# Kinegami: Open-source Software for Creating Kinematic Chains from Tubular Origami

*D. A. Feshbach, W.-H. Chen, D. E. Koditschek, C. R. Sung*

**Abstract**: *Arms, legs, and fingers of animals and robots are all examples of "kinematic chains" - mechanisms with sequences of joints connected by effectively rigid links. Lightweight kinematic chains can be manufactured quickly and cheaply by folding tubes. In recent work [Chen et al. 23], we demonstrated that origami patterns for kinematic chains with arbitrary numbers of degrees of freedom can be constructed algorithmically from a minimal kinematic specification (axes that joints rotate about or translate along). The work was founded on a catalog of tubular crease patterns for revolute joints (rotation about an axis), prismatic joints (translation along an axis), and links, which compose to form the specified design. With this paper, we release an open-source python implementation of these patterns and algorithms. Users can specify kinematic chains as a sequence of degrees of freedom or by specific joint locations and orientations. Our software uses this information to construct a single crease pattern for the corresponding chain. The software also includes functions to move or delete joints in an existing chain and regenerate the connecting links, and a visualization tool so users can check that the chain can achieve their desired configurations. This paper provides a detailed guide to the code and its usage, including an explanation of our proposed representation for tubular crease patterns. We include a number of examples to illustrate the software's capabilities and its potential for robot and mechanism design.*

## 1 Introduction

Origami is a promising platform for constructing mechanisms and robots due to its lightweight structure, parameterizable mechanical properties, and cheap materials and fabrication [Rus and Tolley 18]. The low cost also gives it potential to broaden access to robot construction, but the required design expertise – both for mechanisms and origami patterns – can be a barrier to entry. Software tools can help users generate origami designs with desired mechanical behavior, and algorithm-assisted systems designed for exploratory, human-in-the-loop workflows would be particularly helpful for users without the expertise to directly specify exactly what they want. We present such a tool for kinematic chains (sequences of joints connected by rigid links) made of tubular origami, based on the pattern library and joint placement algorithms from [Chen et al. 23].

Previous work in computational origami has developed pattern generation tools for many classes of static structures including branching figures [Lang 96], tessellations [Bateman 02], polyhedral surfaces [Demaine and Tachi 17], and box pleatings [Lang and Tsai 18]. Other software lets users design their own patterns and simulate their kinematics [Tachi 10, Ghassaei et al. 18, Suto et al. 23] and mechanics [Gillman et al. 18, Liu and Paulino 18].

Our previous work [Chen et al. 23] introduced a catalog (Figure 2) of tubular origami patterns for revolute and prismatic joints connected by bending and twisting links. In that paper, we also provided algorithms to construct the path geometry of a whole chain given the kinematic specifications, i.e., the rotational axes for revolute joints and the translational axes for prismatic joints. These algorithms append joints one by one, solving for the new joint's position and orientation such that it has the specified axis of motion and has a feasible link path from the previous joint.

In this paper, we use the results from [Chen et al. 23] to create an open-source tool for algorithm-assisted design exploration of kinematic chains made of tubular origami. Using our system requires minimal technical background, namely, basic familiarity with matrix-vector math and python. The system supports a variety of user workflows to explore designs and configurations, as summarized in Figure 1. In particular, because the sufficient conditions developed in [Chen et al. 23] that guarantee a non-intersecting physical implementation may yield overly conservative (large limbed) designs, we add editing support to rework the automatically generated crease pattern (e.g., use human intuition to develop a more compact but still non-intersecting implementation). Moreover, compared to [Chen et al. 23], we add capacity to visualize link structures (by their bounding cylinders), and plot a variety of configurations of a given chain design. Our code, and module folding videos, are at https://sung.seas.upenn.edu/research/kinegami/.

Additionally, we provide a fabrication-agnostic mathematical representation for tubular origami patterns. The code from [Chen et al. 23] calculates patterns directly in 2D as rectangles with duplication at the sides: it builds physical tubes by rolling the rectangles around and adhering the duplicated sides together. However, tubular origami structures can be manufactured in other ways [Wickeler et al. 23], so it is more general to separate out the pattern representation from the fabrication file generation. Therefore in this paper we represent patterns with a graph respecting the cylindrical wraparound: the duplication involved in 2D-to-3D fabrication is handled in its own method rather than being part of the underlying representation. Vertices are represented using a 2D parameterization of the prism surface where one coordinate wraps modularly about the tube. Edges represent shortest paths on the surface connecting the vertices. There is initially ambiguity about which shortest path an edge should correspond to, but we resolve this by proving (Lemma 1) that a straight crease in a tubular pattern cannot travel more than halfway about the tube, so an edge is naturally associated with a unique globally-shortest path (unless the vertices are radially opposite: in that case there are exactly two globally-shortest paths, whose representations we can distinguish by vertex ordering, and if either is creased then both must be).

