CEE 345 Final Report

Team Assembly

George Kopf (ECE): Electrical control and packaging
Ella Simmons (MAE): Sliding mechanism design & manufacturing, Kresling manufacturing
Wilson Moyer (MAE): Theoretical analysis, main body design & manufacturing
Remy Garcia-Kakebeen (ORFE): Origami theory & Kresling manufacturing

Overview

Bio-inspired robotics have widespread applications, and in the world of aquatic maneuvering, extant aquatic propulsion structures can have a number of advantages over traditional propeller systems. Navigating tight or complex terrain such as coral reefs, traversing environments such as dense seaweed forests that could tangle traditional propellers, generating thrust without creating unusual noise that stands out and affects local biota, as well as interacting with extant wildlife without modifying their behaviors are all prescribed advantages of using systems that simulate the swimming gait of natural organisms. These are swimming styles that have evolved over billions of years to work effectively in their native niches. Copying those core design principles into our robotic systems is an excellent way to smoothly insert our devices into said niches while maintaining simplicity.

The goal of our project is to effectively create an origami-inspired propulsion system for an aquatic robot that is based on the swimming method of a salp. The idea is to create a swimmer that is less invasive than traditional propeller systems and while being simpler and more efficient than something like a fish-inspired design. The integration of origami design should allow for a lightweight, durable, affordable, and simple-to-construct robot. The main application is intended to be extended, non-invasive data collection of aquatic life in natural ecosystems, although there certainly could be security benefits in underwater surveillance for oil rigs and the Coast Guard.

Introduction to Salps

The salp (*salpidae*) is a gelatinous marine invertebrate with a jellyfish-like, semitransparent body. There are 42 different species under two subfamilies: cyclosalpa and helicosalpa.[1] They are planktonic little creatures that inch across the ocean in a similar fashion to caterpillars on land.

They move through the water using a jet propulsion system naturally built into them, where they contract and relax muscle bands that ring around their bodies.[2] Inspired by these creatures, our project aims to replicate this kind of movement by using an electromagnet, valves, and a tessellation (currently planning to use a water bomb tessellation), as explained further below.

Here is an example that we are referencing for our project.

[1] Anushka Chatterjee, “Salp - Anatomy, Habitat, Diet, Life Cycle, and Pictures,” AnimalFact.com, September 25, 2024, https://animalfact.com/salp.
[2] Ibid.

Kresling Theory and Design

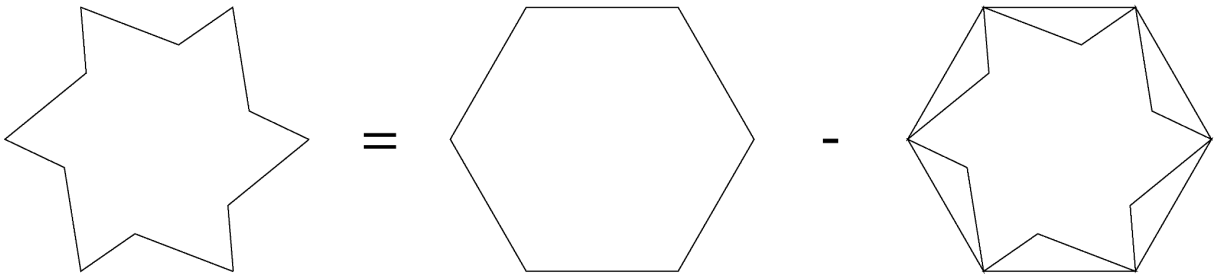
We utilized the lab lecture slides on the Kresling to create a program in Python that calculates the parameters a, α, c, and initial and final twisting angles ψ_1 and ψ_2 based on an edge length b, a number of sides n ($n = 6$ in the hexagon case), and extended and compressed heights H_0 and H. In addition, using a paper written by Liu et al. in 2024, we can calculate the height H_a of the Kresling for any twisting angle using

$$l^2 = H_a^2 - 2r^2 cos(\phi) + 2r^2 \quad (1)$$

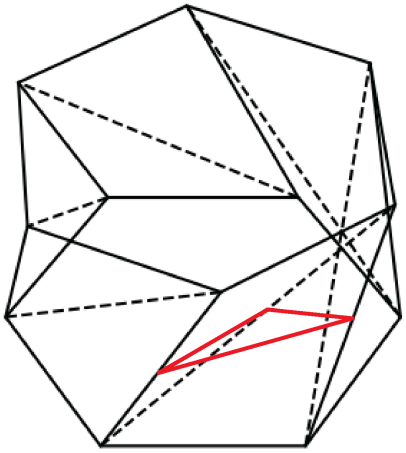
Where r is the radius of the outer circle that defines the hexagon (which is equal to b in this case) and l is a parameter that can be calculated by setting $H_a = H$ and $\phi = \psi_1$.

