandon Wong. New techniques in hex pleating for representational origami design. In *Origami 8: Proceedings of the 8th International Meeting on Origami in Science, Mathematics, and Education*. Springer, 2026. doi: 10.1007/978-981-96-6711-6_2.

[13] Mu-Tsun Tsai. Box Pleating Studio. https://bp-studio.github.io/, 2026.

[14] Travis A. Nolan. 22.5-degree reference optimization via continued fractions. In *Origami 8: Proceedings of the 8th International Meeting on Origami in Science, Mathematics, and Education*, pages 151–167. Springer, 2026. doi: 10.1007/978-981-96-6561-7_10.

[15] Natraj Iyer, Subramaniam Jayanti, Kuiyang Lou, Y Kalyanaraman, and Karthik Ramani. Three-dimensional shape searching: state-of-the-art review and future trends. *Computer-Aided Design*, 37(5):509–530, 2005. doi: 10.1016/j.cad.2004.07.008.

[16] Mathieu Aubry, Ulrich Schlickewei, and Daniel Cremers. The wave kernel signature: A quantum mechanical approach to shape analysis. In *2011 IEEE International Conference on Computer Vision Workshops*, pages 1626–1633, 2011. doi: 10.1109/ICCVW.2011.6130456.

[17] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 7(3):535–547, 2019. doi: 10.1109/TBDATA.2019.2921572.

[18] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008. doi: 10.1007/978-3-540-78800-3_24.

[19] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational geometry: algorithms and applications*. Springer, 2008.

[20] George L Nemhauser and Laurence A Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, 1988.

[21] John Forrest et al. COIN-OR Branch and Cut (CBC). https://github.com/coin-or/Cbc, 2004.

[22] Stuart Mitchell, Michael O'Sullivan, and Iain Dunning. Pulp: A linear programming toolkit for python. *The University of Auckland*, 2011.

[23] Charles Bouillaguet and Claire Delaplace. Sparse Gaussian elimination Modulo p: An update. pages 101–116, 2016. doi: 10.1007/978-3-319-45641-6_8.

[24] Robert J. Lang, Robby Kraft, Thomas Hull, and Tomohiro Tachi. Polygon packing, Hyper-CPs, and 22.5 geometry. Unpublished manuscript, 2022.

[25] Erik D. Demaine, Martin L. Demaine, and Anna Lubiw. Folding and one straight cut suffice. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 891–892, 1999.

[26] Raul Sampedro. py_straight_skeleton: Implementation of straight skeleton computation. https://pypi.org/project/py_straight_skeleton/, 2024.

[27] Lutz Kettner, Kurt Mehlhorn, Sylvain Pion, Stefan Schirra, and Chee Yap. Classroom examples of robustness problems in geometric computations. *Computational Geometry*, 40(1):61–78, 2008. doi: 10.1016/j.comgeo.2007.05.004.

[28] Tomohiro Tachi. Simulation of rigid origami. In *Origami 4*, pages 175–187. A. K. Peters, Ltd., 2009.

[29] Mark Meyer, Mathieu Desbrun, Peter Schröder, and Alan H. Barr. Discrete Differential-Geometry Operators for Triangulated 2-Manifolds. In *Visualization and Mathematics III*, pages 35–57. Springer Berlin Heidelberg, 2003. doi: 10.1007/978-3-662-05105-4_2.

[30] Peter Kuchment. Quantum graphs. i. some basic structures. *Waves in Random and Complex Media*, 14:S107–S128, 2004. doi: 10.1088/0959-7174/14/1/014.

[31] Gabriel Peyré and Marco Cuturi. Computational optimal transport: With applications to data science. *Foundations and Trends in Machine Learning*, 11(5-6):355–607, 2019. doi: 10.1561/2200000073.

[32] Jian Sun, Maks Ovsjanikov, and Leonidas Guibas. A concise and provably informative multi-scale signature based on heat diffusion. In *Computer Graphics Forum*, volume 28, pages 1383–1392, 2009. doi: 10.1111/j.1467-8659.2009.01515.x.

[33] Steve Butler, Erik Demaine, Ron Graham, and Tomohiro Tachi. Constructing points through folding and intersection. *International Journal of Computational Geometry and Applications*, 23(1):49–64, 2013. doi: 10.1142/S0218195913500039.