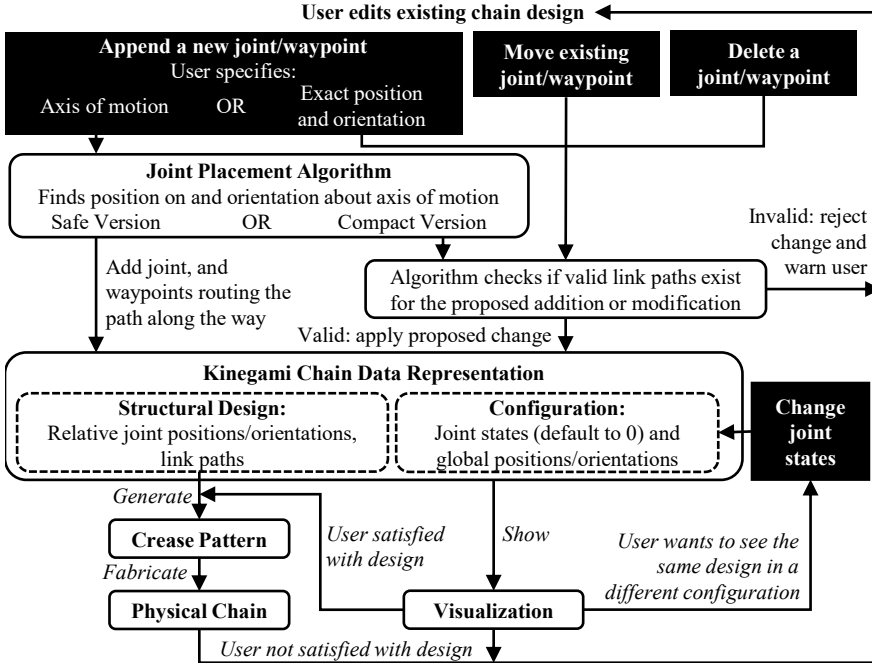


**Figure 1:** *Flow chart depicting system capabilities to illustrate potential user workflows. Boxes with black background are user actions to edit the chain design or configuration, and arrows with italicized labels are user judgements or actions. The crease pattern is based only on the chain's structural design, while the visualization also depends on the current configuration (joint states).*

The remainder of the paper is structured as follows. Section 2 defines the relevant concepts in mechanisms and robotics. Section 3 reviews the tubular origami modules and joint placement algorithms from [Chen et al. 23], gives an overview of our system's capabilities for exploring chain designs and configurations, and shows example chains. Section 4 describes mathematical details of our graph representation for tubular origami, with proof of Lemma 1 and explanation of why it is necessary. Section 5 provides a user guide to our python code. Section 6 concludes with directions for future work.

## 2 Background

### 2.1 3D Rigid Transformations, Reference Frames, and Poses

The *(proper) 3D rigid transformations* describe the ways in which one can move an object in 3D space without deforming or reflecting it. Such transformations map each point in the object to its new location, i.e., they are maps $\mathbb{R}^3 \to \mathbb{R}^3$, but they are commonly encoded by $4 \times 4$ matrices operating on vectors in *homogeneous*

*coordinates* (position vectors with 1 appended). Specifically, a (proper 3D rigid) *transformation matrix* is a $4 \times 4$ matrix of the form

$$T = \begin{pmatrix} R & \mathbf{t} \\ \mathbf{0}^\top & 1 \end{pmatrix}, \quad R = \begin{pmatrix} \hat{\mathbf{x}} & \hat{\mathbf{y}} & \hat{\mathbf{z}} \end{pmatrix} \tag{1}$$

where $R$ is a *rotation matrix* (i.e., $\hat{\mathbf{x}}, \hat{\mathbf{y}}, \hat{\mathbf{z}}$ are an orthonormal basis of $\mathbb{R}^3$ ordered such that $\hat{\mathbf{x}} \times \hat{\mathbf{y}} = \hat{\mathbf{z}}$) and $\mathbf{t}$ is the translation vector. This defines the 3D rigid transformation $\mathbf{v} \mapsto R\mathbf{v} + \mathbf{t}$.

The terms "frame", "pose", and "transformation" are closely related: they are all represented by this class of matrices, but they are used in slightly different contexts. A *pose* is the position and orientation of some object. A (reference) *frame* defines a coordinate system with respect to which other things are encoded: the pose of an object defines a frame, but frames can also be more abstract, such as the *global frame* given by the origin and coordinate axes of the visualization plotting system. A (rigid) *transformation* is a map between frames or poses. Further details can be found in [Waldron and Schmiedeler 16].

### 2.2 Joints and Links

A *kinematic chain* is a sequence of *joints* connected by rigid structures called *links* [Waldron and Schmiedeler 16]. A *joint* is a connection between two rigid structures that constrains their relative motion [Waldron and Schmiedeler 16]. Joints whose motion constraints arise from contact between the body surfaces have six basic types, classified by the types of motion they allow [Waldron and Schmiedeler 16]. Three have a single degree of freedom characterized by a line called the *axis of motion*:

- *Revolute* joints allow rotation about the axis.
- *Prismatic* joints allow translation along the axis.
- *Helical* joints allow screw-like motion: rotation about and translation along the axis, coupled together by a *pitch* ratio into one degree of freedom.

The other three basic joint types have multiple degrees of freedom:

- *Cylindrical* joints allow both rotation about and translation along an axis of motion (not coupled together, so they are two separate degrees of freedom).
- *Spherical* joints allow rotation about a point (three degrees of freedom).
- *Planar* joints allow translation and rotation within a plane (three degrees of freedom).

Each of the multiple-degree-of-freedom joints can be instantiated by composing revolute and prismatic joints. [Chen et al. 23] includes examples of each of these compound joints, and we also generate such examples with our code (Figure 3).

### 2.3 Dubins Paths

Paths with a minimum turning radius, called *Dubins paths* after their introduction in 2D by [Dubins 57], are well-studied in the motion planning literature, for example

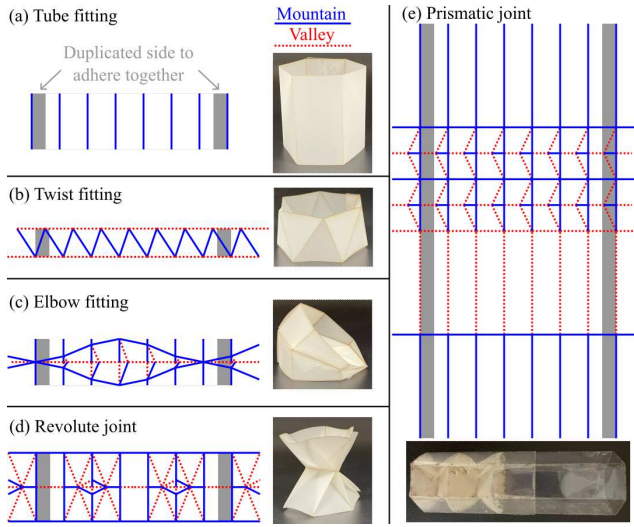


**Figure 2:** *The tubular origami pattern catalog from [Chen et al. 23], with $n = 6$. The fabricated examples are folded from Polyethylene terephthalate (PET) plastic with creases etched by laser-cut dots. The tubular wraparound is adhered with tape. The prismatic joint example is folded from clear PET with the inner REBO structure painted white for visibility.*

for turning-constrained vehicles [Boissonnat et al. 94, Lugo-Cárdenas et al. 14, Cai et al. 17, Karapetyan et al. 18]. In 3D, the shortest Dubins path from a start point and direction to a goal point and direction is either a helicoidal arc or of the form CSC or CCC (or a degenerate case thereof), where S is a straight line segment and C is a circular arc of the minimum turning radius [Sussmann 95]. [Hota and Ghose 10] provides an optimization-based approach to computing CSC Dubins paths in 3D.