We would like to know how the Kresling's volume changes over time. This is useful for deriving the thrust of the salp based on conservation of momentum. In addition, while calculating volume we can extract other useful quantities, such as the minimum effective internal radius, which is necessary for properly designing the Kresling to fit a shaft through it.

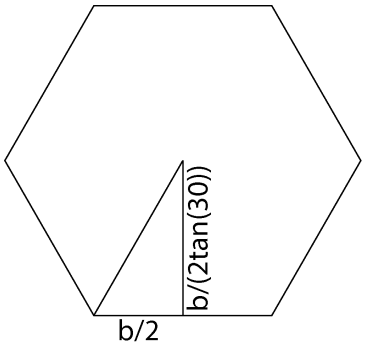
We can approximate the volume of the salp by slicing the Kresling into cross-sections and integrating these cross-sections throughout the height H_a of the Kresling. A critical approximation that we make is that the Kresling is approximately rigid (straight edges and panels) during folding. The Kresling is a non-rigid origami pattern with panel bending and non-straight edges during folding, so this method does not arrive at an exact volume. Nonetheless, this approximation will help us observe general patterns in the Kresling volume. Based on our approximation, the cross section at any height can be expressed as a regular n-gon ($n = 6$ for our hexagonal case) with triangles cut out of it due to the valley folds which approach the center of the Kresling as it compresses:



An example triangle cutout is shown below:

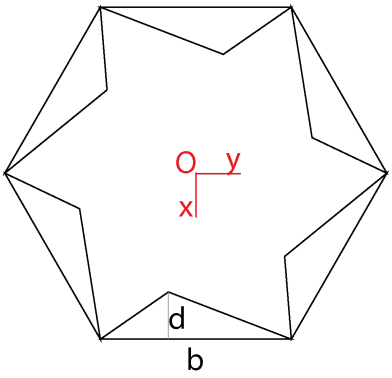


To calculate the total area of each cross-section, we can calculate the hexagon and triangle areas separately. The hexagon consists of 12 right triangles with side lengths $b/2$ and heights $\frac{b}{2\tan(30)}$. Therefore, we calculate the total area of the hexagon with the area of a triangle:



$$A_{Hex} = 12 * \frac{1}{2} * \frac{b}{2} * \frac{b}{2\tan(30)} = \frac{3b^2}{2\tan(30)}$$

To calculate the triangle area, we create a variable d for the height of the triangle:

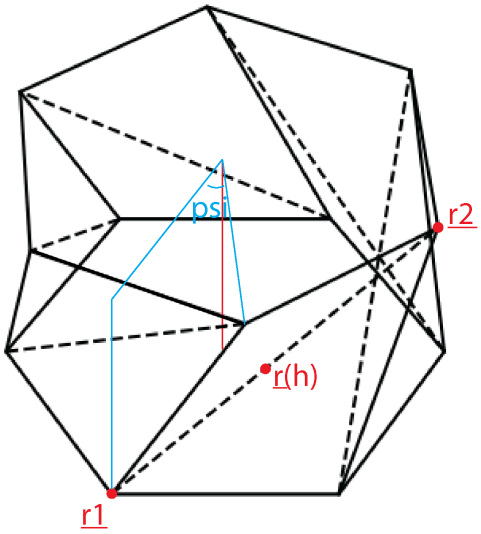


$$A_{Tri} = \frac{1}{2}bd$$

Giving us the total area of the cross-section:

