Computer Physics Communications 328 (2026) 110349

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

Computer Programs in Physics

PyKirigami: An interactive python simulator for Kirigami structures

Qinghai Jiang, Gary P.T. Choi *

Department of Mathematics, The Chinese University of Hong Kong, China

ARTICLE INFO

The review of this paper was arranged by Prof. J Ballantyne

Keywords:
Kirigami
Deployment
Simulation
Shape-morphing

ABSTRACT

In recent years, the concept of kirigami has been used in creating deployable structures for various scientific and technological applications. While high-fidelity Finite Element Analysis (FEA) is the standard for analyzing stress distributions and material deformation, it is computationally intensive and often ill-suited for the rapid exploration of vast kinematic configuration spaces. In this work, we develop PyKirigami, a lightweight, open-source Python framework for the efficient deployment simulation of kirigami structures. Unlike continuum mechanics solvers, PyKirigami models tessellations as articulated rigid-body networks, allowing for the real-time simulation of global deployment trajectories and volumetric transformations. The tool incorporates collision detection and interactive actuation, enabling users to validate folding paths and identify geometric locking states in both 2D and 3D topologies. This framework serves as a fast kinematic prototyping tool for kirigami structures, allowing researchers to verify deployment mechanics and self-contacts prior to performing detailed mechanical analysis or physical fabrication.

Program summary

Program Title: PyKirigami
CPC Library link to program files: (to be added by Technical Editor)
Developer's repository link: https://github.com/andy-qhjiang/PyKirigami
Licensing provisions: Apache-2.0
Programming language: Python
External routines/libraries: NumPy, PyBullet
Nature of problem: Validating the deployment of complex kirigami structures typically relies on computationally expensive Finite Element Analysis (FEA) or physical prototyping. Using FEA for purely kinematic exploration, specifically for checking folding paths, bistability, and self-collisions, is inefficient, particularly when iteratively searching through large design spaces or testing multiple actuation strategies.
Solution method: PyKirigami bypasses the heavy computational load of continuum stress analysis by solving the non-smooth contact dynamics of an articulated rigid-body system. The software implements a graph-based topology parser that maps 2D input tessellations into a 3D constrained multi-body network. It utilizes an impulse-based dynamic solver (Projected Gauss–Seidel) to resolve joint constraints and contacts in real-time. This allows for the lightweight, interactive discovery of collision-free deployment trajectories, serving as a rapid verification step before high-fidelity analysis.

1. Introduction

Kirigami, the traditional paper-cutting art, has recently become a subject of great interest in science and engineering [1–6]. In particular, a wide range of deployable structures with different shape-morphing effects have been developed by introducing cuts on a sheet of materials and connecting different components using suitable joints. Because of their great flexibility and shape-shifting property, kirigami-based structures have been widely applied in soft electronics [7], energy storage [8], and robotics [9].

Most prior works on kirigami design have focused on planar deployable structures and their in-plane deformations based on simple cutting patterns, such as rotating triangles [10], quadrilaterals [11,12], and other periodic tilings [13–15] and aperiodic tilings [16], as well as their applications to the design of graphene kirigami with high stretchability and ductility [17] and crystallographically programmed

* Corresponding author.
E-mail addresses: qhjiang@math.cuhk.edu.hk (Q. Jiang), ptchoi@cuhk.edu.hk (G.P.T. Choi).

https://doi.org/10.1016/j.cpc.2026.110349
Received 16 February 2026; Received in revised form 3 July 2026; Accepted 3 August 2026
Available online 5 August 2026

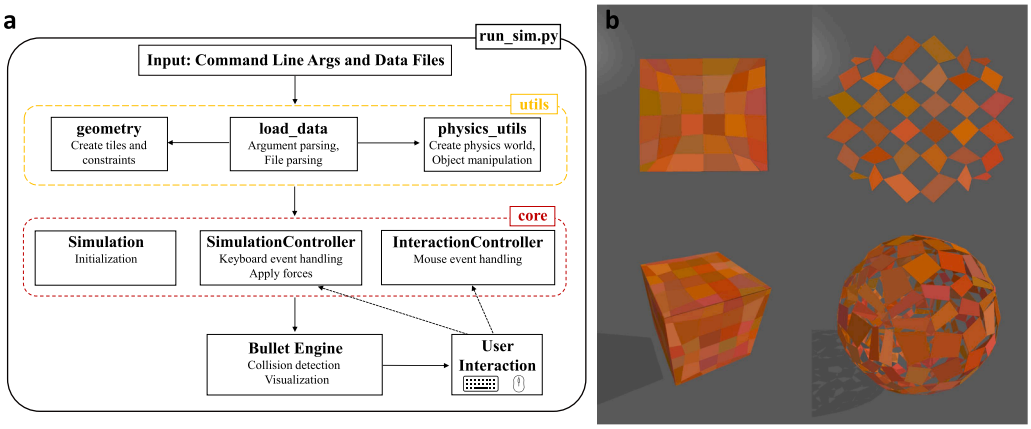

Fig. 1. An overview of PyKirigami. a, The computational framework of PyKirigami. **b**, Using PyKirigami, one can easily simulate the deployment process of different kirigami structures in 2D (top) and 3D (bottom).

metamaterials [18]. In recent years, some works have also explored the design of more general kirigami patterns for achieving different physical and geometrical effects [19–28]. There is also an increasing interest in designing kirigami for achieving not just two-dimensional (2D) but three-dimensional (3D) shape transformations with different desired 2D-to-3D or 3D-to-3D shape-morphing effects [29–36], together with their morphological and mechanical performance [37].

While the design of kirigami structures has been extensively studied, computational methods for the simulation and analysis of their shape-morphing process are much less explored. For origami structures, many existing computational tools have been established for simulating their folding process. For instance, the Rigid Origami Simulator [38] simulates the kinematics of rigid origami from given crease patterns. The Freeform Origami [39] simulates origami folding and provides interactive functionalities for altering the model crease patterns. MERLIN [40] and MERLIN2 [41] are MATLAB-based software tools for simulating the folding of non-rigid origami. Origami Simulator [42] is a GPU-accelerated and web-based origami simulator with interactive folding control and real-time rendering. On the contrary, for kirigami structures, most prior kirigami design works directly used finite element analysis (FEA) software such as ABAQUS (e.g., [28,43,44]) or COMSOL (e.g., [45–47]) to simulate the deployment process of kirigami structures under certain boundary loading conditions. Some other works considered other mechanical models involving linear springs for simulating 2D-to-2D and 2D-to-3D deployments in MATLAB [20,22]. More recently, Liu et al. [16] developed a 2D-to-2D kirigami deployment simulator in Python based on the 2D rigid body physics library Pymunk, with a focus on rigidly deployable planar kirigami patterns. However, in many practical applications, it is necessary to simulate more general kirigami deployment processes in both 2D and 3D. Also, one may need to consider not just the kirigami structures but also their interactions with the environment and external objects throughout the deployment. Moreover, it is highly desirable to have a lightweight, efficient, and freely available deployment simulator with interactive control functionalities that allow for real-time manipulation and analysis of kirigami structures.

In this work, we develop PyKirigami, an open-source Python-based simulator, for simulating the 2D and 3D deployment of general kirigami structures (Fig. 1). Specifically, our simulator utilizes the PyBullet physics engine [48] and is capable of simulating 2D-to-2D, 2D-to-3D, and 3D-to-3D kirigami deployments. It also supports the configurations of environmental effects and stabilization mechanisms for modeling different scenarios. It further allows dynamic control of the boundary conditions throughout the deployment process, thereby providing great flexibility in achieving different desired deployment effects. Using PyKirigami, one can easily simulate the deployment and analyze the properties of a wide range of kirigami structures.

2. Methods

2.1. Overview

The goal of our PyKirigami simulator is to simulate the deployment process of kirigami structures, typically from a compact initial state to a deployed configuration in either 2D or 3D. More specifically, the functionalities of PyKirigami (Fig. 1**a**) can be categorized as follows:

- **Core Simulation Logic**: This component is responsible for the main simulation loop, handling physics calculations, and managing the state of the simulated objects. It includes classes for managing the simulation state, handling user interactions (like pausing and resetting), and applying forces to the objects.
- **Utility Functions**: This group contains modules for parsing command-line arguments, loading input data (such as vertex coordinates and constraints), and performing geometric operations like creating 3D models from 2D definitions.
- **Main Executable**: The main executable script initializes the simulation environment, orchestrates the simulation loop, and integrates the various modules.

The core concepts in PyKirigami are presented in further detail in the following sections. See also Appendix A–Appendix B for more detailed explanations.

2.2. Geometric representation: From 2D polygons to 3D tiles

In our work, we consider polygonal tiles formed by straight cuts, noting that curved cuts can be discretized and approximated using multiple straight cuts in practical applications. Also, in prior works on kirigami design and simulation [16,20], the kirigami tiles were commonly considered as two-dimensional objects with zero thickness for simplicity. By contrast, here we consider the kirigami tiles as 3D rigid bodies ("bricks") in both the in-plane and 3D deployment simulations, thereby enabling a more realistic simulation process.

More mathematically, let $\{\mathcal{T}_i\}_{i=1}^N$ be the collection of all kirigami tiles in a kirigami structure, where each tile $\mathcal{T}_i$ is initially defined by a sequence of vertices $\{v_{i,1}, v_{i,2}, \dots, v_{i,n_i}\}$, where $v_{i,j} \in \mathbb{R}^3$ and n_i is the number of vertices for tile $\{\mathcal{T}_i\}$ (see also Fig. 2**a**). To create a more physically realistic model, each planar tile is extruded into a 3D rigid body, or "brick," with a user-defined thickness, t. This process is detailed in

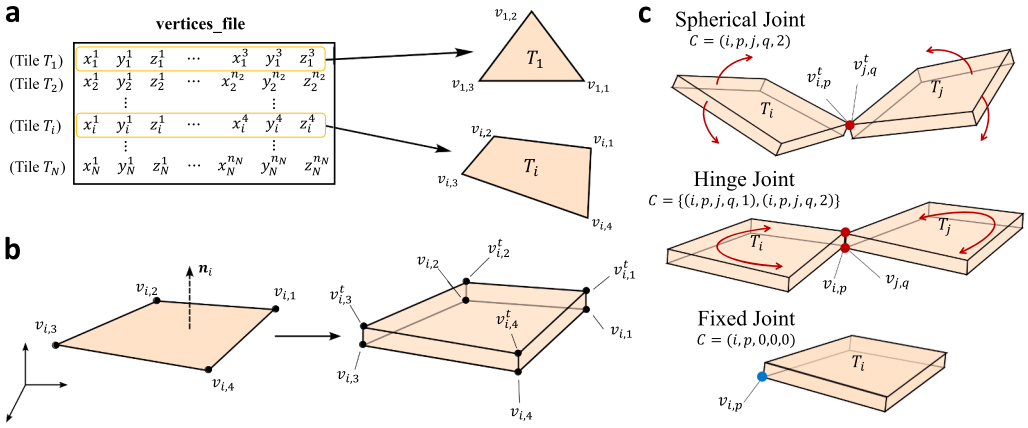

Fig. 2. Tile construction and connections in PyKirigami. a, An illustration of the vertex coordinates encoded in the input file via `--vertices_file`. Here, note that different tiles may contain different number of vertices. **b**, Geometric construction of the 3D kirigami tiles. (Left) The original polygonal tile $\mathcal{T}_i$, with the dotted arrow showing its normal vector. (Right) The 3D tile geometry constructed by extruding $\mathcal{T}_i$ along its normal vector. Here, the thickness of the tile can be prescribed by the user. **c**, Three different types of inter-tile connections in PyKirigami. (Top) Spherical joint applied to tiles $\mathcal{T}_i$ and $\mathcal{T}_j$ such that the rotational degree of freedom equals 3. (Middle) Hinge joint such that the rotational degree of freedom equals 1. (Bottom) Fixed joint applied to $\mathcal{T}_i$ and world frame such that the object is totally fixed. The arrow represents the rotational degree of freedom.

Algorithm 1: 3D tile generation pipeline.

Input: Vertex coordinates file, brick thickness parameter t
Output: PyBullet body IDs, local vertex coordinates for each tile
Load vertex data from input file
for each tile $\mathcal{T}_i$ with vertices $\{v_{i,1}, v_{i,2}, \dots, v_{i,n_i}\}$ **do**
  Compute geometric center: $\mathbf{c}_i = \frac{1}{n_i}\sum_{j=1}^{n_i} v_{i,j}$.
  Calculate normal vector: $\mathbf{n}_i = \frac{(v_{i,2}-v_{i,1})\times(v_{i,3}-v_{i,1})}{\|(v_{i,2}-v_{i,1})\times(v_{i,3}-v_{i,1})\|}$.
  Generate top vertices: $v^t_{i,j} = v_{i,j} + t \cdot \mathbf{n}_i$ for $j = 1, \dots, n_i$.
  Store local coordinates: $v_{i,j} - \mathbf{c}_i$ and $v^t_{i,j} - \mathbf{c}_i$.
  Create visual mesh and collision shape from $2n_i$ vertices.
  Instantiate rigid body in physics engine, return body ID.
end for

Algorithm 1 and illustrated in Fig. 2**b**. See also Appendix B for more details of the geometric representation.

We remark that by modeling the polygonal tiles as 3D rigid bodies, PyKirigami focuses on the rigid deployment of kirigami structures and excludes other complex effects such as panel bending, stretching, crease compliance, material elasticity, plasticity, and fracture. Certain non-rigid deployment scenarios may be further handled by some specific strategies and simulation setups, which are discussed in Appendix C.8.

2.3. Kinematic connections: Modeling joints with constraints

The connections between tiles in kirigami structures are crucial for their kinematic behavior. Building upon existing work, we adopt and extend the constraint representation framework introduced in [16].

Extension from 2D to 3D constraint representation. In the prior 2D work [16], each constraint is represented as a 4-tuple $C_k = (i, p, j, q)$, which enforces that vertex p of tile i coincides with vertex q of tile j. This constraint effectively creates a spherical joint in 2D space such that

$$v_{i,p} = v_{j,q}. \tag{1}$$

Extending this to 3D naturally requires an additional z-coordinate constraint. However, since our tiles are 3D rigid bodies with distinct top and bottom faces, we must specify which face contains the connection point. We therefore extend the constraint representation to a 5-tuple $C_k = (i, p, j, q, t)$, where t is a face specifier: $t = 1$ for bottom face connections and $t = 2$ for top face connections. For backward compatibility, if t is omitted (4-tuple format), we default to bottom face connection.

Joint types and their implementation. By strategically combining different joints in PyBullet, we can realize different types of kinematic connections with varying degrees of freedom. Each connection type corresponds to a specific pattern of vertex coincidence constraints (see Fig. 2**c**):

- **Spherical Joint**: The fundamental connection type, implemented using a single point-to-point constraint that enforces coincidence of one vertex from each tile (Fig. 2**c**, top). This creates a ball-and-socket joint allowing three rotational degrees of freedom around the pivot point.
- **Hinge Joint**: Another connection type constructed using two spherical joints to constrain rotation to a single axis (Fig. 2**c**, middle).
- **Fixed Joint**: A connection type implemented using PyBullet's native `JOINT_FIXED` constraint type (Fig. 2**c**, bottom). This rigid connection completely eliminates all relative degrees of freedom between objects, preventing any relative motion. Fixed joints are particularly useful for creating structural stability and for temporarily immobilizing specific tiles during interactive simulations.