## 3 System Overview and Definitions

Our system enables users to create kinematic chains made of tubular origami. The chains (section 3.1) have revolute and prismatic joints connected by links following CSC Dubins paths. To iteratively construct a chain, users append each joint either by specifying its exact pose or by specifying only its axis of motion and letting an algorithm (section 3.2) place the joint along the axis. Users can input poses and axes either in global coordinates or relative to the previous joint's pose. Additionally, users can modify existing chains by moving or deleting joints. Joint movements can be given as translations along the axis of motion, rotations about the axis of motion, or arbitrary rigid transformations. Joint movements can be set either to propagate to all subsequent joints or to apply only to the given joint.

If a proposed modification cannot generate feasible link paths, the system will reject the change and warn the user. When a user has a candidate design, they can specify the state of each joint to visualize different configurations of the chain, letting them check whether it can do what they want and modify the design accordingly. Figure 1 summarizes available workflows. Figure 3 shows a variety of example chains visualized in our system. The generating code is in the `examples` folder of our repository.

### 3.1 Kinegami Chains and Configurations

Figure 2 shows the catalog of tubular origami modules defined by [Chen et al. 23], instantiated on the hexagonal prism ($n = 6$). We illustrate and fabricate the patterns as a flat sheet with the first side duplicated at the end to adhere the tubular wraparound, but the patterns could also be folded directly from a cylinder.

A link in a kinegami chain is a pair of elbow fitting modules joined by a tube, with a twist if necessary for alignment. Since elbows instantiate rotation equivalent to an arc with the same radius as the tube, a link instantiates a CSC Dubins path (degenerate cases can occur in which one or more of the path sections is empty and thus the corresponding origami modules are omitted). To avoid self-intersection within a link, we do not use C components with turning angle $> \pi$.

The revolute joint is a pair of triangular-prism-like polyhedra joined along the axis of rotation. The prismatic joint is a REBO pattern [Chen et al. 20] with a surrounding cylinder to prevent off-axis bending. Both joint types are compliant (see [Chen et al. 23] for energy analysis). The *state* of a joint is a real number indicating its current displacement (rotation in radians for revolute joints, translation for prismatic joints) from the minimum-energy state (state 0). The *joint pose* of a joint in a chain is located at the center of the joint's physical structure (at state 0), and its $\hat{\mathbf{z}}$ direction is the axis of motion. Note that our prismatic joints connect to links along their axis of motion, while our revolute joints connect to links orthogonally to the axis of motion. Therefore, we define a joint's *path direction* $\hat{\mathbf{a}}$ as the end tangent of the incoming link path: for prismatic joints this is $\hat{\mathbf{z}}$, and for revolute joints this is $\hat{\mathbf{x}}$.

For each joint we also define *proximal* and *distal* frames where they connect to the incoming and outgoing links respectively. The proximal frame has orientation matching that of the joint pose, and it is fixed relative to the joint pose. In contrast, the distal frame's position (and for revolute joints, orientation) relative to the joint pose depends on the joint's current state.

A *configuration* of a $k$-joint chain is a vector in $\mathbb{R}^k$ storing the state of each joint. It is important to distinguish the chain's configuration from its structural design: different configurations do not vary the link shapes, and therefore cannot alter the relative pose of each joint in its predecessor's distal frame.

### 3.2 Joint Placement Algorithm Overview

Since the kinematic behavior of revolute or prismatic joints is given by their axes of motion, we say revolute or prismatic joints of the same type with the same axis of motion are *kinematically equivalent*, and the kinematics of a chain is specified only by its sequence of axes of motion. [Chen et al. 23] provides two algorithms to construct tubular chains given a sequence of axes: a *safe* algorithm and a relatively *compact* algorithm. The core idea is to convert the chain design problem into a path planning problem under the observation that each module has a centerline path and therefore can be considered as the instantiation of a rigid motion resulting in the appropriate transformation of the local frame of the chain. Since the tubular radius constrains the centerline path curvature, this is a Dubins planning problem.

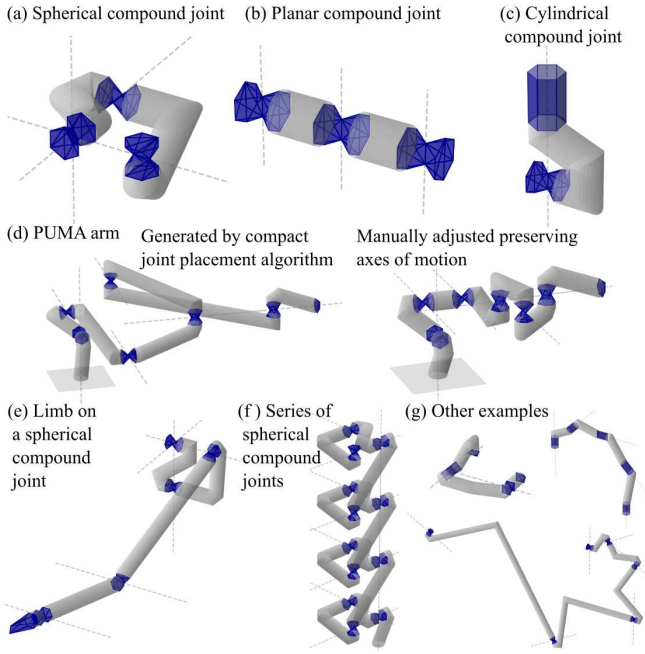


**Figure 3:** *Example chains generated with our system. (a-c) Basic multi-degree-of-freedom joints constructed as compound structures of revolute and prismatic joints. (d) PUMA arm generated using the compact joint placement algorithm (based on Denavit-Hartenberg parameters from [Lee and Ziegler 84]) then modified by manually translating joints along their axes of motion. (e-g) Other examples.*