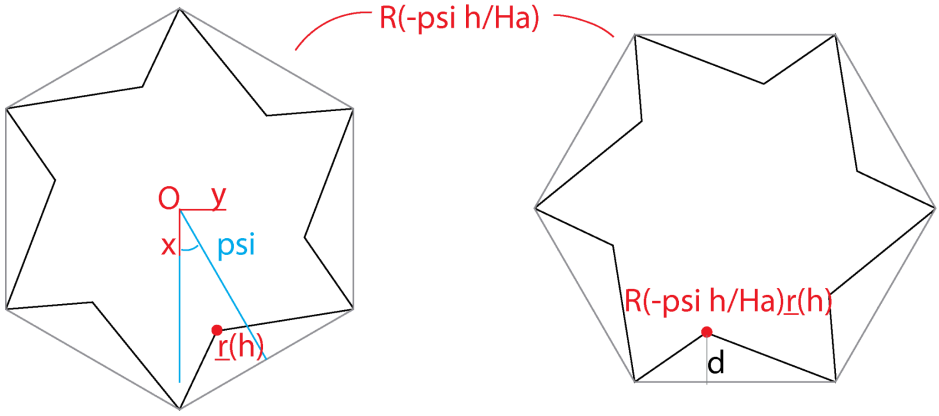
$$A = \frac{3b^2}{2\tan(30)} - 3bd \qquad (2)$$

To calculate the value of d, we can use vectors and linear algebra. We can define 3 vector positions:



With the middle r being a function of h, the height that it is between $r1$ and $r2$. We will set the total height to be H_a, which can be calculated from the twisting angle psi shown above with *(1)*. The vector position of $r1$ is $\overline{r_1} = < \frac{b}{2\tan(30)}, \frac{-b}{2}, 0 >$, and the vector position of $r2$ is equal to a rotation of $r1$ about the red axis shown above by an angle of $\theta = 60° + \phi$, where ϕ is the twisting angle calculated above. We define a rotation matrix $R(\theta)$ about the red axis with entries $R(\theta) = [\cos(\theta), -\sin(\theta), 0; \sin(\theta), \cos(\theta), 0; 0, 0, 1]$. Therefore, we can define $\overline{r_2} = R(\theta)\,\overline{r_1} +< 0, 0, H_a >$. Since $r(h)$ lies between $r1$ and $r2$, we can parameterize it with h and calculate it with $\overline{r}(h) = \overline{r_1} + \frac{h}{H_a}(\overline{r_2} - \overline{r_1})$. We can relate $r(h)$ to d by projecting $r(h)$ onto a line radial to the center of the hexagon and perpendicular to a hexagon edge. The easiest way to do this is to rotate the space $r(h)$ so that the hexagon edge is parallel to the y-axis and then project $r(h)$ onto the x-axis. The rotation can be calculated as a negative rotation of $\frac{\psi h}{H_a}$, and is shown below in calculation and illustration:

$$d(h) = [1\ 0\ 0]R(-\tfrac{h}{H_a}\phi)\,\overline{r}(h)$$



Putting this all together, we get:

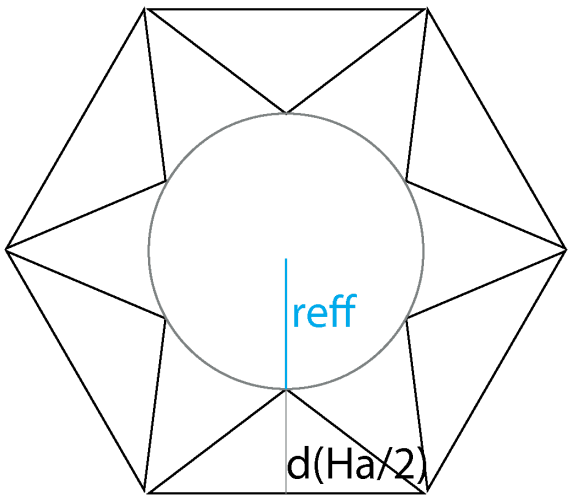
$$d(h) = [1\ 0\ 0]R(-\frac{h}{H_a}\phi)(\overline{r_1} + \frac{h}{H_a}(R(\theta)\ \overline{r_1} + \ < 0,\ 0,\ H_a > \ - \overline{r_1}))$$

The z-axis section $< 0,\ 0,\ H_a >$ falls out, and we can simplify to our final form:

$$d(h) = [1\ 0\ 0]R(-\frac{h}{H_a}\phi)(\overline{r_1}(1 - \frac{h}{H_a}) + \frac{h}{H_a}R(\theta)\ \overline{r_1}) \quad (3)$$