We remark that besides utilizing the user-provided connection information, PyKirigami also provides an automatic connection-type detection functionality for users who are unsure about the connection types to be used. See Appendix B.3 for more details.

2.4. Force models for actuation and dynamics

After constructing the kirigami tiles and their connections, we implement a physics-based deployment simulation. The dynamic behavior of the kirigami structure is governed by various configurable force models, including simple expansion forces for basic radial deployment and more advanced target-based actuation.

Center-of-mass expansion forces. For basic radial expansion of kirigami structures, PyKirigami provides an automatic expansion mode that applies forces to each tile's center of mass, directing them outward from the structure's global center. This method is particularly useful when the target positions are not specified, serving as a fallback or simplified actuation approach.

Target-based deployment forces. While simple radial expansion can be achieved using center-of-mass forces described above, more complex deployments require a more sophisticated approach. Our simulator implements a vertex-based target model that provides precise control over the

deployment process. In this model, each vertex of each tile is assigned a target position, and forces are applied to guide the vertices toward these targets. The deployment process is implemented through Algorithm 2, which calculates and applies appropriate forces and torques to achieve the desired configuration:

Algorithm 2: Vertex-based target deployment framework.

Input: Body IDs $\{b_i\}$; local vertex coordinates $\{\mathbf{u}_{i,j}\}_{j=1}^{m_i}$ per tile; target vertex positions $\{\tilde{v}_{i,j}\}_{j=1}^{m_i}$; spring stiffnesses k_f, k_τ; damping coefficients μ_v, μ_ω
for each body b_i representing tile $\mathcal{T}_i$ **do**
  Query center of mass $\mathbf{c}_i$, orientation R_i, linear velocity $\mathbf{v}_i$, and angular velocity $\boldsymbol{\omega}_i$ from the physics engine.
  Transform local vertices to world frame: $v_{i,j} = R_i\,\mathbf{u}_{i,j} + \mathbf{c}_i$.
  Initialize $\mathbf{F}_i \leftarrow \mathbf{0}$, $\boldsymbol{\tau}_i \leftarrow \mathbf{0}$.
  for $j = 1$ **to** m_i **do**
    Compute displacement: $\boldsymbol{\delta}_{i,j} = \tilde{v}_{i,j} - v_{i,j}$.
    Compute moment arm from CoM: $\mathbf{r}_{i,j} = v_{i,j} - \mathbf{c}_i$.
    $\mathbf{F}_i \mathrel{+}= k_f\,\boldsymbol{\delta}_{i,j}$
    $\boldsymbol{\tau}_i \mathrel{+}= k_\tau\,(\mathbf{r}_{i,j} \times \boldsymbol{\delta}_{i,j})$
  end for
  Apply damped force and torque to body b_i:
  $$\hat{\mathbf{F}}_i = \mathbf{F}_i - \mu_v\,\mathbf{v}_i, \qquad \hat{\boldsymbol{\tau}}_i = \boldsymbol{\tau}_i - \mu_\omega\,\boldsymbol{\omega}_i.$$
end for

This vertex-based approach offers several advantages over simpler force models, including precise shape control, natural rotational behavior, and the ability to follow complex deployment paths.

Adaptive stiffness. We remark that an issue with the above-mentioned linear-spring actuation is that the force $k_f(\tilde{v}_{i,j} - v_{i,j})$ decays to zero as each vertex approaches its target, making it difficult to overcome residual friction or geometric locking. To address this, PyKirigami employs an adaptive stiffness scheme. During the automatic deployment, the controller monitors the maximum vertex error across all active tiles. If the error fails to improve for 60 consecutive simulation steps, the spring constant k_f is doubled (up to a maximum of $\times 16$).

Environmental and stabilization forces. Beyond the actuation forces, PyKirigami models key environmental effects and stabilization mechanisms, which are fully configurable by the user to simulate diverse physical scenarios:

- **Gravitational Loading:** The magnitude and direction of the gravity vector can be specified to model deployment under various load conditions.
- **Numerical Damping:** To enhance stability and simulate energy dissipation, the model includes two configurable damping mechanisms. First, tunable linear and angular damping coefficients are applied to each rigid body to dissipate system-wide kinetic energy and ensure stable convergence to quasi-static equilibrium. Second, the actuation forces incorporate a damping factor (as seen in Algorithm 2) to promote smooth and stable deployment dynamics.
- **Contact Modeling:** The simulator supports interactions with external surfaces, such as a configurable ground plane, and models the resulting collision and friction forces.

Self-penetration avoidance. PyKirigami tries to avoid self-penetration during the deployment simulation through several complementary mechanisms that operate at different levels of the simulation pipeline. In particular, besides the built-in rigid body collision detection offered in PyBullet, PyKirigami also includes functionalities for avoiding self-penetration potentially caused by the inter-tile connection type and incorrect global target assignment. It further provides functions for decomposing complex deployments into a sequence of simpler sub-deployments to avoid self-penetration due to complex target shape changes. A more detailed description is provided in Appendix B.6.

3. Results

3.1. 2D and 3D deployment simulations

To demonstrate the versatility and effectiveness of PyKirigami, we present a curated set of examples grouped by deployment dimensionality. These cases showcase how the simulator's automated deployment mechanisms, environmental controls, and interactive features are applied to solve distinct challenges in 2D and 3D kirigami systems. For detailed simulation parameters and per-example configurations, see Appendix C. For the software environment and detailed commands for reproducing all examples, see Appendix E. See also Supplementary Video S1–S4 for the videos of the deployment process of various 2D and 3D kirigami examples.

3.1.1. 2D deployments and interactive morphing

Planar kirigami systems often involve in-plane expansion or complex shape-morphing between multiple stable states. We demonstrate PyKirigami's capabilities for these scenarios with two distinct examples.

First, we consider the 2D expansion of the square-to-circle kirigami structure from [20] as shown in Fig. 3**a**. This example highlights the use of the basic Center-of-Mass Expansion Forces for straightforward radial deployment. More importantly, it demonstrates the essential role of Environmental and Stabilization Forces. By enabling a ground plane (Contact Modeling) and a downward gravity vector (Gravitational Loading), the deployment is constrained to the XY-plane, effectively simulating deployment on a physical surface and preventing the out-of-plane buckling that would otherwise occur in such a flexible structure.

Second, we showcase a compact reconfigurable kirigami model from [27], which has multiple closed and compact contracted states (see Fig. 3**b**). Unlike the open deployment in panel **a**, this transition involves strong hinge rotations, contact, and friction, and is sensitive to dissipation and control. We therefore analyze its passive dynamics (implicit hinge elasticity) and show how to achieve stable, deterministic deployment via target-based actuation in the next subsection.

3.1.2. 3D deployments and volumetric transformations

The transformation from flat sheets to 3D surfaces, or between different 3D configurations, represents a significant challenge in kirigami simulation. Our next examples demonstrate how PyKirigami addresses these complex, out-of-plane dynamics.

The 2D-to-3D deployment of a square-to-spherical-cap kirigami model shown in Fig. 3**c** exemplifies the necessity of the Target-Based Deployment Forces (Algorithm 2). Simple expansion or manual interaction is insufficient to control the complex, coupled bending and stretching deformations required for this transformation. The vertex-based target model provides the precise control needed to guide the structure through its out-of-plane path, ensuring stable convergence to the desired spherical geometry. This case also underscores the importance of parameter tuning, where the actuation spring stiffness (k_f) and damping (μ_v) are calibrated to balance deployment speed with stability.

Finally, we demonstrate a 3D-to-3D deployment with a compact reconfigurable cylinder kirigami structure as shown in Fig. 3**d**. This highlights the ability of the Target-Based Deployment Forces for guiding complex, non-radial paths in three dimensions. Actuating the structure from a compact, folded state to its fully deployed form requires targets that define not just a final shape, but a collision-free volumetric transformation. This proves the model's suitability for simulating advanced systems like deployable space structures or reconfigurable robotics.

For completeness, while all examples in Fig. 3 use automated actuation (radial or target-based forces applied each simulation step),

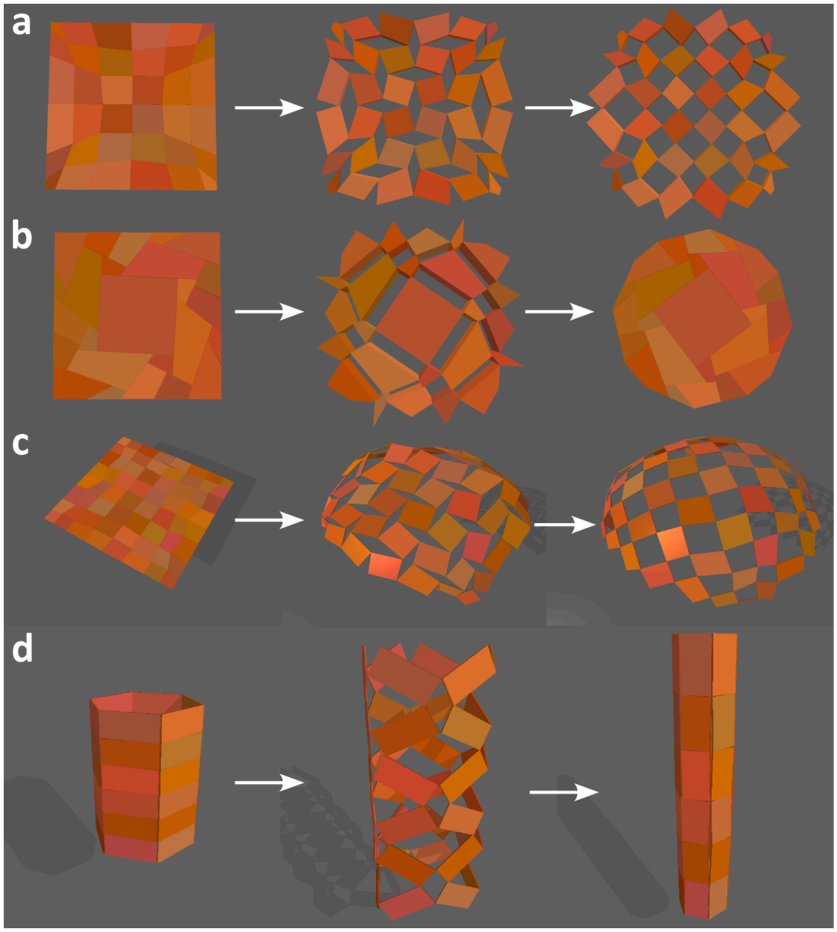

Fig. 3. Examples of 2D and 3D kirigami deployment achieved by PyKirigami. a, The 2D deployment of the square-to-circle kirigami structure achieved by PyKirigami with automated radial expansion, stabilized by environmental forces. **b**, The 2D deployment of the compact reconfigurable kirigami model produced by PyKirigami. **c**, The 2D-to-3D deployment of the square-to-spherical-cap kirigami model achieved by PyKirigami, driven by the target-based force algorithm. **d**, The 3D-to-3D deployment and reconfiguration of the cylinder kirigami model achieved by PyKirigami, showcasing volumetric shape control.

PyKirigami also supports a no-force mode and fully interactive runs in the GUI (manual actuation, pin/unpin, pause/reset). In Appendix C, we show how to apply these functionalities to achieve the target shape of Fig. 3**b**.

Besides the above examples, in Appendix C we further discuss how more complex deployments of kirigami structures can be achieved in PyKirigami. In particular, we demonstrate the ability of PyKirigami for handling the deployment of non-rigidly deployable kirigami structures as well as kirigami structures with highly complex deployment paths and complex topologies.

3.2. Geometric fidelity of deployment

In PyKirigami, inter-tiles joints are enforced iteratively by the underlying physics engine at each simulation timestep. Unlike reduced-coordinate formulations that encode kinematic loops as exact algebraic constraints, iterative enforcement inherently permits small constraint violation - a positional drift between nominally coincident vertices - whose magnitude depends on the number of solver iterations. To rigorously assess the impact of this numerical relaxation on kinematic accuracy, we perform a quantitative validation on two distinct kirigami structures: a parametric rectangular tessellation kirigami structure and a complex square-to-disk kirigami structure. In both cases, the simulated deployment trajectories are compared directly against exact analytical solutions derived from the geometric design parameters.

3.2.1. Benchmark case 1: Rectangular tessellation

We first consider a standard $M \times N$ rectangular tessellation kirigami structure with deployment angle θ as a benchmark. The theoretical vertex position p_i^{th} at θ can be derived easily (see Appendix D). We initialize a 4×4 grid in PyKirigami and actuate it from $\theta = 0$ to $\theta = \pi/2$. At each time step, we compute the Euclidean error $\|p_i^{\text{sim}} - p_i^{\text{th}}\|$ for all vertices i.

As shown in Fig. 4, the impulse-based solver possesses a very low positional error throughout the deployment process. The simulation initializes with a transient spike in maximum deviation 2.5% as the solver resolves the ill-conditioned constraints near the compact state. Crucially, the mean relative error remains exceptionally low throughout the entire deployment process. This confirms that the transient spike is isolated to a few specific tiles and joints and does not propagate through the system. The negligible mean error demonstrates that PyKirigami maintains vertex-level coherence, verifying the implementation of the topology constraint.

3.2.2. Benchmark case 2: Square-to-disk structure

Having validated the solver's precision on the above benchmark case, we consider analyzing the deployment of the more complex square-to-disk structure as shown in Fig. 3**b**. As explained in the theoretical construction of the structure in [27], the deployment of it is also a single-degree-of-freedom mechanism (see Appendix D) where the global radius R is a deterministic function of the tile rotation angle θ. Therefore, in Fig. 5 we plot the global radius against the local deployment angle obtained by Pykirigami and compare the result with the analytical result

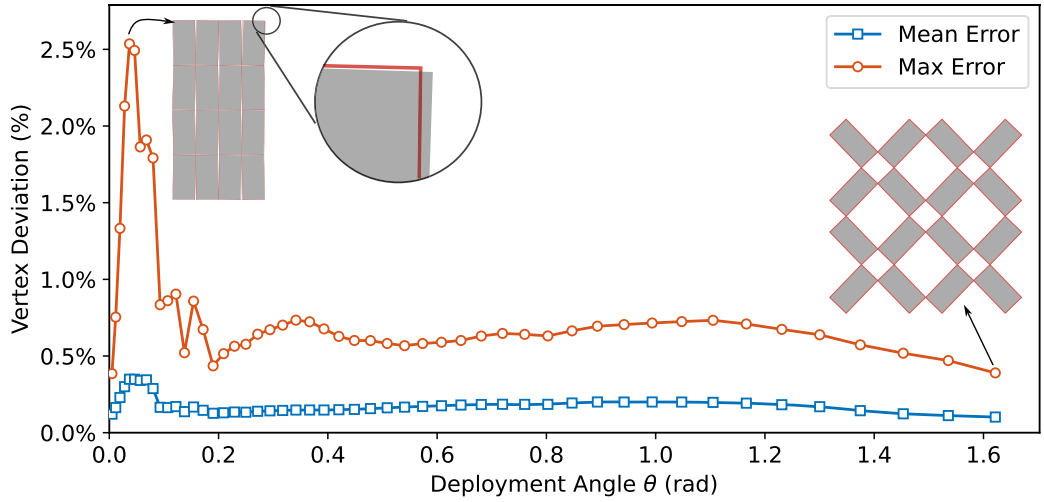

Fig. 4. Quantitative validation of PyKirigami using the rectangular tessellation kirigami structure. The plot shows the maximal and mean vertex error versus the deployment angle θ. Insets show a comparison of the simulated geometry (gray solid) against the theoretical trajectory (red wireframe) at two points. The left inset shows tessellation at the maximum deviation ($\theta \approx 0.04$) with the magnified view highlighting the drift. The right inset displays the fully deployed state, in which the error is negligible. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