Links are constructed to instantiate CSC paths (section 3.1), so the joint placement algorithms solve for poses (along the given axes of motion) far enough apart that CSC paths exist linking the joints. The *compact* joint placement algorithm places a new joint at least $4r$ from the current chain's bounding sphere, and attaches it with a CSC path found by an optimization from [Hota and Ghose 10]. However, such CSC paths may have turn angles $> \pi$ which would cause collisions in our chain structures, so the alternate *safe* joint placement algorithm routes the chain through intermediate *waypoint* poses connected by CSC paths with turn angles $\leq \pi$. These waypoints often route the links along unnecessarily convoluted paths, which is why the first algorithm is called (relatively) "compact". However, even the compact algorithm often generates chains which can be made much shorter with human intuition moving joints along their axes (as in Figure 3(d)), motivating the chain editing tools we introduce in this paper.

## 4 Representing Tubular Origami Crease Patterns

This section discusses formally how we represent tubular origami patterns as graphs, and specifically which path along the surface corresponds to a given edge. We encode vertices (points on the pattern connected by creases) in a manner similar to the angle-height parameterization of a cylinder surface, but applied to the prism tube. Specifically, the tubular patterns from [Chen et al. 23] are based on right prisms whose bases are regular $n$-gons of circumradius $r$ (with $n \geq 4$ even). One base is considered the *starting base* and one side of it the *starting side*. We parameterize

vertices as $(x,y)$ where $y$ is the height along the tube and $x$ is the distance about the prism modulo $nb$, where $b = 2r\cos\frac{\pi(n-2)}{2n}$ is the side length of the polygon base. These are measured relative to the point $(0,0)$ bisecting the starting side.

We can define a *tubular origami surface* as a polyhedral surface *isometric* (deformable without stretching or tearing) to a cylinder. A crease is an edge of this polyhedral surface, so it is a line segment in $\mathbb{R}^3$ along the folded surface. Since a line segment is a *geodesic* (a locally-shortest path) and isometries preserve geodesics [Do Carmo 16], it corresponds to a geodesic on the cylinder. In a flat origami pattern there is a unique geodesic (a line segment) connecting a given pair of vertices, so creases can easily be represented by edges (vertex pairs) and therefore patterns can be represented by graphs. In a tubular pattern, however, there is ambiguity to be resolved regarding exactly which geodesic a vertex pair should map to. The geodesics on the cylinder are the helices [Do Carmo 16] and there are infinitely many helices on a cylinder connecting a given pair of points (a helix can proceed either clockwise or counterclockwise about the cylinder, and can make arbitrarily many full turns around it between the two points).

We resolve this ambiguity by defining the path corresponding to a pair of vertices as the globally-shortest path along the surface connecting the vertices. We say a path is *creased* if it corresponds to a line segment joining adjacent facets in the folded polyhedron. An edge in our graph represents the path for that vertex pair being creased. If the vertices are radially opposite (i.e., their $x$ coordinates differ by exactly $nb/2$) then there are exactly two globally-shortest paths connecting them, which we distinguish via vertex order: if one is creased then the other must be as well, but they may have different mountain-valley labels. This is justified by the following observation that specifies the natural helix an edge should correspond to.

**Lemma 1.** A crease in a tubular origami pattern cannot travel more than halfway about the tube, i.e., the corresponding path in the cylinder winds at most $\pi$ about its central axis. If it travels exactly halfway around, then the symmetrical path in the other direction about the tube is also creased.

*Proof.* Let $\mathbf{F} \subset \mathbb{R}^3$ be a tubular origami surface and $f : [0,nb) \times [0,h] \to \mathbf{F}$ be its parameterization map as described above. Let $\mathbf{C} \subset \mathbb{R}^3$ be the isometrically corresponding cylinder surface and $c : [0,nb) \times [0,h] \to \mathbf{C}$ be its parameterization.

Let $v_1 = (x_1,y_1)$ and $v_2 = (x_2,y_2)$ be vertices where $(v_1,v_2)$ is an edge. Using an edge to encode a crease means that in the folded structure, the line segment connecting these vertices is entirely on the folded structure, i.e., $\overline{f(v_1)f(v_2)} \subset \mathbf{F}$. Since a line segment on a surface is a geodesic, it corresponds to a helix $h$ on $\mathbf{C}$. Since isometries preserves length [Do Carmo 16], we have $||h|| = ||f(v_1) - f(v_2)||$.

Let $h'$ be any helix connecting $c(v_1)$ and $c(v_2)$. Since $h'$ is also a geodesic and isometries preserve length, it corresponds to a geodesic on $\mathbf{F}$ of length $||h'||$ connecting $f(v_1)$ and $f(v_2)$. By the triangle inequality, $||h'|| \geq ||f(v_1) - f(v_2)|| = ||h||$. Therefore $h$ is a *globally* shortest helix connecting $c(v_1)$ and $c(v_2)$.