This value has the useful property of also being able to tell us the internal radius of the vertices of the Kresling. We can set $h = \frac{H_a}{2}$ to calculate $d(\frac{H_a}{2})$ at the center. At this point, the triangle is symmetric, so we can calculate the effective internal radius of the Kresling, with the following equation:

$$r_{eff} = \frac{b}{2\tan(30)} - d(\frac{H_a}{2})$$



This result is very useful in determining the optimal parameters for our Kresling. We determined that we would like a minimum height H_0 to be 0 when in the fully compact stable state, and the edge length b to be 1.85 in (4.70 cm), which is the minimum size that still fits the servo motor inside. Because the code parameterizes the entire Kresling using H_0, b, and H, we can tune the final parameter H in order to have an optimal effective internal diameter. Our goal is to minimize the size of the robot to make it compact and easy to manufacture. However, we need to fit a shaft of radius 1.4 cm through the center of the Kresling in order to properly actuate the Kresling. To achieve a minimum size while having the shaft fit correctly, we can calculate the minimum effective radius of the Kresling, which is the Kresling's radius when $H_a = H_0$. Below is a graph that demonstrates the relationship between H and minimum effective radius.

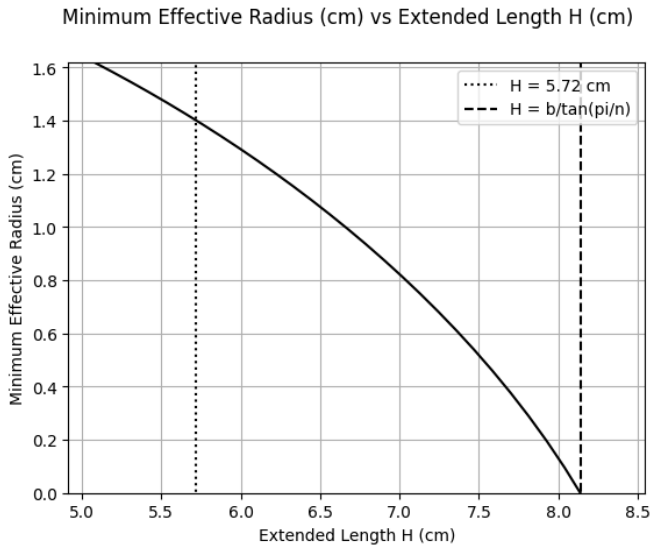


Figure 1: Graph of Kresling's Extended Length vs Kresling's Minimum Effective Radius in cm.

Based on the graph and related calculations, we conclude that $H = 5.72$ cm (2.25 in) matches our design parameters. The above plot also validates our calculations, which we can demonstrate with the following equation from the lab slides:

$$\left|\left(\frac{H}{b}\right)^2 + \left(\frac{H_0}{b}\right)^2\right| \leq \cot^2\left(\frac{\pi}{n}\right)$$

When $H_0 = 0$, we know that the minimum effective radius will be zero at $H = \frac{b}{\tan(\pi/n)}$, which we can see does in fact occur in the above plot. Experimentally, we validated the effectiveness of our code by designing a Kresling with the selected parameters. The designed Kresling fits around the shaft very well, as shown below.

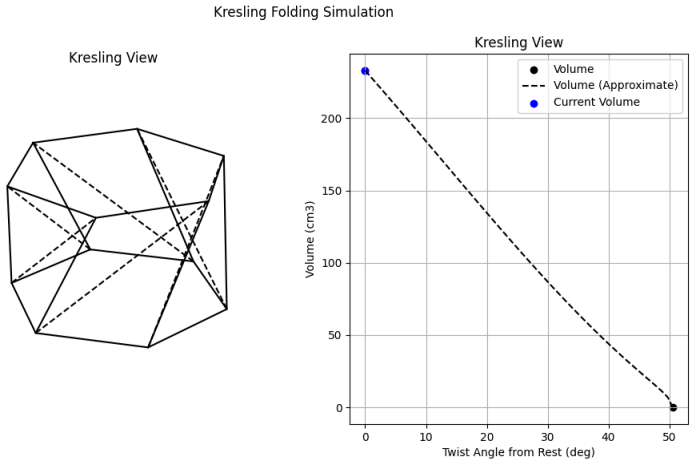


Figure 2: Our folded Kresling (made of PET) around the 3D printed shaft.