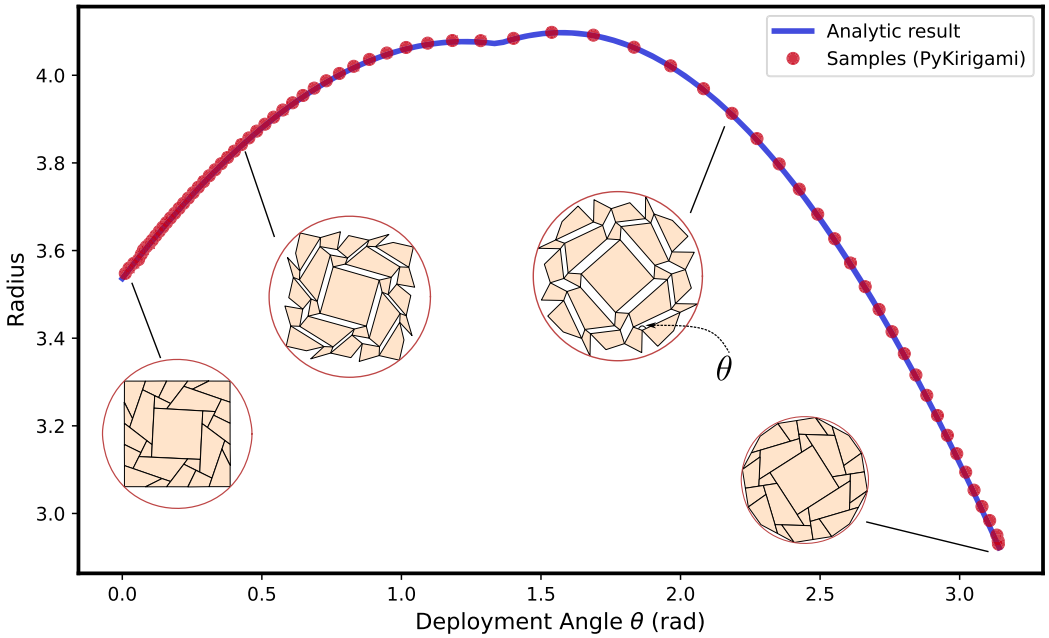

Fig. 5. Quantitative validation of PyKirigami using the square-to-disk kirigami structure. Here, we plot the global radius R versus the deployment angle θ for both the simulations obtained by PyKirigami (red dots) and the analytical result by [27] (blue solid line). The PyKirigami simulation data align closely with the analytical prediction, confirming that the solver captures the correct kinematic mode. The insets show four snapshots from the simulation. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

in [27], from which we can see that the discrete data samples obtained by PyKirigami at different time points of the deployment (red circles) show excellent agreement with the theoretical curve $R(\theta)$ (blue solid line). While micro-deviations exist at the hinge level (drift), the global topology remains locked to the correct deployment path, validating the tool for macroscopic shape analysis.

3.3. Sensitivity analysis and parameter tuning

To validate the force model in Algorithm 2 and guide parameter selection for users of PyKirigami, we evaluate the influence of three key parameters: linear spring stiffness k_f, force damping μ_v, and brick thickness h. Two representative models are considered, namely the square-to-disk model (focusing on 2D deployment) and the square-to-spherical-cap model (focusing on 3D deployment). Deployment quality is measured by the maximum vertex error, i.e. the largest Euclidean distance between any simulated vertex and its target position, monitored throughout the simulation. Unless otherwise stated, all parameters take the default values listed in Table B.3.

Spring stiffness. The stiffnesses k_f and k_τ govern the magnitude of the translational and torsional actuation forces respectively. Sweeping $k_f \in \{0, 25, 50, 100, 200\}$ at fixed $k_\tau = 100$ reveals a qualitative difference between 2D and 3D deployments (Fig. 6**a,b**).

For the planar square-to-disk model (Fig. 6**a**), the dominant deployment motion is in-plane rotation of each tile about its geometric center. The linear spring force $k_f(\tilde{v}_{i,j} - v_{i,j})$ pulls each vertex directly toward its target along a chord, whereas the physical deployment path traces a

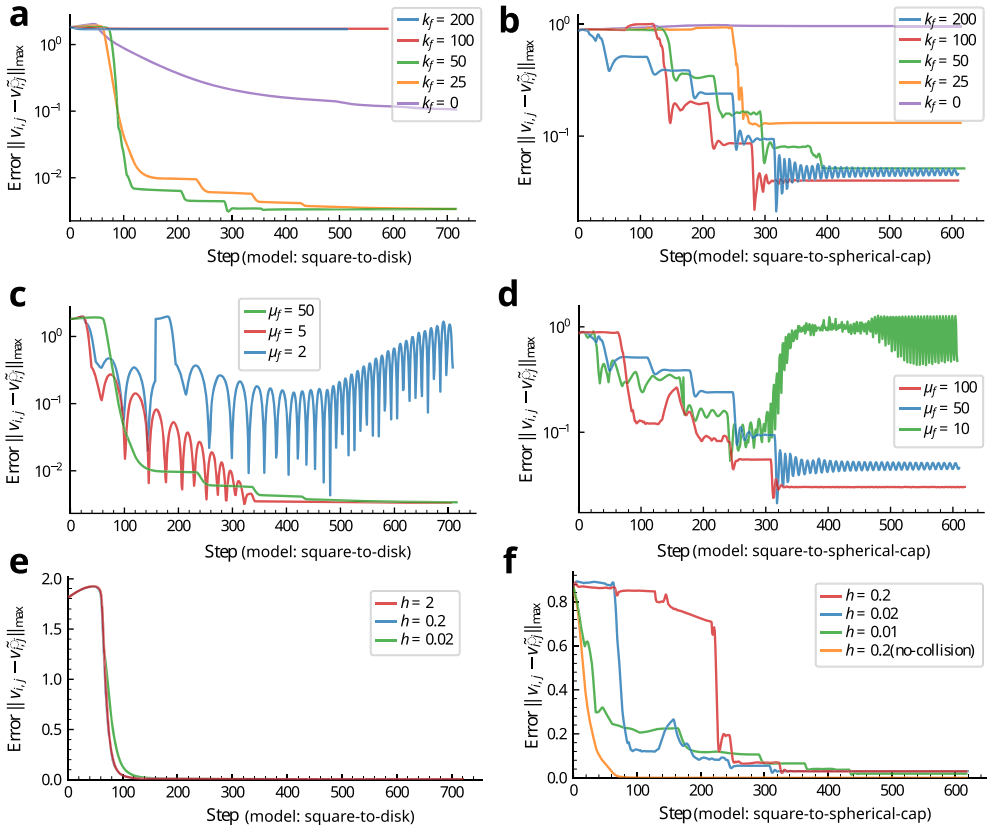

Fig. 6. Sensitivity of deployment quality to key simulation parameters on two representative models. The left column (**a,c,e**) used the square-to-disk model with default thickness $h = 0.2$; the right column(**b,d,f**) used the square-to-spherical-cap model with default thickness $h = 0.02$. In each row, one parameter is varied. **a–b**, Linear spring stiffness $k_f \in \{0, 25, 50, 100, 200\}$ at fixed $k_\tau = 100$. **c–d**, force damping μ_f, swept at the respective stiffness $k_f = 25$ and $k_f = 200$. **e–f**, Brick thickness h.

rotational arc. Therefore, linear spring stiffness competes with required rotation rather than assisting it in the beginning, and the structure stalls at an intermediate state with the default linear spring stiffness $k_f = 100$. The torsional-only case ($k_f = 0$) avoids this issue, since k_τ generates torque directly aligned with the rotational degree of freedom. However, without any translational force, the structure would stall at a residual error of $\mathcal{O}(10^{-1})$ due to the existence of ground friction and damping effects. A moderate linear spring $k_f \in \{25, 50\}$ balances both effects: it is small enough not to cancel out the required rotation in the beginning phase while still providing positional correction to drive the final error below $\mathcal{O}(10^{-2})$.

For the 2D-to-3D spherical-cap deployment (Fig. 6**b**), torsional actuation along $k_f = 0$ cannot drive tiles off the initial plane, so an extra linear spring k_f is necessary. At $k_f = 200$, visible oscillation appears. As discussed below, this reflects a damping mismatch. In both subplots, the staircase pattern "a plateau followed by an apparent error drop" is the signature of the adaptive stiffness mechanism: whenever deployments stall and maximal error is below the given tolerance, k_f and k_τ are doubled until convergence resumes.

Translation damping. Insufficient damping may cause oscillation or divergence when the spring force is large relative to the tile's inertia. Using the stiffness values identified above ($k_f = 25$ for the square-to-disk; $k_f = 200$ for the spherical cap), we sweep μ_v on each model (Fig. 6**c,d**). For the square-to-disk (Fig. 6**c**), reducing μ_v from 50 to 5 introduces oscillation near stiffness-doubling events but ultimately converges; $\mu_v = 2$ produces uncontrolled fluctuation from which the structure cannot recover. For the spherical cap (Fig. 6**d**), the required damping is correspondingly higher: $\mu_v = 50$ already fluctuates at $k_f = 200$, which is stabilized at $\mu_v = 100$, while $\mu_v = 10$ leads to divergence. These results confirm the expected coupling between k_f and μ_f: as stiffness increases, the required damping must increase proportionally to maintain a stable regime.

Thickness. Tile thickness h affects deployment through the physical extent of the collision geometry. For the purely planar rigidly deployable model like square-to-disk (Fig. 6**e**), sweeping $h \in \{0.02, 0.2, 2\}$ produces nearly identical error curves. For the 2D-to-3D deployment (Fig. 6**f**), thickness becomes a controlling factor. Due to the non-rigid property of the initial planar pattern, the structure must first buckle with non-planar voids before converging to the target (see Appendix C.8). The staircase plateaus in the plot correspond to this waiting period: the linear spring and torsional spring accumulate an imbalanced force distribution across the tile network, exploiting the kirigami structure to push some tiles upward and others downward until the collision geometry permits separation. The same pattern with larger thickness needs a spring with larger stiffness to be deployed. In contrast, the deployment with thickness $h = 0.2$ when penetration between tiles is allowed is smooth, yielding a monotonically smooth convergence curve.

3.4. Comparison with prior kirigami deployment simulator

It is natural to compare our method with other open-source software for kirigami deployment simulation. Note that the prior 2D kirigami deployment simulator in [16] utilizes the 2D rigid body physics engine Pymunk, focusing exclusively on 2D kirigami deployment. Also, the simulator only supports 2D manual interactive deployments and automatic radial deployments. By contrast, PyKirigami simulates both 2D and 3D kirigami deployment using the 3D PyBullet engine, and supports both 3D manual interactive deployments and more flexible target-based

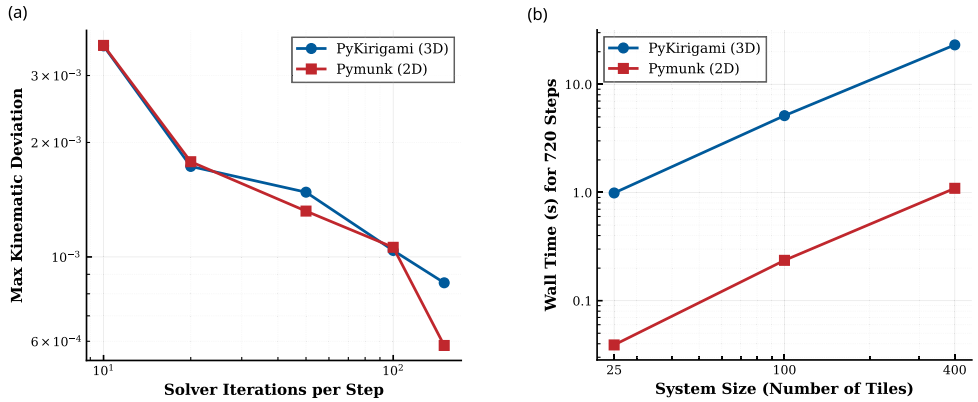

Fig. 7. Quantitative comparison between PyKirigami and Pymunk. a, Convergence of maximum kinematic deviation as a function of constraint solver iterations for a 10×10 grid where each grid is a unit square. Both engines exhibit monotonic decay, successfully reaching high-fidelity accuracy ($\mathcal{O}(10^{-4})$). **b**, Runtime scaling as a function of system size over a 720-step deployment at a fixed 100 solver iterations. Both simulators exhibit sub-quadratic $\mathcal{O}(N^{1.2})$ algorithmic scaling. The constant vertical offset reflects the baseline computational cost of PyKirigami's 3D rigid-body formulation (6 DOF) compared to Pymunk's 2D planar formulation (3 DOF).

automatic deployments. Therefore, to provide a systematic and quantitative comparison, here we can only focus on a common 2D deployment task using the standard rectangular tessellations that both simulators can handle and compare their performance. Also, because of the inherent differences between the two engines (e.g., PyBullet's Projected Gauss–Seidel solver vs. Pymunk's sequential impulse solver), one cannot synchronize all solver-specific parameters that control fundamentally different internal mechanisms, such as the Error Reduction Parameter (ERP) and `error_bias`, without introducing bias. Therefore, we focus on evaluating (1) the achievable accuracy of each engine against a shared analytical ground truth, and (2) the runtime scaling as a function of system size under matched kinematic conditions.

We simulate the deployment of a 2D rectangular tessellation in both PyKirigami and Pymunk over 720 steps ($\Delta t = 1/240\,\mathrm{s}$). To isolate the constraint solver's performance, gravity, friction, and damping are disabled, and collisions are bypassed. Fig. 7**a** illustrates the convergence behavior for a 10×10 grid. We evaluate the maximum kinematic deviation across all joints, as this represents the worst-case tolerance failure in mechanism design. As solver iterations increase, both engines exhibit a strict monotonic decay in maximum deviation, demonstrating that both architectures can achieve high-fidelity deployments ($\mathcal{O}(10^{-4})$ deviation) given sufficient computational budget.

To evaluate runtime scaling, we fix the solver iterations at 100 and measure the wall-clock time across varying grid resolutions (Fig. 7**b**). For running a complete 720-step deployment, PyKirigami typically takes only a few seconds for moderately sized patterns and also only takes slightly over 10 seconds for a large pattern with $20 \times 20 = 400$ tiles. In terms of comparison with Pymunk, note that both simulators exhibit highly efficient, sub-quadratic algorithmic scaling ($\sim \mathcal{O}(N^{1.2})$); a 16× increase in system size (from 5×5 to 20×20) results in a 23× increase in compute time for PyKirigami, and a 28× increase for Pymunk. In absolute time, one may see that Pymunk is faster than PyKirigami for equivalent grid sizes. This difference is fundamentally architectural: Pymunk solves strictly 2D planar dynamics (3 degrees of freedom per body), whereas PyKirigami utilizes a maximal-coordinate 3D formulation (6 degrees of freedom per body). This baseline computational overhead is the necessary and justified cost of PyKirigami's 3D architecture, which natively captures phenomena that 2D tools fundamentally cannot simulate, such as tile thickness, out-of-plane buckling, and complex 3D self-collisions.