Since $h$ is globally shortest, it cannot have complete wraparound, i.e., its angular travel is $\leq 2\pi$. This leaves two candidates connecting $c(v_1)$ and $c(v_2)$, a

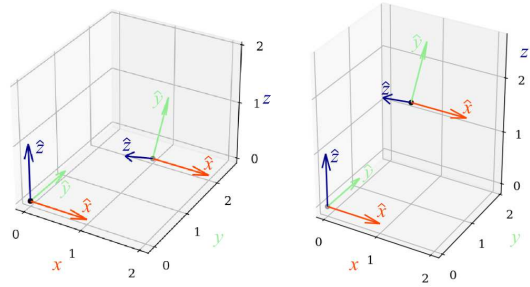

**Figure 4:** *Poses obtained by composing a rotation and a transformation in either order, relative to the global frame* `SE3()` *in the bottom left of each plot.*
*Left:* `SE3.Trans(1,2,0) @ SE3.Rx(np.pi/3)` *translates by* $(1,2,0)$ *in the global frame and then rotates about* $\hat{x}$ *by* $\pi/3$.
*Right:* `SE3.Rx(np.pi/3) @ SE3.Trans(1,2,0)` *first rotates by about* $\hat{x}$ *by* $\pi/3$ *and then translates by* $(1,2,0)$ *relative to the already-rotated frame axes.*

clockwise helix $h_{cw}$ and a counterclockwise helix $h_{ccw}$, whose angular travels sum to $2\pi$. If their angular travels are not equal (i.e., if $|x_1 - x_2| \neq nb/2$), the one with angular travel $< \pi$ is shorter, so it is $h$. Otherwise (i.e., if $|x_1 - x_2| = nb/2$), $c(v_1)$ and $c(v_2)$ are radially opposite on $\mathbf{C}$, so $h_{cw}$ and $h_{ccw}$ each have angular travel $\pi$ and $||h_{cw}|| = ||h_{ccw}|| = ||f(v_1) - f(v_2)||$. Then by the triangle inequality, each of them must correspond to $\overline{f(v_1)f(v_2)}$, i.e., they are both creased in $\mathbf{F}$. □

## 5 User Guide to Kinegami Code

In this section, we provide an overview of how the above concepts translate into our code implementation, and how a user may interact with the code to create their own kinematic chains.

We represent chains with a `KinematicChain` class. A chain stores a list of `Joint` objects (discussed in section 5.2), which each store their pose (section 5.1). `KinematicChain` also computes and stores links as a list of `LinkCSC` objects (section 5.3), and has a `show` method for visualization.

To make a chain, initialize it with a starting joint and then append further joints (section 5.4) using the `append` method on `KinematicChain`. This method has options to place the new joint at its stored pose or to use a joint placement algorithm to find a pose on its axis of motion. Users can also edit an existing chain by moving or deleting joints (section 5.5), and can change joint states to visualize different configurations of the same chain (section 5.6). Once they are satisfied with a chain design, they can export its crease pattern (section 5.7).

### 5.1 3D Rigid Transformations using [Corke and Haviland 21]

We represent 3D rigid transformations (and thus poses and reference frames) using the `SE3` class in the `spatialmath` package from [Corke and Haviland 21]. It

includes a variety of construction options such as `SE3()` for the identity transformation, `SE3.Trans(x,y,z)` for a translation, `SE3.Rx(angleInRadians)` for rotation about $\hat{\mathbf{x}}$ (and similarly `SE3.Ry` and `SE3.Rz`), and `SE3(T)` where `T` is an array encoding a transformation matrix. Transformations are composed with the operator `@` (see Figure 4) and applied to vectors (represented by numpy arrays) with the operator `*`.

### 5.2 Joints

We represent joints with an abstract base class `Joint` and subclasses for each specific joint type. `Joint` stores the joint's current `state` (a float, initialized to 0), its `Pose`, the tubular radius `r`, and its length `neutralLength` (the distance in state 0 between proximal and distal frames). It has an abstract method (a method implemented separately for each subclass) `pathIndex` specifying the path direction by returning an axis index (0 for $\hat{\mathbf{x}}$ or 2 for $\hat{\mathbf{z}}$). There are four subclasses:

- A `RevoluteJoint` (Figure 2(d)) is constructed from parameters `numSides`, tube radius `r`, angle `totalBendingAngle` (defining the range of states to be $\pm$`totalBendingAngle`$/2$), and `Pose`. It also has an optional parameter `numSinkLayers` defaulting to 1, which can be increased to insert "recursive sink gadget" layers to its origami pattern that increase its stiffness (see [Chen et al. 23]).
- A `PrismaticJoint` (Figure 2(e)) is constructed from parameters `numSides`, `r`, `neutralLength`, `numLayers`, `coneAngle`, and `Pose`. The parameters `numLayers` and `coneAngle` (an angle in $(\frac{\pi}{6}, \frac{\pi}{2})$) affect the joint's extension range and stiffness by controlling the number and internal angle of REBO pattern layers [Chen et al. 20] inside the joint.
- A `Waypoint` is a pose that the path routes through. It has no physical structure or changeable state (i.e., its state range is $\{0\}$). The constructor parameters are `numSides`, `r`, `Pose`, and optionally `pathIndex` (defaults to 2 for $\hat{\mathbf{z}}$).
- `StartTip` and `EndTip` are static structures closing the beginning and end of a tube respectively. The constructor parameters are `numSides`, `r`, `Pose`, and `length`. Its path direction is $\hat{\mathbf{z}}$. It does not have changeable state, i.e., its state range is $\{0\}$. The pattern consists of half of the revolute joint pattern.

### 5.3 Links

The `LinkCSC` class stores the radius, the start and end poses, and a CSC Dubins path connecting them. The S components are implemented as tube and twist fittings, while the C arcs are implemented as elbow fittings. Since elbow fittings protrude increasingly outwards as the turn angle approaches $\pi$, we avoid excessively long elbows by replacing them with a concatenation of several shorter elbows. The `LinkCSC` constructor has an optional parameter `maxAnglePerElbow`, defaulting to $\pi/2$, to do this: when implementing a C component with turn angle $\theta >$ `maxAnglePerElbow`, it will concatenate $k = \lceil \theta/$`maxAnglePerElbow`$\rceil$ elbow patterns each with turn angle $\theta/k$.