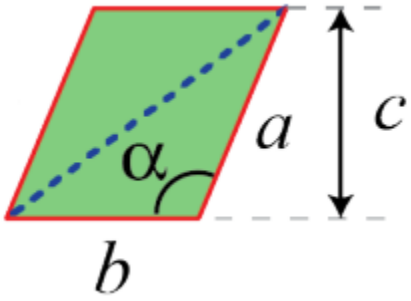
In terms of calculating volume, we can plug *(3)* into *(2)* and integrate from $h = 0$ to $h = H_a$ to get the total volume with the following form:

$$V = \int_0^{H_a} \frac{3b^2}{2\tan(30)} - 3bd(h)dh$$

Using this result alongside equation *(1)* from Liu et al, we can derive the volume of the Kresling based on twist angle. The result for the Kresling with our parameters is shown below:



Since we have a method of obtaining the optimal parameters for the Kresling, we can use these parameters to physically manufacture the Kresling itself. This can be achieved by calculating the parameters a and α, as shown from the Kresling lab slides below:



As determined in the Kresling lab slides, we can calculate a and α from the parameters H_0, H, and b. Using these results, we can use code to calculate the positions of all vertices in the 2D Kresling cutout. The code defines a row of 7 vertices v_i, which are collinear, horizontal, and spaced by a distance b, forming 6 edges that define the base of the Kresling. The code calculates each vertex u_i on the next level by adding a base vertex v_i to a translation defined by a:

$$\overline{u_i} = \overline{v_i} + \ < a\cos(\pi - \alpha),\ a\sin(\pi - \alpha) >$$

The code then uses the ezdxf Python library to convert these vertices and edges into a .dxf file, which can then be cut. The code also adds tabs for ease of assembly. The resulting .dxf file is shown below.

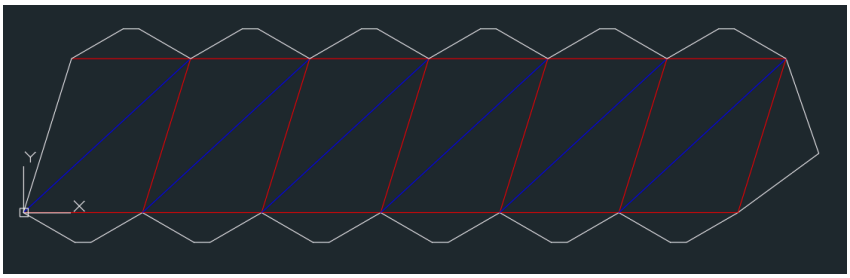


Figure 3: The .dxf file showing our Kresling's crease pattern, with mountain folds denoted with red lines and valley folds denoted with blue lines.

Bibliography:

Liu, Ruiwei, et al. "A Kresling Origami Metamaterial with Reprogrammable Shock Stiffness." *Theoretical and Applied Mechanics Letters*, vol. 14, no. 4, July 2024, p. 100546. *DOI.org (Crossref)*, https://doi.org/10.1016/j.taml.2024.100546.

Moitzi, Manfred. *Ezdxf*. 2025, https://ezdxf.readthedocs.io/en/stable/.

Paulino, Glaucio. "Lab6: Kresling Units." 2025.

Salp Design

The salp design underwent several iterations to best utilize origami in the attempt to make a non-invasive swimming robot. The first design involved using an electromagnet stationed between two magnets at either end of an origami-inspired sac. As current flowed through the electromagnet one way, the magnets would be repelled and move away from the center, elongating the salp shape. Likewise, as the current changed direction, the magnets would be attracted and move closer to the center. This change in direction was intended to replicate the motion of a salp.