4. Discussion

In this work, we have developed PyKirigami, an efficient and open-source computational tool for the deployment simulation of kirigami structures. Specifically, we have demonstrated the ability of our PyKirigami simulator for handling different 2D and 3D kirigami structures with different customized settings in terms of the tile geometries, tile connections, and other deployment parameters. Altogether, the significantly expanded functionalities of PyKirigami over prior open-source kirigami simulation software facilitate the computational modeling process for kirigami structures, thereby paving a new way for the design of shape-morphing metamaterials.

While PyKirigami provides an useful framework for rigid-body dynamics, we acknowledge its current limitations. First, note that the simulator considers the 3D kirigami tiles and their connections without capturing material elasticity, plasticity, or fracture. Also, high-fidelity stress/strain analysis would still require FEA methods. Besides, the current implementation of the simulator assumes the preservation of the kirigami tile connections throughout the deployment, while real-time cutting or topology changes are not supported. Our simulator is therefore best positioned as a tool for rapid prototyping, kinematic analysis, and exploring the global deployment dynamics of complex kirigami assemblies. For instance, it may be utilized for rapidly simulating the deployment effects and trajectories of kirigami structures under large-scale parametric sweeps across different deployment ranges, collision constraints, and morphing configurations, thereby forming a dedicated kinematic pre-screening pipeline for kirigami design.

In the future, we plan to enhance the functionalities of PyKirigami to include the consideration of elasticity and contact-induced deformations. Specifically, besides rigid-body dynamics, the underlying PyBullet engine also supports certain soft-body and deformable-object simulations, though they are known to be prone to numerical instability under large timesteps and body collisions. By carefully adjusting the relevant parameter setups, it may be possible to integrate these simulations into PyKirigami to allow for a wider range of kirigami deployment effects. Another future direction is to develop PyKirigami into an interactive website similar to Origami Simulator [49], making the software tool more user-friendly and accessible.

5. Video captions

Supplementary Video 1: The 2D-to-2D deployment of the square-to-disk kirigami model achieved by PyKirigami.

Supplementary Video 2: The 2D-to-3D deployment of the square-to-spherical-cap kirigami model achieved by PyKirigami.

Supplementary Video 3: The 3D-to-3D deployment and reconfiguration of the cylinder kirigami model achieved by PyKirigami.

Supplementary Video 4: The 3D-to-3D deployment of the cube-to-sphere kirigami model achieved by PyKirigami.

CRediT authorship contribution statement

Qinghai Jiang: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Formal analysis, Data curation, Conceptualization; **Gary P.T. Choi:** Writing - review & editing, Writing - original draft, Supervision, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Conceptualization.

Funding

This work was supported in part by the CUHK Faculty of Science Direct Grant for Research, grant no. 4,053,710 (to G.P.T.C.).

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The code and data are available on GitHub at https://github.com/andy-qhjiang/PyKirigami.

Supplementary material

Supplementary material associated with this article can be found in the online version at 10.1016/j.cpc.2026.110349.

Appendix A. Overview

PyKirigami is an open-source Python toolbox for simulating the 2D and 3D deployment of general kirigami structures. The simulator utilizes the PyBullet physics engine and is capable of simulating 2D-to-2D, 2D-to-3D, and 3D-to-3D kirigami deployments with interactive controls. The toolbox is freely available on GitHub: https://github.com/andy-qhjiang/PyKirigami

The detailed installation instructions, usages, and example projects can be found in the user manual: https://github.com/andy-qhjiang/PyKirigami/wiki

In this supplementary document, we focus on how we realize the key features and functionalities of PyKirigami to make it easy to be understood and extended by interested users.

B. Implementation walkthrough

B.1. Coordinate system management

A fundamental concept in PyBullet, and 3D simulations in general, is the distinction between **world coordinates** and **local coordinates**. Understanding this is key to efficiently defining connections and analyzing the kirigami structure's motion.

Frames and notation. We first define the following:

- World frame: The fixed global frame.
- Local (base) frames: one per tile, with origins at the tile centers O_1 (tile A) and O_2 (tile B).
- Rotations: $R_1, R_2 \in \mathrm{SO}(3)$ map local coordinates to world for tiles A and B.
- Coordinates: $\mathbf{p}_l$ and $\mathbf{q}_l$ are the local coordinates of P and Q; $\mathbf{p}_w$ and $\mathbf{q}_w$ are the corresponding world coordinates.

Local-world transforms. The exact mappings are

$$\mathbf{p}_w = R_1\,\mathbf{p}_l + O_1, \qquad \mathbf{q}_w = R_2\,\mathbf{q}_l + \mathrm{O}_2, \tag{B.1}$$

and, inversely,

$$\mathbf{p}_l = R_1^{\top}\,(\mathbf{p}_w - O_1), \qquad \mathbf{q}_l = R_2^{\top}\,(\mathbf{q}_w - \mathrm{O}_2). \tag{B.2}$$

See Fig. B.1 for an illustration. For convenience, we provide helper functions that implement the local-world transforms:

```
import pybullet as p
import numpy as np

def local_to_world(body_id, p_local):
    pos, orn = p.getBasePositionAndOrientation(body_id)
    # pos is the world frame position of tile center O_i
    R = np.array(p.getMatrixFromQuaternion(orn)).reshape(3, 3
                                        )  # R_i
    return (R @ np.asarray(p_local)) + np.asarray(pos)

def world_to_local(body_id, p_world):
    pos, orn = p.getBasePositionAndOrientation(body_id)
    R = np.array(p.getMatrixFromQuaternion(orn)).reshape(3, 3
                                        )
    return (R.T @ (np.asarray(p_world) - np.asarray(pos)))
```

B.2. Object creation in PyBullet

Given the planar vertices of a tile in world coordinates, we construct a 3D "brick" of thickness t. The core of this process is to create two distinct geometric representations for the tile, both defined in its local (base) frame: a simple **collision shape** for physics calculations and a detailed **visual shape** for rendering. This separation optimizes performance while ensuring visual fidelity.

Extrusion and face orientation. As described in the main text (see also main text Fig. 2), we define the tile's outward unit normal $\mathbf{n}$ in world coordinates and extrude along $+\mathbf{n}$ to obtain the top face, while the original tile becomes the bottom face. In Fig. B.2, we further illustrate this idea for 3D kirigami structures. Specifically, this local-normal convention is essential in 3D-to-3D deployments, where the world z-axis generally does not align with the tile's normal.

Collision shape construction. The collision shape is used by the physics engine to detect contact, resolve forces, and simulate dynamics. For stability and speed, we set it as a simple convex volume since this already meets our requirements for the simulation. After extrusion, we pass the list of local coordinates

$$\{u_j = v_j - O, u_j^t = v_j^t - O\}$$

to `p.createCollisionShape`. PyBullet automatically computes the convex hull of the vertices to form an efficient, though invisible, collision mesh (Fig. B.3). Below is an example of the collision shape creation for a triangular tile:

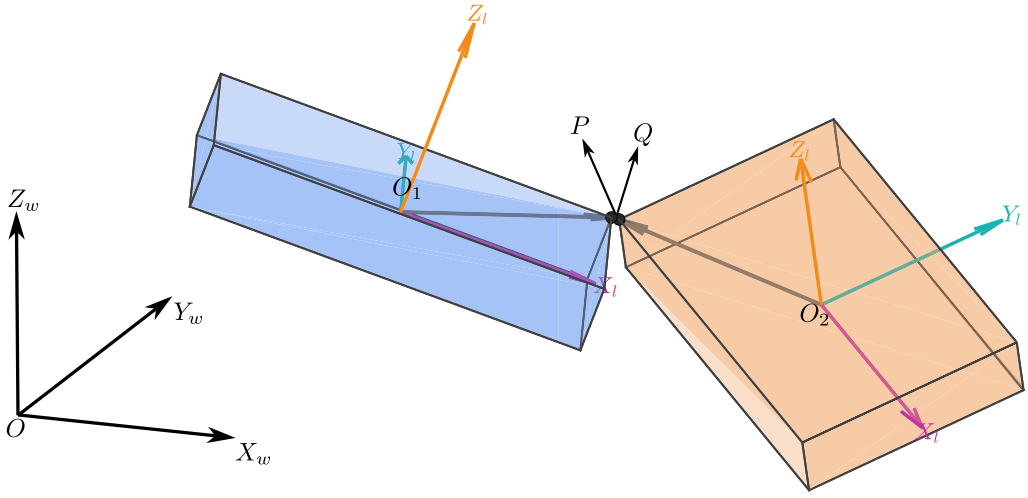

Fig. B.1. Coordinate systems in PyBullet. The fixed global axes (X_w, Y_w, Z_w) define the world coordinate system. Each kirigami tile (represented by a box) has its own local coordinate system (X_l, Y_l, Z_l) attached to its center. A corner vertex of the blue box, such as P, has constant local coordinates $\mathbf{p}_l = P - O_1$ in its body's frame, but its world coordinates $\mathbf{p}_w$ change as the body moves. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

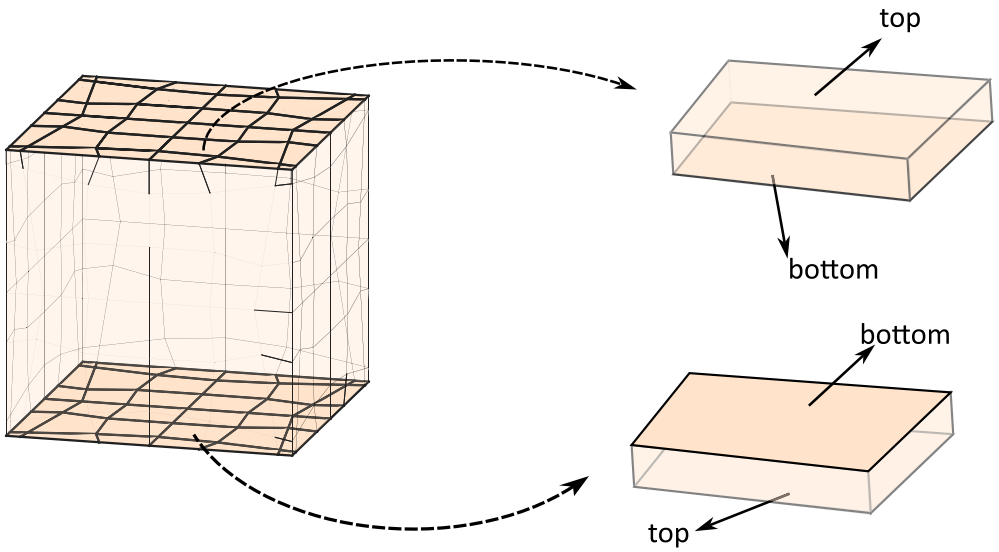

Fig. B.2. An illustration of face extrusion for 3D kirigami structures. Here, note that the quadrilateral patches on opposite faces of the cube are congruent, but their orientations are different. As the outward normal flips ($\mathbf{n} \mapsto -\mathbf{n}$), the vertex order must be reversed to maintain outward-facing normals. If the top face uses (v_0, v_1, v_2, v_3) in counter-clockwise order (viewed from outside), the corresponding bottom face should be indexed (v_0, v_3, v_2, v_1).

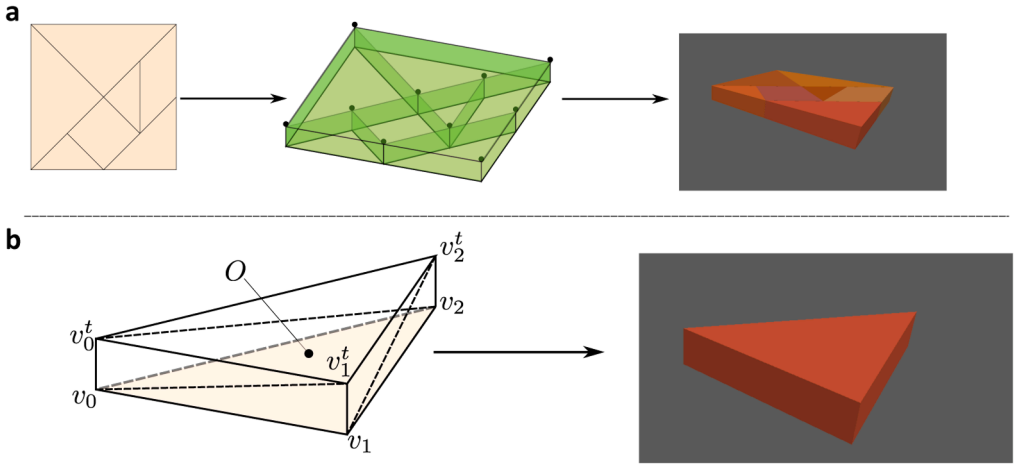

Fig. B.3. Brick creation in the local frame. a, Extrusion: We compute the world-space top vertices $v_j^t = v_j + t \cdot \mathbf{n}$ and the base origin O (tile center), then convert both bottom and top vertices to local coordinates via $u_j = v_j - O$ and $u_j^t = v_j^t - O$. **b**, Mesh construction: We build a convex collision vertex cloud (local) and a per-vertex-normal visual mesh (local) with cap and side faces.

```
    # PyBullet computes the convex hull based on local
                                    coordinates
    # which is exactly the triangular prism
    col_shape = p.createCollisionShape(
        shapeType=p.GEOM_MESH,
        vertices=[
                [x1, y1, z1], #  coordinates of u0
                [x2, y2, z2], #  coordinates of u1
                [x3, y3, z3], #  coordinates of u2
                [x4, y4, z4], #  coordinates of u0t
                [x5, y5, z5], #  coordinates of u1t
                [x6, y6, z6], #  coordinates of u2t
                ]
    )
```

Visual shape construction. The visual shape provides a detailed mesh for rendering sharp edges and correct lighting. The key to achieving this is understanding that PyBullet's graphics API assigns normals on a **per-vertex** basis. To create flat-shaded faces with hard edges, we must often duplicate

geometric vertices in our data buffers, pairing each copy with the correct face normal. We illustrate this by building a single rectangular side face of the brick illustrated in Fig. B.3**b**, defined by local bottom vertices (u_0, u_1) and top vertices (u_0^t, u_1^t). Below is an example:

```
# Define the 4 unique vertices of the rectangular face
vis_vertices = [
    u0,  # index 0
    u1,  # index 1
    u1t, # index 2
    u0t  # index 3
]

# The API requires one normal per vertex. Identical For a
                                flat face.
n_side = calculate_face_normal(v0, v1, v1t)
vis_normals = [n_side]*len(vis_vertices)

# Define two triangles using indices into the vertex/normal
                                lists
vis_indices = [
    0, 1, 2,  # First triangle: (v0, v1, v1t)
    0, 2, 3   # Second triangle: (v0, v1t, v0t)
]

# These lists are passed to create the visual component of
                                the body
vis_shape = p.createVisualShape(
    shapeType=p.GEOM_MESH,
    vertices=vis_vertices,
    indices=vis_indices,
    normals=vis_normals,
    rgbaColor=[...],
    specularColor=[0.8, 0.8, 0.8] # Glare; smaller is less
                                    shiny
)
```

Visual customization. The tile's appearance can be modified at creation time or dynamically via the `rgbaColor` parameter. We use sky blue color as an indicator of fixing an object during simulation:

```
p.changeVisualShape(bodyUniqueID=body_id, rgbaColor=[0.53, 0.
                                81, 0.92, 1.0])
```

Assembly. With both shapes defined in the local frame, instantiating the rigid body is straightforward. The only world quantity to be supplied is the base pose (O, R); typically $R = I$ at load time.

```
def create_brick_body(col_shape, vis_shape, base_origin_world
                                , mass=1.0, base_quat=[0,0,0,1
                                ]):
    return p.createMultiBody(
                baseMass=mass,
                baseCollisionShapeIndex=col_shape,
                baseVisualShapeIndex=vis_shape,
                basePosition=base_origin_world, # world
                                                position of
                                                local origin
                baseOrientation=base_quat # usually identity;
                                            optional
)
```

This completes the brick creation pipeline: (i) extrude and define local geometry, (ii) build a convex collision shape and a flat-shaded visual mesh with per-vertex normals via vertex duplication, and (iii) instantiate the rigid body at its world pose.