### 5.4 Chain Generation

Our `KinematicChain` class stores on a list `Joints` of `Joint` objects and a list `Links` of `LinkCSC` objects. Specifically, `Links[i]` stores the link from the distal frame of joint `i-1` to the proximal frame of joint `i` (the first link `Links[0]` is an empty path at the proximal frame of joint 0). `KinematicChain` also stores and maintains a `boundingBall` encompassing the whole structure. Chains are initialized with a given `startJoint`. The constructor also has optional parameter `maxAnglePerElbow`, defaulting to `np.pi/2`, which it uses when constructing links as described in section 5.3.

Chains are constructed by repeatedly adding joints to the end of the chain with the `append` method (details below), for example:

**Example 1A.**

```
from KinematicChain import *
r = 1
numSides = 4
chain = KinematicChain(StartTip(numSides, r, Pose=SE3.Trans(3,3,0),
                                 length=1.5))
prismatic = PrismaticJoint(numSides, r, neutralLength=3,
            numLayers=3, coneAngle=np.pi/4, Pose=SE3.Trans([5,5,0]))
prismaticIndex = chain.append(prismatic)
revolute = RevoluteJoint(numSides, r, np.pi, SE3.Ry(np.pi/4))
revoluteIndex = chain.append(revolute)
end = EndTip(numSides, r, Pose=SE3.Ry(np.pi/2), length=1.5)
endIndex = chain.append(end)
chain.show(showGlobalFrame=True) #Figure 5(a)
```

Figure 5(a) depicts the result of the call to `KinematicChain`'s `show` method. This method has a variety of optional parameters controlling which features are displayed. Color and opacity default values are defined in the `style.py` file.

The `append` method takes parameter `newJoint` (a `Joint` object) and adds it to the end of the chain. It returns the index of the new joint. Depending on the following optional parameters, it will either use a joint placement algorithm (section 3.2) to find the joint pose on its axis of motion ($\hat{\mathbf{z}}$), or use `newJoint`'s pose exactly as given.

- `relative`, defaulting to `True`, indicates whether `newJoint.Pose` should be interpreted as relative to the previous joint's frame (`True`) or as already in the global frame (`False`). `KinematicChain` stores joints in global coordinates, so if `relative=True` it converts the input from local to global coordinates.
- `safe`, defaulting to `True`, indicates whether it should use the safe version of the joint placement algorithm (see section 3.2). If `safe` is `False`, it will use the compact joint placement algorithm instead, unless `fixedPosition` is `True` in which case it does not algorithmically find joint placement at all.
- `fixedPosition`, defaulting to `False`, indicates whether the joint should be placed exactly at the position in its given pose (`True`) or should use the joint placement algorithms to choose somewhere kinematically equivalent (i.e., along its $\hat{\mathbf{z}}$ axis). This and `safe` cannot both be `True`.

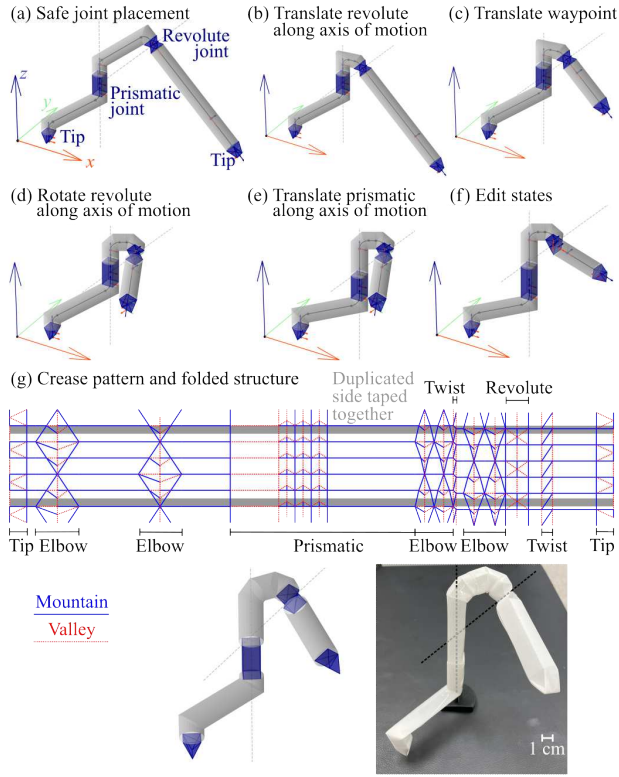


**Figure 5:** *Results from the running example. (a) Example 1A generates a chain with the safe joint placement algorithm. (b-e) Example 1B adjusts this chain by transforming joints and waypoints in ways which preserve their axes of motion and therefore preserve the chain's overall kinematics. (f) Example 1C shows the same chain design in a different configuration, one with each of the joints in nonzero states. (g) The origami pattern generated for this design, and a physical construction of this pattern (alongside the program visualization for comparison).*

- `fixedOrientation`, defaulting to `False`, indicates whether the joint should be placed with exactly its given orientation (`True`) or with a kinematically equivalent orientation (i.e., rotated about its $\hat{\mathbf{z}}$ axis) such that its $\hat{\mathbf{x}}$ points along the common normal from the previous joint's $\hat{\mathbf{z}}$ to the new joint's $\hat{\mathbf{z}}$.[1] This and `safe` cannot both be `True`.

The safe algorithm is guaranteed to find a valid path to the new joint, but if `safe` is False it may fail to find a path. In this case it will print a warning and return `None`, leaving the chain unchanged.

### 5.5 Adjusting an Existing Chain

`KinematicChain` has several methods to edit existing joints:

- `translateJointAlongAxisOfMotion` takes parameters `jointIndex` and `distance`.
- `rotateJointAboutAxisOfMotion` takes parameters `jointIndex` and `angle`.