B.3. Creating inter-tile constraints

With each tile defined as a rigid body in its own local frame, we now connect them to form the kirigami structure. PyKirigami uses PyBullet's constraint system to model the joints between tiles. All constraints are defined by specifying pivot points in the **local coordinates** of the connected bodies, leveraging the framework established in the previous sections.

Constraint validation. Before creating any physics objects, it is crucial to validate that the provided vertex and constraint data are consistent. For a kirigami structure to be physically valid, the world-space positions of any two vertices designated as a pivot pair must coincide. PyKirigami performs this check before entering the simulation loop:

```
def validate_constraints(vertices, constraints, tolerance=1e-
                                6):
    # Unpack constraint data (e.g., for a spherical joint)
    for i, constraint in enumerate(constraints):
        f1, v1, f2, v2 = constraint[:4]
        # Get vertex positions for constraint endpoints
        p1_world = np.array(vertices[f1][3*v1:3*v1+3])
        p2_world = np.array(vertices[f2][3*v2:3*v2+3])
        # Check distance
        dist = np.linalg.norm(np.array(p1_world) - np.array(
                                        p2_world))
        if dist > tolerance:
            raise ValueError(f"Constraint validation failed:
                                         distance is {dist}
                                         ")
```

Spherical joint. As illustrated in Fig. B.1, a connection is defined by a pair of pivot points, such as P on tile A and Q on tile B. We provide PyBullet with the constant local coordinates of these pivots, $\mathbf{p}_l$ and $\mathbf{q}_l$. The physics engine then ensures that the world-space positions of these points, $\mathbf{p}_w$ and $\mathbf{q}_w$, satisfy the constraint equation

$$R_1 \cdot \mathbf{p}_l + O_1 = R_2 \cdot \mathbf{q}_l + O_2, \tag{B.3}$$

at every time step. It is implemented with a single `JOINT_POINT2POINT` constraint as follows:

```
# create spherical joint and return cid (constraint ID)
def connect_spherical(bodyA, bodyB, pA_local, pB_local):
    return p.createConstraint(
                    parentBodyUniqueId=bodyA,
                    parentLinkIndex=-1,
                    childBodyUniqueId=bodyB,
                    childLinkIndex=-1,
                    jointType=p.JOINT_POINT2POINT,
                    jointAxis=[0,0,0],
                    parentFramePosition=pA_local,
                    childFramePosition=pB_local
)
```

Since the kirigami bricks have thickness, we must specify whether a pivot lies on the bottom face (original vertices) or the top face (extruded vertices) as illustrated in the main text. Our constraint file format supports an optional face specifier `t` (1 for bottom, 2 for top). When parsing, we select the appropriate local vertex vector, either $\mathbf{u}_j$ (bottom) or $\mathbf{u}_j^t$ (top), to use as the pivot.

Hinge joint. A hinge restricts rotation to a single axis, providing one rotational degree of freedom. In PyKirigami, we implement a hinge by using **two** non-coincident spherical joints. The line connecting the two pivot pairs defines the hinge axis. This method is numerically stable and works well for bodies with finite thickness:

```
def connect_hinge(bodyA, bodyB, pA1, pB1, pA2, pB2):
    cid1 = connect_spherical(bodyA, bodyB, pA1, pB1)
    cid2 = connect_spherical(bodyA, bodyB, pA2, pB2)
    return [cid1, cid2]
```

Fixed joint. This joint removes all six relative degrees of freedom between two bodies, effectively welding them together. We use the native `JOINT_FIXED` constraint. This is particularly useful for anchoring a tile to the world (by constraining it to a static body) or for creating rigid composite structures:

```
def fix_object_to_world(body_id):
    pos, orn = p.getBasePositionAndOrientation(body_id)
    constraint_id = p.createConstraint(
        parentBodyUniqueId=body_id,
        parentLinkIndex=-1,
        childBodyUniqueId=-1,  # World frame
        childLinkIndex=-1,
        jointType=p.JOINT_FIXED,
        jointAxis=[0, 0, 0],
        parentFramePosition=[0, 0, 0],
        childFramePosition=pos,  # tile center O_i in world
        parentFrameOrientation=[0, 0, 0, 1],
        childFrameOrientation=orn    # tile orientation in
                                       world
    )
    p.changeConstraint(cid, maxForce=max_force)
    return cid
```

Remark. To make a body immovable, one could either use a `JOINT_FIXED` constraint or set the body's mass to zero. For an articulated system like a kirigami structure, the constraint-based approach is recommended since an anchored tile must still be able to transmit forces and torques from the rest of the structure to the world. A fixed joint allows for this, as the body remains a dynamic participant in the simulation. In contrast, setting a body's mass to zero removes it from the dynamics calculation, which can cause solver instability when other dynamic bodies are joined to it. The practice of setting mass to zero is best reserved for defining permanent, static environment geometry like floors or immovable obstacles.

Automatic connection-type detection. When a target file is provided, our PyKirigami tool can automatically determine whether to connect the bottom or top face of each tile pair (controlled by the `--auto_detect_connections` flag). The decision is made by analyzing the target geometry:

- **Shared-edge case.** If two tiles share an edge in the target, the signed dihedral angle δ is computed via $\delta = \operatorname{atan2}((\mathbf{n}_a \times \mathbf{n}_b) \cdot \mathbf{e},\ \mathbf{n}_a \cdot \mathbf{n}_b)$, where $\mathbf{n}_a, \mathbf{n}_b$ are face normals and $\mathbf{e}$ is the shared edge direction. $\delta < \pi$ indicates a "valley fold" (top connection); $\delta \geq \pi$ indicates a "mountain fold" (bottom connection).
- **Negative-space case.** If tiles meet only at a single vertex, the face normals are projected onto the inter-centroid vector. Converging normals indicate a "valley fold" (top connection); diverging normals indicate a "mountain fold" (bottom connection); neutral cases default to bottom.
- **Planar-to-planar case.** If both initial and target configurations are planar, both faces are connected (type 3) for stability.

B.4. Interactive control and simulation state management

The runtime system comprises three layers:

- Entry point and GUI loop (run_sim.py)
- Simulation (initialization and force selection)
- Controllers: SimulationController (runtime stepping/state) and InteractionController (pin/unpin, snapshots, and save data)

Simulation: initialization and force selection. The Simulation class constructs all rigid bodies and constraints from files, configures damping/environment, and prepares data for runtime control. It returns a dictionary `simulation_data` for downstream controllers, summarized in Table B.1.

SimulationController: runtime state and stepping. Two controllers drive the runtime behavior, each consuming the entries in `simulation_data`:

- Stepping: When `step_simulation()` is called, if the simulation is not paused, it first calls the `apply_forces()` function (which selects the appropriate actuation model) and then calls `p.stepSimulation()`.
- State Management: It handles `reset_simulation()` by calling the function `initialize_ simulation()` to rebuild the world from scratch.

Table B.1
Contents of the `simulation_data` dictionary passed to controllers.

Key	Description
`args`	Parsed command-line configuration (timestep, damping, thickness, modes, etc.).
`bricks`	List of PyBullet body IDs (one per tile).
`local_coords`	Bottom-face local vertex coordinates per tile (used for local-to-world transforms during runtime).
`constraint_ids`	List of PyBullet constraint IDs created between tiles.
`visual_mesh`	List of visual mesh for each tile to be exported as an OBJ file.
`target_vertices`	Optional world-space target vertices per tile (bottom face), used for target-based actuation.

InteractionController: user input and visual feedback. The InteractionController handles real-time user input from the mouse and provides visual feedback. It also uses `simulation_data` to identify which objects are part of the kirigami structure.

- Mouse Events: In `process_mouse_events()`, it performs a ray cast from the camera through the mouse cursor. If the ray hits an object whose ID is in the bricks list from `simulation_data`, it triggers the pin/unpin action.
- Data Output: An OBJ file is exported.

Main loop. The entry point `run_sim.py` wires everything together and runs an interactive loop. The core structure is given in Algorithm 3:

Algorithm 3: Interactive main loop.

```
Initialize physics, Simulation, SimulationController, InteractionController
while Bullet client is connected do
    keys ← getKeyboardEvents()
    for each (key, handler) in handlers do
        if key was triggered then
            call handler()
            if key == 'q' then
                break loop
            end if
        end if
    end for
    InteractionController.process_mouse_events()
    SimulationController.step_simulation()
    sleep(timestep)
end while
disconnect()
```

For completeness and reproducibility, we summarize the default interactive controls used in PyKirigami's GUI mode. These inputs allow manual actuation, dynamic boundary condition changes, camera manipulation, and on-demand export. Key bindings can be customized in the codebase; the mappings shown in Table B.2 reflect the default configuration used in this paper.

B.5. Physical environment construction: CLI parameters and defaults

This section documents the primary command-line interface (CLI) flags used to configure and run the simulations. The table below groups related parameters and lists their default values and usage. For simplicity and portability, input files can be organised into a single directory within the `data/` folder and loaded using the `--model` flag. The CLI parameters and default values are given in Table B.3.

B.6. Self-penetration avoidance

PyKirigami tries to avoid self-penetration during the deployment simulation through several complementary mechanisms, operating at different levels of the simulation pipeline:

1. **Physical-engine-level collision detection (per time step).** The most fundamental layer of protection is provided by PyBullet's built-in rigid body collision detection, which runs at every simulation step. When two tiles approach within a configurable collision margin ϵ, the physical engine applies a corrective impulse to prevent interpenetration. This ensures that, even if actuation forces attempt to drive two tiles into the same spatial region, the solver will resist penetration up to numerical tolerance. Notice that a small residual overlap (on the order of the collision margin ϵ, typically $\sim 1\%$ of tile size) is unavoidable, which is standard in rigid-body simulation.
2. **Automatic connection-type detection at the tile level.** Potential penetration between tiles exists due to the extrusion of planar polygons. For example, the boundary of the top face of both tiles would collide if both tiles are driven to a valley shape while they are connected by a bottom joint. Therefore, we provide users with an option for PyKirigami to automatically classify each inter-tile connection based on the target vertices, ensuring that the structure's connectivity topology is consistent with the actuation forces applied to each tile.

Table B.2
Interactive control scheme for the PyKirigami simulator.

Control Input	Functionality
Left-click + drag	Apply external, damped forces to tiles for manual guidance and actuation during deployment.
Right-click	Toggle fixed-world constraints on a selected tile.
`Space bar`	Toggle pause/resume the physics simulation.
`F`	Toggle automatic deployment forces on/off.
`R`	Reset the simulation to its initial configuration.
`O`	Export the result as an OBJ file.
`Q`	Quit the simulation application.
Scroll / Ctrl + drag	Zoom, rotate, and pan the camera for detailed visual inspection.

Table B.3
Command-Line Interface (CLI) parameters and default values. Parameter groups are distinguished by alternating background colors.

Flag	Default	Description
Model and File Configuration		
`--model`	(none)	Specifies a model folder `data/<name>/` containing input files like `vertices.txt`, `constraints.txt` and optional `target.txt`.
Simulation Control and Stepping		
`--timestep`	1/240	Discrete virtual time increment for solving rigid-body dynamic inside PyBullet
Physical Environment		
`--gravity`	0	Gravitational acceleration in $-z$.
`--ground_plane`	(switch)	Adds ground plane to the simulation.
`--filter_collision`	(switch)	Cancel collision detection during simulation.
Actuation Model		
`--cm_expansion`	(switch)	Enables center-of-mass radial expansion force model.
`--spring_stiffness`	100	Linear spring stiffness k_s for target-based forces.
`--torque_stiffness`	100	Torsional spring stiffness k_τ for target-based torques.
`--force_damping`	50	Actuator (viscous) damping μ_v for target-based forces.
`--torque_damping`	2	Actuator (viscous) damping μ_ω for target-based torques.
Geometry and Visualization		
`--brick_thickness`	0.02	Thickness of extruded tiles (m).

3. **Kabsch alignment for global correspondence.** A more subtle source of self-intersection of kirigami patterns during automatic deployment arises from incorrect global target assignment. If the target vertex positions are naively matched to source tiles without accounting for global orientation, a tile on the left side of the pattern may be assigned a target on the right side, and vice versa. The resulting crossed actuation force would drive different parts of the structure through each other, causing catastrophic self-intersection that collision detection alone cannot prevent. PyKirigami resolves this by applying the Kabsch algorithm to compute the optimal rigid-body alignment between all points from vertices file and target file before applying target-based force. This can avoid crossing trajectories to some extent.
4. **User-defined intermediate targets for complex deployment.** For highly complex target shape changes or deployment trajectories, we suggest decomposing the deployment into a sequence of simpler sub-deployments, analogous to waypoint-based motion in robotics. We have functions integrated in PyKirigami that export the current position of the structure and convert it to the same format as target vertices, so that the user can decompose the deployment into pieces by selecting the original pattern and target pattern themselves.

Appendix C. Example configurations and simulation details

In the main text, we presented several 2D-to-2D, 2D-to-3D, and 3D-to-3D deployments obtained with PyKirigami. Here we provide additional snapshots and, crucially, the corresponding physical environment and solver settings used for each case so that readers can reproduce the results precisely. All CLI flags follow Table B.3. All simulations were run with a fixed timestep $\Delta t = 1/240\,\mathrm{s}$ and $n_{\mathrm{sub}} = 20$ internal substeps unless otherwise noted.

C.1. Example: 2D-to-2D square-to-circle kirigami model

The 2D-to-2D square-to-circle kirigami model example (Fig. C.4; see also main text Fig. 3**a**) serves as an excellent test case for a 2D radial expansion because of the pattern symmetry. The primary challenge is to enforce planar motion and prevent the structure from buckling out-of-plane. Our strategy combines hinge kinematics with specific environmental and geometric settings, driven by an automated radial force model.

Key simulation parameters. A stable, planar, and automated deployment is achieved with the following configuration:

- **Kinematic Model:** We use **hinge joints** to connect the tiles. This is the most important choice, as it constrains tile rotation to the Z-axis, fundamentally preventing out-of-plane tumbling.
- **Actuation Model:** The deployment is driven by the automated Center-of-Mass Expansion model (`--cm_expansion`). No target file is needed.
- **Physical Environment:** `--gravity -10`, `--ground_plane`
- **Geometry:** `--brick_thickness 0.2`. This provides the tiles with rotational inertia to resist torsional instabilities.

C.2. Example: 2D-to-2D square-to-disk kirigami model (interactive workflow)

Fig. C.5 demonstrates the fully interactive workflow for the 2D-to-2D compact reconfigurable square-to-disk model shown in main text Fig. 3**b**. While the main text analyzes this structure's automated deployment, here we show how to achieve the same reconfiguration manually using the GUI controls detailed in Table B.2. This process highlights the use of the simulator for hands-on exploration and state capture.

With the simulation running in a no-force mode, a user can manually reconfigure the structure from the "square" to the "disk" state. This is achieved by applying the interactive controls from Table B.2 in a strategic sequence, as illustrated in Fig. C.5.

Key simulation parameters:.

- **No Scripted Actuation:** To enable manual control, we launch the simulation without any automated force models. This is done by omitting the `--cm_expansion` flag and ensuring no `--target_vertices_file` is specified.
- **Physical Environment:** `--gravity -10` `--ground_plane`
- **Visualization** `--brick_thickness 0.2`.

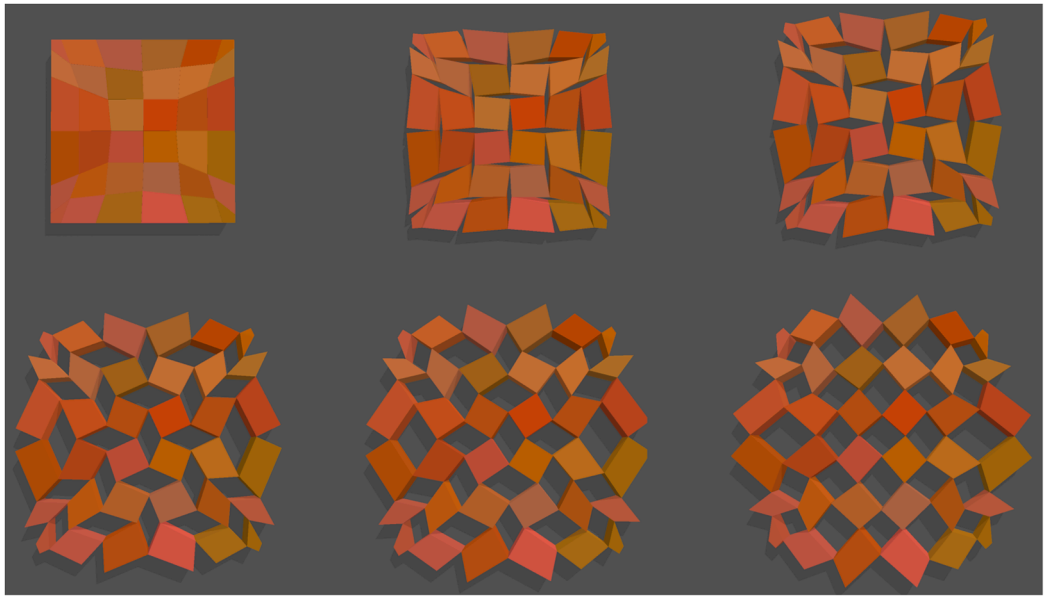

Fig. C.4. Additional snapshots of the 2D-to-2D deployment of the square-to-circle kirigami model achieved by PyKirigami.

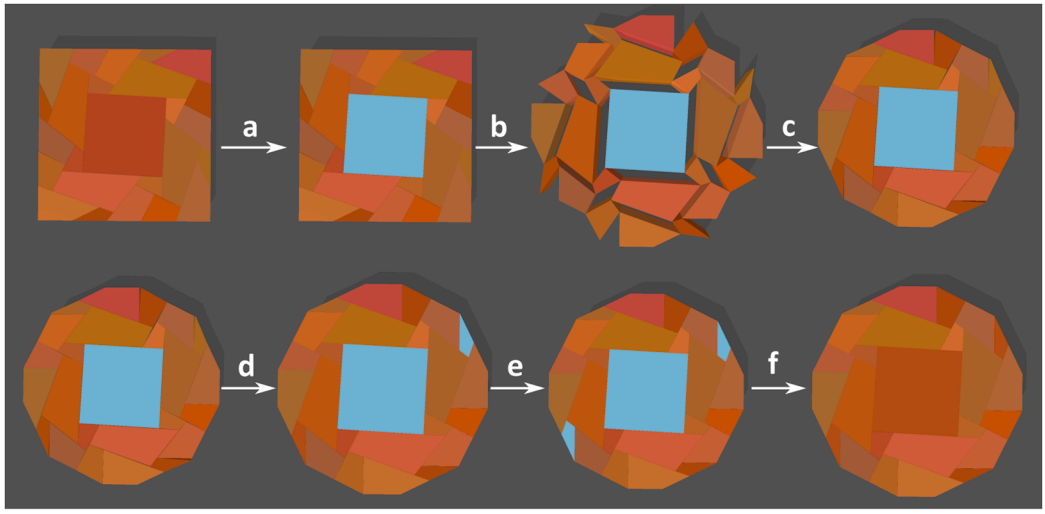

Fig. C.5. Interactive workflow for the manual reconfiguration of the square-to-disk kirigami model. a, A central tile is pinned (`Right-click`) to create a stable reference frame. **b**, An outer tile is dragged (`Left-click + drag`) to begin the radial expansion. **c**, The structure is guided into an approximate "disk" shape. At this stage, small gaps often persist due to the implicit hinge forces. **d**, To close a gap, a tile is dragged into the correct position through `Left-click + drag` and the simulation is then paused (`Space bar`), and the tile is pinned (`Right-click`) while paused. This "pause-and-pin" technique is a powerful feature, allowing a constraint to be applied to a desired geometry before the physics solver can react. **e**, Step **d** is repeated for other misaligned tiles. **f**, All pins are removed while paused, leaving the structure in a stable "disk" state.

C.3. Example: 2D-to-3D square-to-spherical-cap kirigami model

The square-to-spherical-cap kirigami model example (Fig. C.6; see also main text Fig. 3**c**) demonstrates a 2D-to-3D transformation from a flat sheet into a curved surface. This complex, out-of-plane motion is impossible to guide with simple radial forces and necessitates the use of the target-based actuation model for precise control. Here, additional snapshots showing the progression from a planar to a spherical configuration are provided. Also, one can notice that out-of-plane motion provides extra degrees of freedom, allowing the sheet to lift/twist to resolve incompatibilities which can be seen in [22].

Key simulation parameters:.

- **Kinematic Model:** We use **spherical joints** to connect the bottom face of the tiles, allowing for the necessary three-dimensional rotational freedom.
- **Actuation Model:** The deployment is driven by the **Target-Based model**, with a target file defining the final spherical geometry.
- **Key Actuation Parameters:** `--spring_stiffness 100`, `--force_damping 50`.
- **Physical Environment:** `--ground_plane`. It can serve as a background to visualize the shadow.
- **Geometry:** `--brick_thickness 0.02`. It is noteworthy that if the tiles are too thick, they will block the deployment.

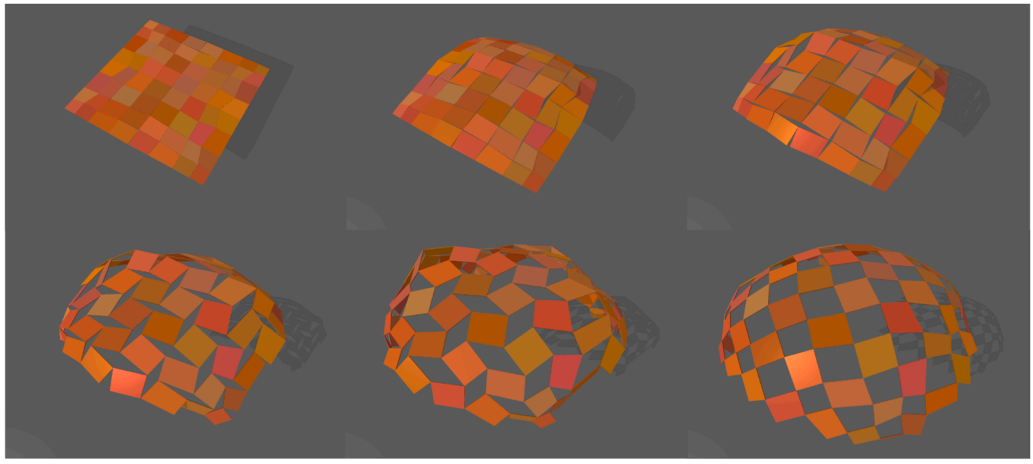

Fig. C.6. Additional snapshots of the 2D-to-3D deployment of the square-to-spherical-cap kirigami model achieved by PyKirigami, driven by the target-based force model.

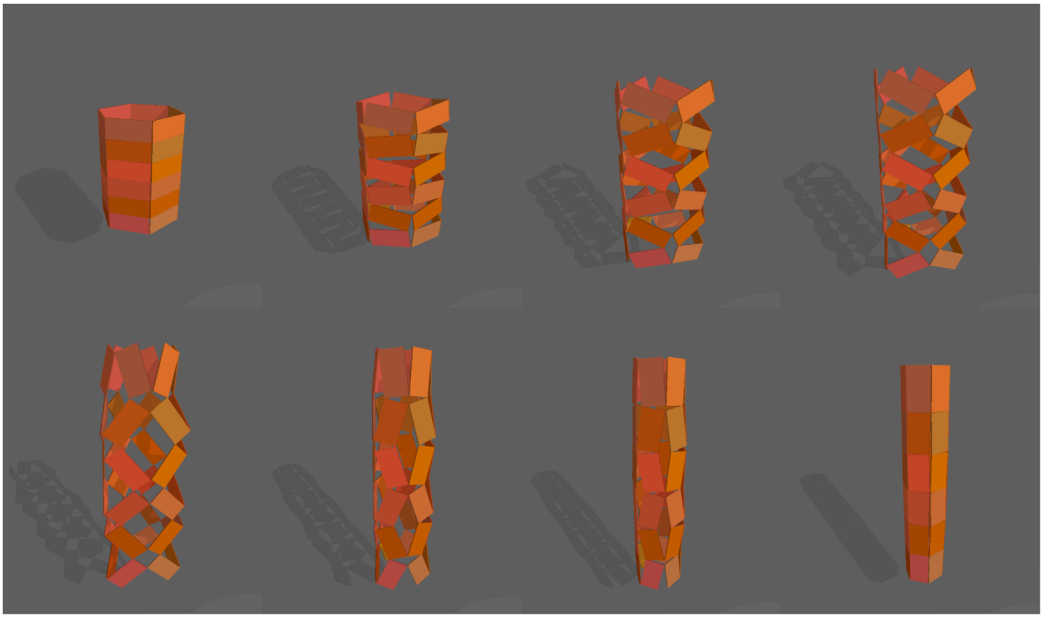

Fig. C.7. Additional snapshots of the reconfiguration process of the 3D-to-3D cylinder kirigami structure achieved by PyKirigami.

C.4. Example: 3D-to-3D compact reconfigurable cylinder kirigami model

The 3D-to-3D compact reconfigurable cylinder kirigami model example (Fig. C.7; see also main text Fig. 3**d**) involves the reconfiguration between two distinct 3D states, from a compact, folded cylinder to another compact, folded state. The target-based model is essential for guiding the structure through a collision-free volumetric transformation.

Key simulation parameters:.

- **Kinematic Model:** We use **spherical joints** to allow for full 3D motion between the tiles.
- **Actuation Model:** The reconfiguration is driven by the **Target-Based model**.
- **Key Actuation Parameters:** `--spring_stiffness 100`, `--force_damping 80`.
- **Physical Environment:** `--gravity 0`
- **Geometry:** `--brick_thickness 0.02`. Note that thinner tiles yield a better deployment.

C.5. Example: 2D-to-2D stampfli quasicrystal kirigami models

Note that the above kirigami structures primarily consist of quadrilateral tiles, and the overall structures exhibit a highly regular topology. To further demonstrate the applicability of PyKirigami to other kirigami structures, we consider simulating the deployment of two quasicrystal kirigami structures from [16] with highly different topologies.

More specifically, in Fig. C.8 we show the deployment of a 12-fold Stampfli quasicrystal kirigami structure created using the Hamiltonian construction method in [16]. Here, note that the structure contains not only quadrilateral tiles but also triangular tiles, and the tile connection is also

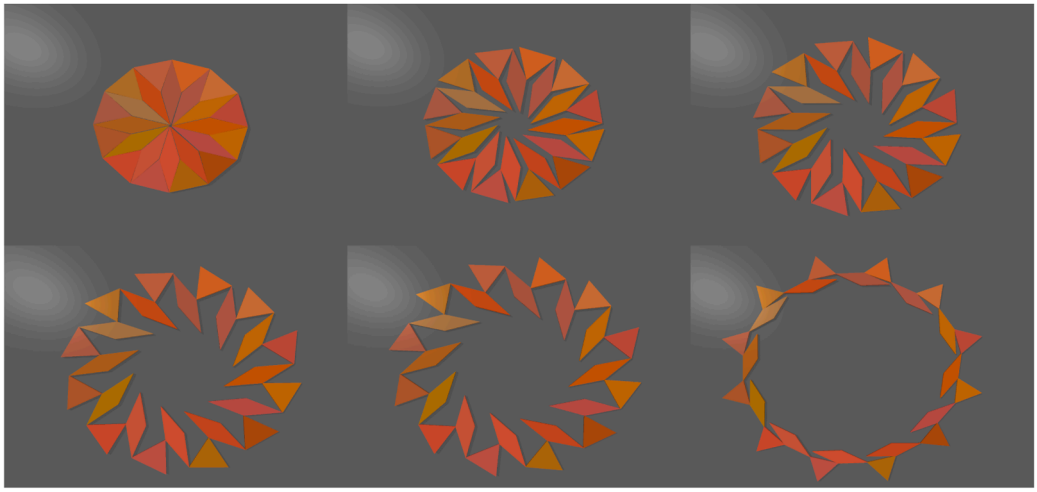

Fig. C.8. The simulated deployment of the 12-fold Stampfli quasicrystal kirigami structure from the Hamiltonian construction method in [16] achieved by PyKirigami with automated radial expansion, stabilized by environmental forces.

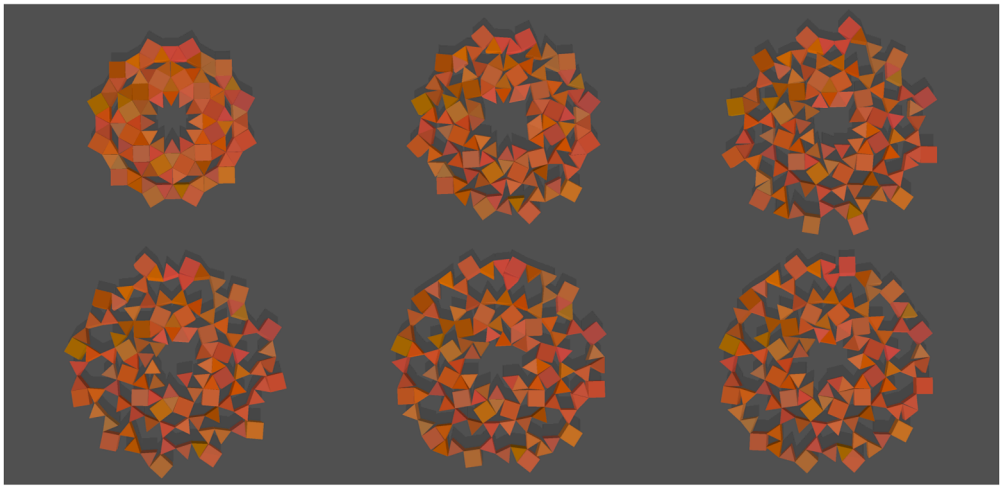

Fig. C.9. The simulated deployment of the 12-fold Stampfli quasicrystal kirigami structure from the Tile Removal construction method in [16] achieved by PyKirigami with automated radial expansion, stabilized by environmental forces.

highly different from the ones shown above, with all tiles forming one big closed loop. It can be observed that PyKirigami is capable of producing a deployment simulation of this structure with automated radial expansion.