[1] This choice is to match how frames are specified by Denavit-Hartenberg parameters [Denavit and Hartenberg 21, Waldron and Schmiedeler 16].

- `transformJoint` takes parameters `jointIndex` and `Transformation`.
- `delete` removes joint `jointIndex` and recomputes links accordingly.

Each method returns whether it succeeded in finding new paths for the links: if it succeeds it edits the chain accordingly, and if it fails it leaves the chain unchanged and prints a warning. Each method has optional parameter `propogate`, defaulting to `True`, indicating whether to apply the same transformation to the rest of the chain (`True` case) or only to the given joint (`False` case).

Kinematic-preserving methods `translateJointAlongAxisOfMotion` and `rotateJointAboutAxisOfMotion` each have an additional optional parameter `applyToPreviousWaypoint` defaulting to `False`. If set to `True`, this will check if joint `jointIndex-1` is a waypoint: if so, it will apply the same transformation matrix it applied to `jointIndex`. Continuing from Example 1A, with the results of `chain.show()` in Figure 5(b)–(d):

**Example 1B.**

```
chain.translateJointAlongAxisOfMotion(revoluteIndex, -7)
chain.show(showGlobalFrame=True) #Figure 5(b)
chain.translateJointAlongAxisOfMotion(endIndex, -10,
                                      applyToPreviousWaypoint=True)
chain.show(showGlobalFrame=True) #Figure 5(c)
chain.rotateJointAboutAxisOfMotion(revoluteIndex, -np.pi/3)
chain.show(showGlobalFrame=True)
chain.translateJointAlongAxisOfMotion(prismaticIndex, -2,
                                            propogate=False)
chain.show(showGlobalFrame=True) #Figure 5(d)
pattern = chain.creasePattern()
pattern.save(dxfName="examplePatterns/example1.dxf")
```

### 5.6 Changing Joint States

The state of a joint in a `KinematicChain` can be adjusted by the chain method `setJointState`, which takes parameters `jointIndex` and `newState`. It will return whether this succeeded, i.e., whether `newState` is in the joint's valid state range - and if it fails it will also print a warning. `Joint` has a `stateRange` method to help with this. Continuing from Example 1B, with the `chain.show` result in Figure 5(f):

**Example 1C.**

```
minPrismaticState, maxPrismaticState = chain.Joints[prismaticIndex].
                                      stateRange()
chain.setJointState(prismaticIndex, maxPrismaticState)
chain.setJointState(revoluteIndex, np.pi/2)
chain.show(showGlobalFrame=True) # Figure 5(f)
```

### 5.7 Tubular Origami Pattern Generation

The method `creasePattern` on `KinematicChain` outputs the tubular origami crease pattern implementing the chain as a `TubularPattern` object. This class

has a `show` method to plot and a `save` method to create a DXF file. To display the wraparound, it duplicates the starting panel (and all folds crossing it) along each end. This also creates area along which to adhere a flat sheet into a tube. Figure 5(g) shows the crease pattern for Example 1B and the resulting folded structure.

## 6 Conclusion

We present a python library for design exploration and crease pattern generation of tubular kinematic chains, accessible to users with only basic familiarity with matrix-vector math and python. The library has been demonstrated on chains with a wide variety of morphologies and degrees of freedom. In the future, to further facilitate intuitive design of origami robots and mechanisms, we plan to add an interactive graphical user interface and features for post-generation optimization to reduce overall length or volume or to guarantee a desired workspace. Finally, full robot or animal bodies often include branching structures such as hands and multiple limbs, so we plan to expand the system to support kinematic trees by providing new joint placement algorithms and fabrication techniques.

## 7 Acknowledgements

Support for this project has been provided in part by NSF Grant No. 2322898 and in part by the Army Research Office under the SLICE Multidisciplinary University Research Initiatives Program award under Grant W911NF1810327. The authors also thank Dongsheng Chen, Shivangi Misra, Jessica Weakly, and Gabriel Unger for helpful feedback on the writing.

## References

[Bateman 02] Alex Bateman. "Computer Tools and Algorithms for Origami Tessellation Design." In *Origami 3: Third International Meeting of Origami Science, Mathematics, and Education*, 2002.

[Boissonnat et al. 94] Jean-Daniel Boissonnat, André Cérézo, and Juliette Leblond. "Shortest Paths of Bounded Curvature in the Plane." *Journal of Intelligent and Robotic Systems* 11 (1994), 5–20. doi:10.1007/BF01258291.

[Cai et al. 17] Wenyu Cai, Meiyan Zhang, and Yahong Rosa Zheng. "Task Assignment and Path Planning for Multiple Autonomous Underwater Vehicles Using 3D Dubins Curves." *Sensors* 17:7. doi:10.3390/s17071607.

[Chen et al. 20] Wei-Hsi Chen, Shivangi Misra, Yuchong Gao, Young-Joo Lee, Daniel E. Koditschek, Shu Yang, and Cynthia R. Sung. "A Programmably Compliant Origami Mechanism for Dynamically Dexterous Robots." *IEEE Robotics and Automation Letters* 5:2 (2020), 2131–2137. doi:10.1109/LRA.2020.2970637.

[Chen et al. 23] Wei-Hsi Chen, Woohyeok Yang, Lucien Peach, Daniel E. Koditschek, and Cynthia R. Sung. "Kinegami: Algorithmic Design of Compliant Kinematic Chains From Tubular Origami." *IEEE Transactions on Robotics* 39:2 (2023), 1260–1280. doi:10.1109/TRO.2022.3206711.

[Corke and Haviland 21] Peter Corke and Jesse Haviland. "Not Your Grandmother's Toolbox–the Robotics Toolbox Reinvented for Python." In *IEEE International Conference on Robotics and Automation*. IEEE, 2021. doi:10.1109/ICRA48506.2021.9561366.

[Demaine and Tachi 17] Erik D. Demaine and Tomohiro Tachi. "Origamizer: A Practical Algorithm for Folding Any Polyhedron." In *33rd International Symposium on Computational Geometry*, 2017. doi:10.4230/LIPIcs.SoCG.2017.34.