In Fig. C.9, we consider another 12-fold Stampfli quasicrystal kirigami structure created using the Tile Removal construction method in [16], which consists of both square and triangular tiles but possesses a highly different topology. Specifically, it contains a large hole at the center as well as multiple holes in the surrounding regions with different sizes and shapes. The successful deployment achieved by PyKirigami showcases the capability of PyKirigami in handling kirigami structures with different topologies.

C.6. Example: 2D-to-2D compact reconfigurable fan kirigami model

To further demonstrate the versatility of PyKirigami, we present another example focusing on its interactive reconfiguration functionality. Fig. C.10 shows the compact reconfigurable fan kirigami model, which has two different closed and compact contracted states. Users can apply the mentioned interactive operations to deploy it under the same physical environment as the square-to-disk model as shown in Appendix C.2.

C.7. Example: 3D-to-3D cube-to-sphere kirigami model

Finally, we present an interesting 3D-to-3D transformation example using the cube-to-sphere kirigami model (Fig. C.11; see also main text Fig. 1**b**). For this model, the six faces of the cube are deployed into a spherical approximation throughout the deployment process, which requires large, coordinated out-of-plane rotations and precise positioning to ensure the edges meet correctly without self-intersection. Such a transformation is infeasible with simple or manual forces and serves as a powerful demonstration of the target-based actuation model's capabilities.

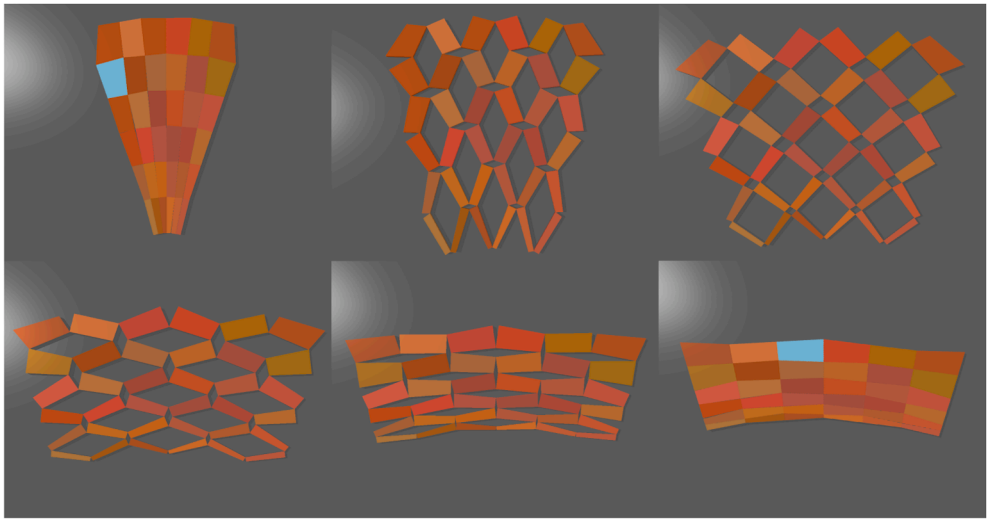

Fig. C.10. The simulated deployment of the compact reconfigurable fan kirigami model produced by PyKirigami with interactive shape morphing.

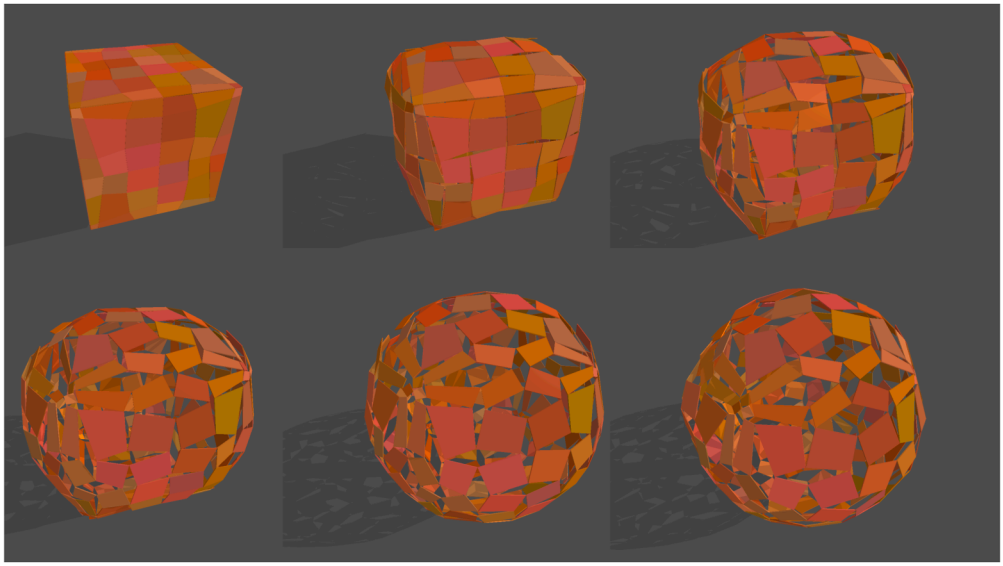

Fig. C.11. The simulated deployment of the cube-to-sphere kirigami model produced by PyKirigami. In this simulation, large, coordinated out-of-plane rotations and precise positioning are applied to control the deployment effect.

C.8. Handling non-rigidly deployable structures

While PyKirigami primarily considers kirigami tiles as 3D rigid bodies and focuses on rigid deployments, it can also be utilized for handling kirigami structures that are not rigidly deployable.

Specifically, to handle the deployment of non-rigidly deployable planar kirigami structures, one strategy is to utilize the additional degree of freedom provided in the spatial dimension. For instance, Fig. C.12 illustrates a simple 2D non-rigidly deployable kirigami structure consisting of four quadrilaterals. As the slit at the interior does not form a straight line, this pattern is not rigidly deployable in 2D theoretically [22]. Panel **a** shows that attempting to deploy this pattern in the plane leads to tile overlap, suggesting that a strictly planar deployment is geometrically impossible. Nevertheless, since PyKirigami handles kirigami structures in the 3D space, we can easily handle such cases by allowing the structure to move out of plane as shown in Panel **b**. In the second frame, the unit deploys in 3D space, where the additional degree of freedom resolves the geometric conflict. Crucially, the negative space (the gap between tiles) at this intermediate state is nonplanar, as the four tile centers do not lie in a single plane. This “nonplanar negative space” is a characteristic feature of 3D kirigami deployment and distinguishes it from purely 2D mechanisms. At the third frame, the structure is flattened back to the plane with negative spaces introduced, achieving a fully deployed planar configuration that was impossible in Panel **a**.

More generally, to model non-rigid deployments of kirigami structures in either 2D or 3D, the user may consider a subdivision of a tile into smaller units and connect them appropriately, analogous to the use of a triangulated origami mesh in prior non-rigid origami folding simulators [39–42]. For instance, the bending of a kirigami tile can be modeled by replacing the tile with two or multiple smaller tiles connected by hinge joints along the tile edges, which provides additional flexibility in simulating 3D kirigami deployment. Similarly, the stretching and deformation of a tile can be modeled by replacing it with multiple smaller tiles connected by joints, so that the smaller tiles together with the negative spaces formed during the deployment can serve as a representation of the tile stretching and deformation.

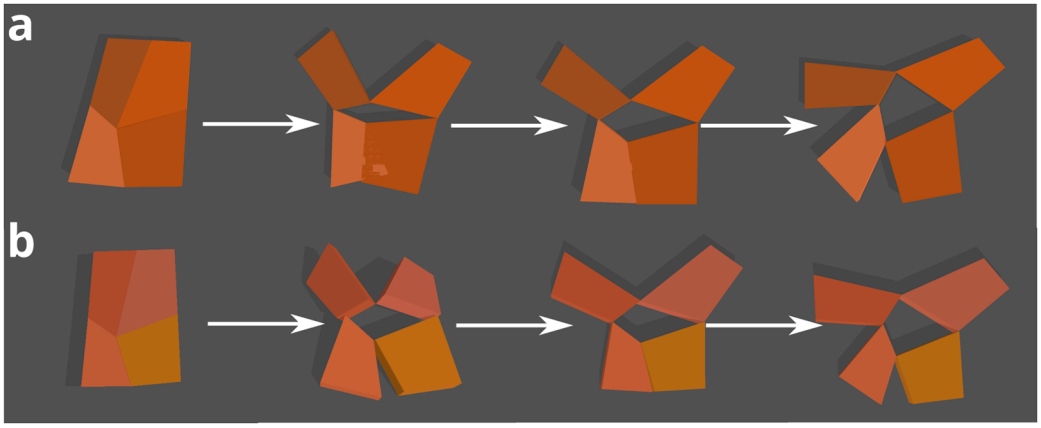

Fig. C.12. Deployment of a non-rigidly-deployable kirigami structure. a, Planar rigid deployment is geometrically impossible due to tile overlap. **b**, By utilizing the 3D deployment ability in PyKirigami, the incompatibility can be resolved: the unit deploys out of plane (second frame), creating nonplanar negative space, then flattens back with the introduced gaps (third frame).

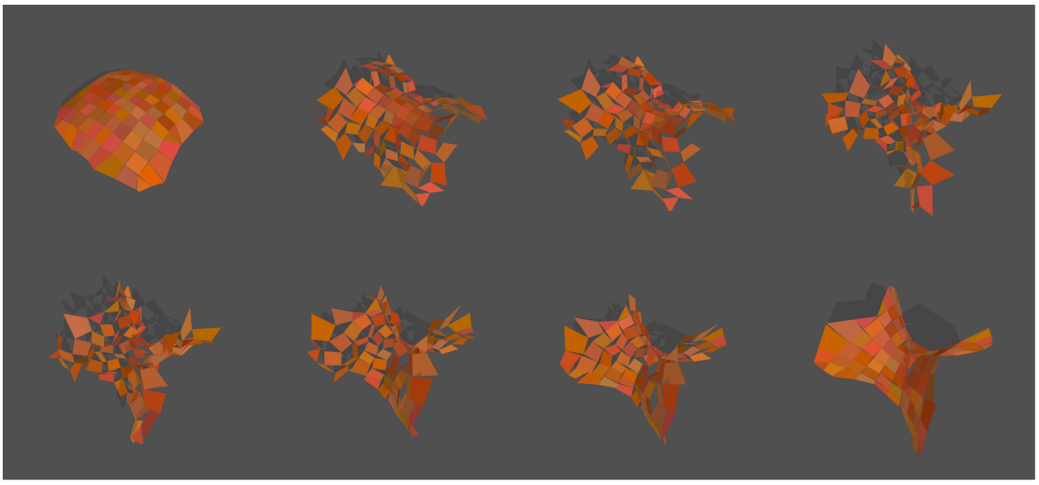

Fig. C.13. The simulated deployment of the sphere cap to saddle produced by PyKirigami. Automatic connection-type detection assigns 135 bottom and 45 top joints based on the saddle target geometry. A very small brick thickness (0.001) renders initial constraint violations negligible.

C.9. Handling more complex deployments

As a challenging test case, we simulate the deployment of a kirigami structure with 100 tiles and 180 joints from a compact spherical-cap shape to a compact saddle shape (Fig. C.13). It is noteworthy that the initial and target configurations exhibit significantly different surface curvatures, making the deployment process challenging. Nevertheless, PyKirigami is capable of producing a real-time deployment that runs interactively at 240 HZ with adaptive stiffness.

Note that the interaction between thickness and connection type introduces a further consideration for 3D kirigami deployment to curved targets. Ideal kirigami tiles are planar polygons without thickness, and in that case the choice of "mountain fold" (bottom connection) or "valley fold" (top connection) does not matter. However, for approximating a curved target using kirigami structures with tile thickness, the connection type plays an important role. Using the automatic connection-type detection (`--auto_detect_connections`) with the saddle as the target geometry, PyKirigami assigns 135 bottom-face joints and 45 top-face joints based on the surface curvature of the target geometry to facilitate the deployment. Also, since we form 3D tile bricks by extruding the planar tile and choosing the original planar tile as the bottom face, it is natural to take the thickness into account and analyze the numerical error when the connection type is 2, i.e., a joint connecting the top faces between two tiles. In this case, the distance between the corresponding bottom vertices is given by

$$\epsilon(h,\theta) = \|v_{i,p} - v_{j,q}\| = 2h\cos(\theta/2),$$

where $v^t_{i,p} = v^t_{j,q}$ are enforced by constraint tuple $(i,p,j,q,2)$ and θ is the dihedral angle of valley fold. This error is already incorporated in the maximal deviation from the target, and one can reduce this part by selecting a small thickness. To handle the deployment of this kirigami structure, we use a very small brick thickness ($h = 0.001$).

Next, we further demonstrate the effectiveness of PyKirigami in handling deployments involving structural twists and orientation changes. Fig. C.14 shows the deployment of a kirigami structure from a rectangular band to a Möbius strip. Specifically, note that the underlying shapes at the two states have different topologies, and another challenging aspect is the change of the outward surface normal direction of the tiles throughout the deployment. As shown in the simulation results, PyKirigami successfully produces a collision-free deployment path to the single-sided target Möbius strip geometry.

Altogether, these examples demonstrate the capability of PyKirigami in achieving desired complex shape-morphing effects.

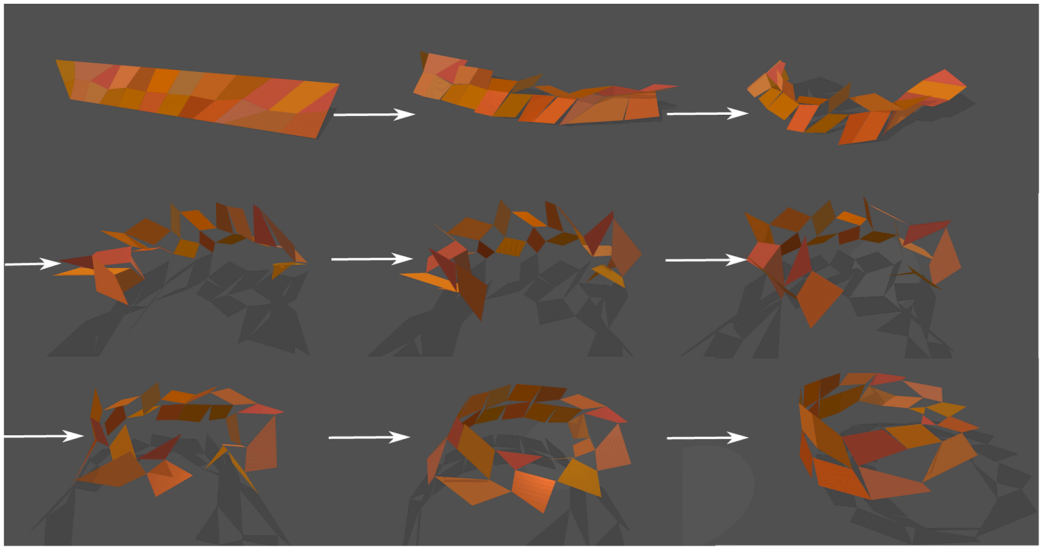

Fig. C.14. 3D deployment sequence of a 24-tile kirigami structure from a rectangular band to a Möbius strip. The 3×3 frame sequence illustrates the full shape-morphing process from the planar rectangular configuration to a topologically non-trivial single-sided Möbius configuration.