[Denavit and Hartenberg 21] J. Denavit and R. S. Hartenberg. "A Kinematic Notation for Lower-Pair Mechanisms Based on Matrices." *Journal of Applied Mechanics* 22:2 (2021), 215–221. doi:10.1115/1.4011045.

[Do Carmo 16] Manfredo P Do Carmo. *Differential Geometry of Curves and Surfaces: Revised and Updated Second Edition*. Courier Dover Publications, 2016.

[Dubins 57] Lester E Dubins. "On Curves of Minimal Cength With a Constraint On Average Curvature, and With Prescribed Initial and Terminal Positions and Tangents." *American Journal of Mathematics* 79:3 (1957), 497–516.

[Ghassaei et al. 18] Amanda Ghassaei, Erik D. Demaine, and Neil Gershenfeld. "Fast, Interactive Origami Simulation using GPU Computation." In *Origami 7: Seventh International Meeting of Origami Science, Mathematics, and Education*, 2018.

[Gillman et al. 18] A. Gillman, K. Fuchi, and P.R. Buskohl. "Truss-based Nonlinear Mechanical Analysis for Origami Structures Exhibiting Bifurcation and Limit Point Instabilities." *International Journal of Solids and Structures* 147 (2018), 80–93. doi:10.1016/j.ijsolstr.2018.05.011.

[Hota and Ghose 10] Sikha Hota and Debasish Ghose. "Optimal Geometrical Path In 3D With Curvature Constraint." In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2010. doi:10.1109/IROS.2010.5653663.

[Karapetyan et al. 18] Nare Karapetyan, Jason Moulton, Jeremy S. Lewis, Alberto Quattrini Li, Jason M. O'Kane, and Ioannis Rekleitis. "Multi-robot Dubins Coverage with Autonomous Surface Vehicles." In *IEEE International Conference on Robotics and Automation*, 2018. doi:10.1109/ICRA.2018.8460661.

[Lang and Tsai 18] Robert J. Lang and Mu-Tsun Tsai. "Generalized Offset Pythagorean Stretches in Box-Pleated Uniaxial Bases." In *Origami 7: Seventh International Meeting of Origami Science, Mathematics, and Education*, 2018.

[Lang 96] Robert J. Lang. "A Computational Algorithm for Origami Design." In *Twelfth Annual Symposium on Computational Geometry*, 1996. doi:10.1145/237218.237249.

[Lee and Ziegler 84] C.S.G. Lee and M. Ziegler. "Geometric Approach in Solving Inverse Kinematics of PUMA Robots." *IEEE Transactions on Aerospace and Electronic Systems* AES-20:6 (1984), 695–706. doi:10.1109/TAES.1984.310452.

[Liu and Paulino 18] Ke Liu and Glaucio Paulino. "Highly Efficient Nonlinear Structural Analysis of Origami Assemblages Using the MERLIN2 Software." In *Origami 7: Seventh International Meeting of Origami Science, Mathematics, and Education*, 2018.

[Lugo-Cárdenas et al. 14] Israel Lugo-Cárdenas, Gerardo Flores, Sergio Salazar, and Rogelio Lozano. "Dubins Path Generation for a Fixed Wing UAV." In *International Conference on Unmanned Aircraft Systems*, pp. 339–346, 2014. doi:10.1109/ICUAS.2014.6842272.

[Rus and Tolley 18] Daniela Rus and Michael T Tolley. "Design, Fabrication and Control of Origami Robots." *Nature Reviews Materials* 3:6 (2018), 101–112. doi:10.1038/s41578-018-0009-8.

[Sussmann 95] H.J. Sussmann. "Shortest 3-Dimensional Paths With a Prescribed Curvature Bound." In *IEEE Conference on Decision and Control*, 1995. doi:10.1109/CDC.1995.478997.

[Suto et al. 23] Kai Suto, Yuta Noma, Kotaro Tanimichi, Koya Narumi, and Tomohiro Tachi. "Crane: An Integrated Computational Design Platform for Functional, Foldable, and Fabricable Origami Products." *ACM Transactions on Computer-Human Interaction* 30:4. doi:10.1145/3576856.

[Tachi 10] Tomohiro Tachi. "Freeform Variations of Origami." *Journal for Geometry and Graphics* 14:2 (2010), 203–215.

[Waldron and Schmiedeler 16] Kenneth J. Waldron and James Schmiedeler. "Kinematics." In *Springer Handbook of Robotics*, edited by Bruno Siciliano and Oussama Khatib, pp. 11–36. Springer, 2016.

[Wickeler et al. 23] Anastasia L Wickeler, Kyra McLellan, Yu-Chen Sun, and Hani E Naguib. "4D Printed Origami-Inspired Accordion, Kresling and Yoshimura Tubes." *Journal of Intelligent Material Systems and Structures* 34:20 (2023), 2379–2392. doi:10.1177/1045389X231181940.

---

Daniel A. Feshbach
General Robotics, Automation, Sensing & Perception (GRASP) Lab,
University of Pennsylvania, 3101 Walnut St, Philadelphia, PA 19104, USA
e-mail: feshbach@seas.upenn.edu

Wei-Hsi Chen
General Robotics, Automation, Sensing & Perception (GRASP) Lab,
University of Pennsylvania, 3101 Walnut St, Philadelphia, PA 19104, USA
e-mail: weicc@seas.upenn.edu

Daniel E. Koditschek
General Robotics, Automation, Sensing & Perception (GRASP) Lab,
University of Pennsylvania, 3101 Walnut St, Philadelphia, PA 19104, USA
e-mail: kod@seas.upenn.edu

Cynthia R. Sung
General Robotics, Automation, Sensing & Perception (GRASP) Lab,
University of Pennsylvania, 3101 Walnut St, Philadelphia, PA 19104, USA
e-mail: crsung@seas.upenn.edu