Appendix D. Analytic kinematic models for quantitative validation

To validate the numerical accuracy of PyKirigami, we derive the exact kinematic equations for the two benchmark kirigami structures used in main text. Both systems function as single-degree-of-freedom (1-DOF) mechanisms, where the global configuration is uniquely determined by the deployment angle θ.

D.1. Rectangular tessellation

The rectangular tessellation is frequently used as a base case in kirigami design, and its analytical vertex position along with deployment angle θ can be easily derived. As shown in Fig. D.1**a**, the unit cell consisting of four rectangles has the size $L \times W$ where

$$\begin{pmatrix} L \\ W \end{pmatrix} = 2\begin{pmatrix} \sin(\eta) & \cos(\eta) \\ \cos(\eta) & \sin(\eta) \end{pmatrix}\begin{pmatrix} l \\ w \end{pmatrix}, \quad \eta = \theta/2,$$

assuming that the small rectangles have size $l \times w$.

D.2. Square-to-disk kirigami structure

The compact reconfigurable square-to-disk kirigami structure exhibits a more complex, non-linear kinematic path. The radial position R of boundary vertices is coupled to the deployment angle θ (Fig. D.1**b**). The position of all inner vertices $\mathbf{v}_k$ in the assembly can be derived via a matrix

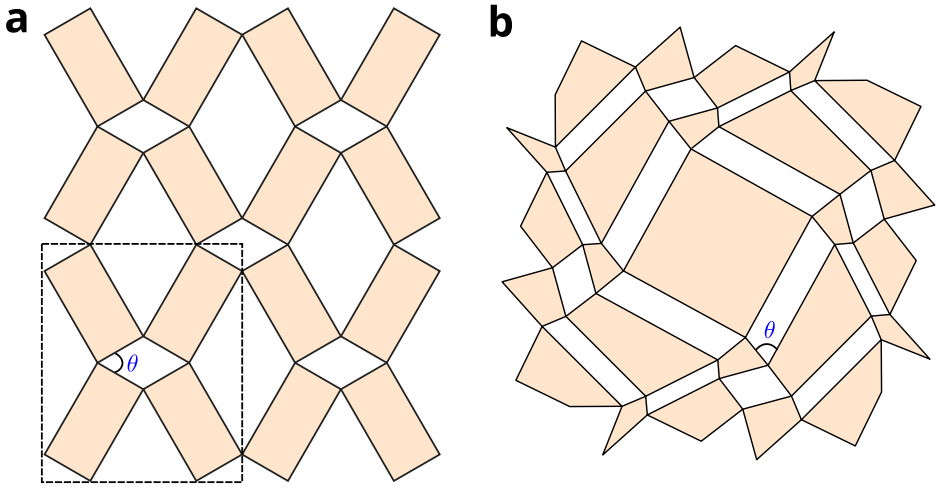

Fig. D.1. The definition of the deployment angle in the benchmark cases. a, The rectangular tessellation kirigami structure. The dashed line shows a unit cell of the tessellation, in which the deployment angle θ can be defined. **b**, The rigidly-deployable square-to-disk kirigami structure from [27], in which the deployment can be characterized by a single deployment angle θ.

multiplication

$$\mathbf{v}_k = MV,$$

where the matrix M is related to the deployment angle and V are the seed vertices. More details can be found in [27].

Appendix E. Software environment and reproducibility

E.1. Software environment

PyKirigami has been developed and tested on the platforms and software versions listed in Table E.1. The experiments to analyze sensitivity and compare PyKirigami and Pymunk are implemented on Macbook Air (M1).

Table E.1
Tested software environment.

Component	Tested version	Notes
Python	3.13	`python -V` to check
NumPy	2.4.1	`conda list numpy`
PyBullet	3.25	`conda list pybullet`
	Ubuntu 24.04 LTS	fully tested
Operating system	macOS 26 (Tahoe)	fully tested
	Windows 11	GUI tested; headless supported

E.2. Reproduction commands for all examples

Table E.2 lists the exact commands used to produce every result in the paper. All required input files are included in the `data/` directory of the repository. The `--headless` flag may be appended to any command to run without a GUI for automated batch execution.

Table E.2
Reproduction commands for all examples. All commands are run from the repository root. Only non-default flags are shown; defaults follow Table B.3. Append `--headless` to any command for batch execution without a GUI.

2D deployments

Square-to-circle — *Fig. 3**a**, C.4*
```
python run_sim.py --model square2circle_w3_h3 --brick_thickness 0.2 --spring_stiffness 300 --ground_plane
--gravity -10 --cm_expansion
```
Square-to-disk — *Fig. 3**b**, 5, C.5*
```
python run_sim.py --model square2disk --brick_thickness 0.2 --spring_stiffness 25 --ground_plane --gravity
-10
```
Stampfli-24 quasicrystal — *Fig. C.8*
```
python run_sim.py --model stampfli24 --brick_thickness 0.2 --ground_plane --gravity -10 --cm_expansion
```
Stampfli-132 quasicrystal — *Fig. C.9*
```
python run_sim.py --model stampfli132 --brick_thickness 0.2 --ground_plane --gravity -10 --cm_expansion
```
Fan model — *Fig. C.10*
```
python run_sim.py --model fan --brick_thickness 0.2 --ground_plane --gravity -10
```
Nonrigid Demo — *Fig. C.12**a***
```
python run_sim.py --model demo_nonrigid --brick_thickness 0.2 --ground_plane --gravity -10
--filter_collision
```

3D deployments

Square-to-spherical-cap — *Fig. 3**c**, C.6*
```
python run_sim.py --model partialSphere
```
Cylinder — *Fig. 3**d**, C.7*
```
python run_sim.py --model cylinder --force_damping 80
```
Cube-to-sphere — *Fig. C.11*
```
python run_sim.py --model cube2sphere_w3_h3
```
Nonrigid Demo — *Fig. C.12**b***
```
python run_sim.py --model demo_nonrigid --brick_thickness 0.2 --ground_plane --gravity -10
```
Sphere-to-saddle — *Fig. C.13*
```
python run_sim.py --model sphere-to-saddle --brick_thickness 0.001 --auto_detect_connections
```
Möbius strip — *Fig. C.14*
```
python run_sim.py --model mobius --brick_thickness 0.005 --spring_stiffness 25 --torque_stiffness 25
```

References

[1] E. Barchiesi, M. Spagnuolo, L. Placidi, Mechanical metamaterials: a state of the art, Math. Mech. Solids 24 (1) (2019) 212–234.
[2] Z. Zhai, L. Wu, H. Jiang, Mechanical metamaterials based on origami and kirigami, Appl. Phys. Rev. 8 (4) (2021) 041319.
[3] X. Li, W. Peng, W. Wu, J. Xiong, Y. Lu, Auxetic mechanical metamaterials: from soft to stiff, Int. J. Extreme Manuf. 5 (4) (2023) 042003.
[4] G.P.T. Choi, Computational design of art-inspired metamaterials, Nat. Comput. Sci. 4 (8) (2024) 549–552.
[5] L. Jin, S. Yang, Engineering kirigami frameworks toward real-world applications, Adv. Mater. 36 (9) (2024) 2308560.
[6] X. Dang, G. Paulino, Kirigami engineering: the interplay between geometry and mechanics, Appl. Mech. Rev. 77 (5) (2025) 050801.
[7] S. Jiang, X. Liu, J. Liu, D. Ye, Y. Duan, K. Li, Z. Yin, Y. Huang, Flexible metamaterial electronics, Adv. Mater. 34 (52) (2022) 2200070.
[8] Z. Song, X. Wang, C. Lv, Y. An, M. Liang, T. Ma, D. He, Y.-J. Zheng, S.-Q. Huang, H. Yu, et al., Kirigami-based stretchable lithium-ion batteries, Sci. Rep. 5 (1) (2015) 10988.
[9] Y. Yang, K. Vella, D.P. Holmes, Grasping with kirigami shells, Sci. Robot. 6 (54) (2021) eabd6426.
[10] J.N. Grima, K.E. Evans, Auxetic behavior from rotating triangles, J. Mater. Sci. 41 (10) (2006) 3193–3196.
[11] J.N. Grima, K.E. Evans, Auxetic behavior from rotating squares, J. Mater. Sci. Lett. 19 (2000) 1563–1565.
[12] D. Attard, J.N. Grima, Auxetic behaviour from rotating rhombi, Phys. Status Solidi B 245 (11) (2008) 2395–2404.
[13] A. Rafsanjani, D. Pasini, Bistable auxetic mechanical metamaterials inspired by ancient geometric motifs, Extreme Mech. Lett. 9 (2016) 291–296.
[14] M. Stavric, A. Wiltsche, Geometrical elaboration of auxetic structures, Nexus Netw. J. 21 (2019) 79–90.
[15] L. Liu, G.P.T. Choi, L. Mahadevan, Wallpaper group kirigami, Proc. R. Soc. A 477 (2252) (2021) 20210161.
[16] L. Liu, G.P.T. Choi, L. Mahadevan, Quasicrystal kirigami, Phys. Rev. Res. 4 (3) (2022) 033114.
[17] P. Shi, Y. Chen, Y. Wei, J. Feng, T. Guo, Y. Tu, P. Sareh, Deformation response of highly stretchable and ductile graphene kirigami under uniaxial and biaxial tension, Phys. Rev. B 108 (13) (2023) 134105.
[18] R. He, Y. Chen, J. Liang, Y. Sun, J. Feng, P. Sareh, Crystallographically programmed kirigami metamaterials, J. Mech. Phys. Solids 193 (2024) 105903.
[19] B.G.-g. Chen, B. Liu, A.A. Evans, J. Paulose, I. Cohen, V. Vitelli, C.D. Santangelo, Topological mechanics of origami and kirigami, Phys. Rev. Lett. 116 (13) (2016) 135501.
[20] G.P.T. Choi, L.H. Dudte, L. Mahadevan, Programming shape using kirigami tessellations, Nat. Mater. 18 (9) (2019) 999–1004.
[21] S. Chen, G.P.T. Choi, L. Mahadevan, Deterministic and stochastic control of kirigami topology, Proc. Natl. Acad. Sci. 117 (9) (2020) 4511–4517.
[22] G.P.T. Choi, L.H. Dudte, L. Mahadevan, Compact reconfigurable kirigami, Phys. Rev. Res. 3 (4) (2021) 043030.
[23] X. Dang, F. Feng, H. Duan, J. Wang, Theorem for the design of deployable kirigami tessellations with different topologies, Phys. Rev. E 104 (5) (2021) 055006.
[24] Y. Zheng, I. Niloy, P. Celli, I. Tobasco, P. Plucinsky, Continuum field theory for the deformations of planar kirigami, Phys. Rev. Lett. 128 (20) (2022) 208003.
[25] Y. Hong, Y. Chi, S. Wu, Y. Li, Y. Zhu, J. Yin, Boundary curvature guided programmable shape-morphing kirigami sheets, Nat. Commun. 13 (1) (2022) 530.
[26] L. Wang, Y. Chang, S. Wu, R.R. Zhao, W. Chen, Physics-aware differentiable design of magnetically actuated kirigami for shape morphing, Nat. Commun. 14 (1) (2023) 8516.
[27] L.H. Dudte, G.P.T. Choi, K.P. Becker, L. Mahadevan, An additive framework for kirigami design, Nat. Comput. Sci. 3 (5) (2023) 443–454.
[28] C. Qiao, S. Chen, Y. Chen, Z. Zhou, W. Jiang, Q. Wang, X. Tian, D. Pasini, Inverse design of kirigami through shape programming of rotating units, Phys. Rev. Lett. 134 (17) (2025) 176103.
[29] D.M. Sussman, Y. Cho, T. Castle, X. Gong, E. Jung, S. Yang, R.D. Kamien, Algorithmic lattice kirigami: a route to pluripotent materials, Proc. Natl. Acad. Sci. 112 (24) (2015) 7449–7453.
[30] M. Konaković, K. Crane, B. Deng, S. Bouaziz, D. Piker, M. Pauly, Beyond developable: computational design and fabrication with auxetic materials, ACM Trans. Graph. 35 (4) (2016) 1–11.
[31] M. Konaković-Luković, J. Panetta, K. Crane, M. Pauly, Rapid deployment of curved surfaces via programmable auxetics, ACM Trans. Graph. 37 (4) (2018) 1–13.
[32] X. Wang, S.D. Guest, R.D. Kamien, Keeping it together: interleaved kirigami extension assembly, Phys. Rev. X 10 (1) (2020) 011013.
[33] T. Chen, J. Panetta, M. Schnaubelt, M. Pauly, Bistable auxetic surface structures, ACM Trans. Graph. 40 (4) (2021) 1–9.
[34] X. Dang, F. Feng, H. Duan, J. Wang, Theorem on the compatibility of spherical kirigami tessellations, Phys. Rev. Lett. 128 (3) (2022) 035501.
[35] Y. Li, Q. Zhang, Y. Hong, J. Yin, 3D transformable modular kirigami based programmable metamaterials, Adv. Funct. Mater. 31 (43) (2021) 2105641.
[36] X. Dang, S. Chen, A.E. Acha, L. Wu, D. Pasini, Shape and topology morphing of closed surfaces integrating origami and kirigami, Sci. Adv. 11 (18) (2025) eads5659.
[37] R. He, Y. Chen, J. Shi, Y. Bai, J. Feng, Programming morphological and mechanical performance of cyclic ori-kirigami via design-feasible parameter space, Thin-Walled Struct. 207 (2025) 112706.
[38] T. Tachi, Simulation of rigid origami, Origami 4 (08) (2009) 175–187.
[39] T. Tachi, Freeform variations of origami, J. Geom. Graph 14 (2) (2010) 203–215.
[40] K. Liu, G.H. Paulino, MERLIN: a MATLAB implementation to capture highly nonlinear behavior of non-rigid origami, in: Proceedings of IASS Annual Symposia, 2016, International Association for Shell and Spatial Structures (IASS), 2016, pp. 1–10.
[41] K. Liu, G.H. Paulino, et al., Highly efficient nonlinear structural analysis of origami assemblages using the MERLIN2 software, Origami 7 (2018) 1167–1182.
[42] A. Ghassaei, E.D. Demaine, N. Gershenfeld, Fast, interactive origami simulation using GPU computation, Origami 7 (2018) 1151–1166.
[43] L. Jin, A.E. Forte, B. Deng, A. Rafsanjani, K. Bertoldi, Kirigami-inspired inflatables with programmable shapes, Adv. Mater. 32 (33) (2020) 2001863.
[44] N. An, A.G. Domel, J. Zhou, A. Rafsanjani, K. Bertoldi, Programmable hierarchical kirigami, Adv. Funct. Mater. 30 (6) (2020) 1906711.
[45] R. Zhu, H. Yasuda, G.L. Huang, J.K. Yang, Kirigami-based elastic metamaterials with anisotropic mass density for subwavelength flexural wave control, Sci. Rep. 8 (1) (2018) 483.
[46] N.A. Alderete, L. Medina, L. Lamberti, C. Sciammarella, H.D. Espinosa, Programmable 3D structures via kirigami engineering and controlled stretching, Extreme Mech. Lett. 43 (2021) 101146.
[47] X. Ying, D. Fernando, M.A. Dias, Inverse design of programmable shape-morphing kirigami structures, Int. J. Mech. Sci. 286 (2025) 109840.
[48] E. Coumans, Y. Bai, PyBullet, a python module for physics simulation for games, robotics and machine learning, 2016, http://pybullet.org.
[49] Origami simulator, 2018, https://origamisimulator.org/.