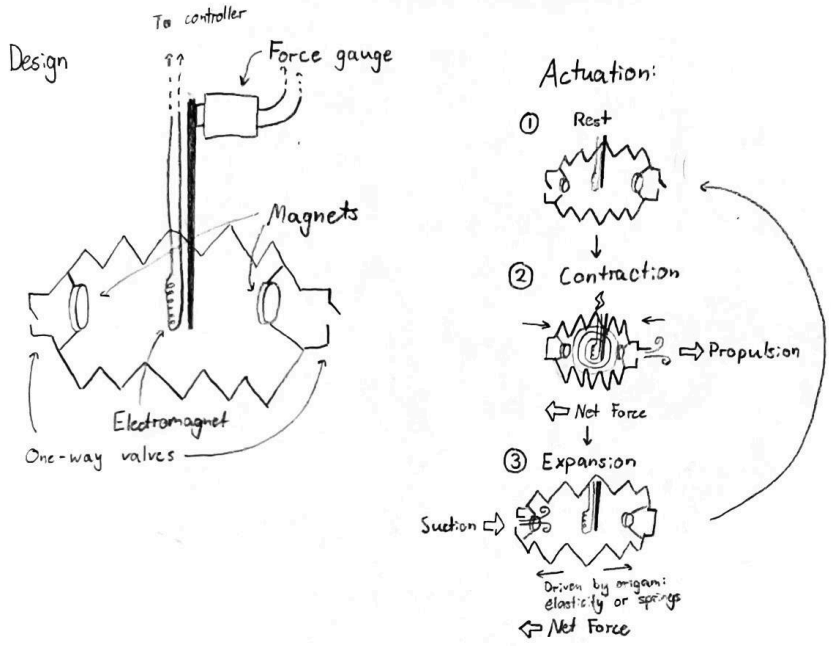


Figure 4: Our initial plan for our salp-inspired aquatic swimmer. The plan involved using an electromagnet that pushed and pulled magnets on the ends of the salp depending on which direction the current would flow through it.

However, there were some issues with the design: the electromagnet would only provide limited motion, as the force required to move the magnets would require them to be very close to the electromagnet at all times. Thus, we decided to pivot away from an electromagnetic actuation method.

The next three designs in Figure 1 below revolved around motor actuation, using either a traditional DC motor or a motor-controlled linear actuator. Ideas 1 and 2 required linear actuators to pull the sac closer and push it apart to create the thrust. Comparatively, idea 3 uses the geometry and stability points of a Kresling to translate limited rotational movement to linear motion. Thus, idea 3 was chosen to move forward with as it better utilizes origami principles to create novel designs and leverages the stability points of the Kresling.

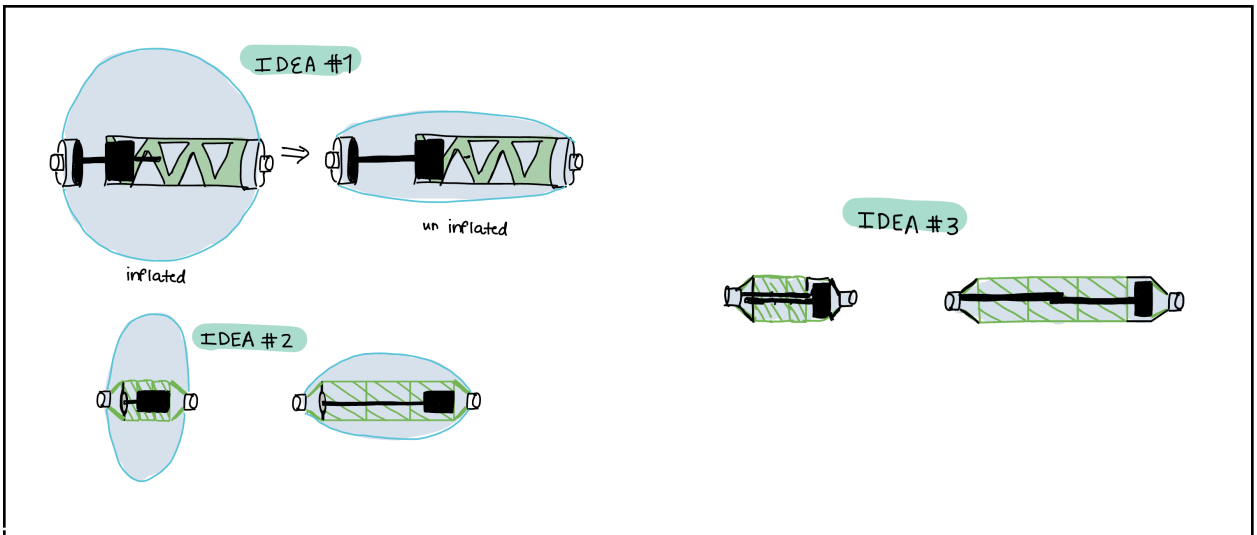


Figure 5: Our newer designs, Ideas 1 and 2 on the left, with our chosen path, Idea 3, on the right.

Idea 3 is shown in the figure below in greater detail:

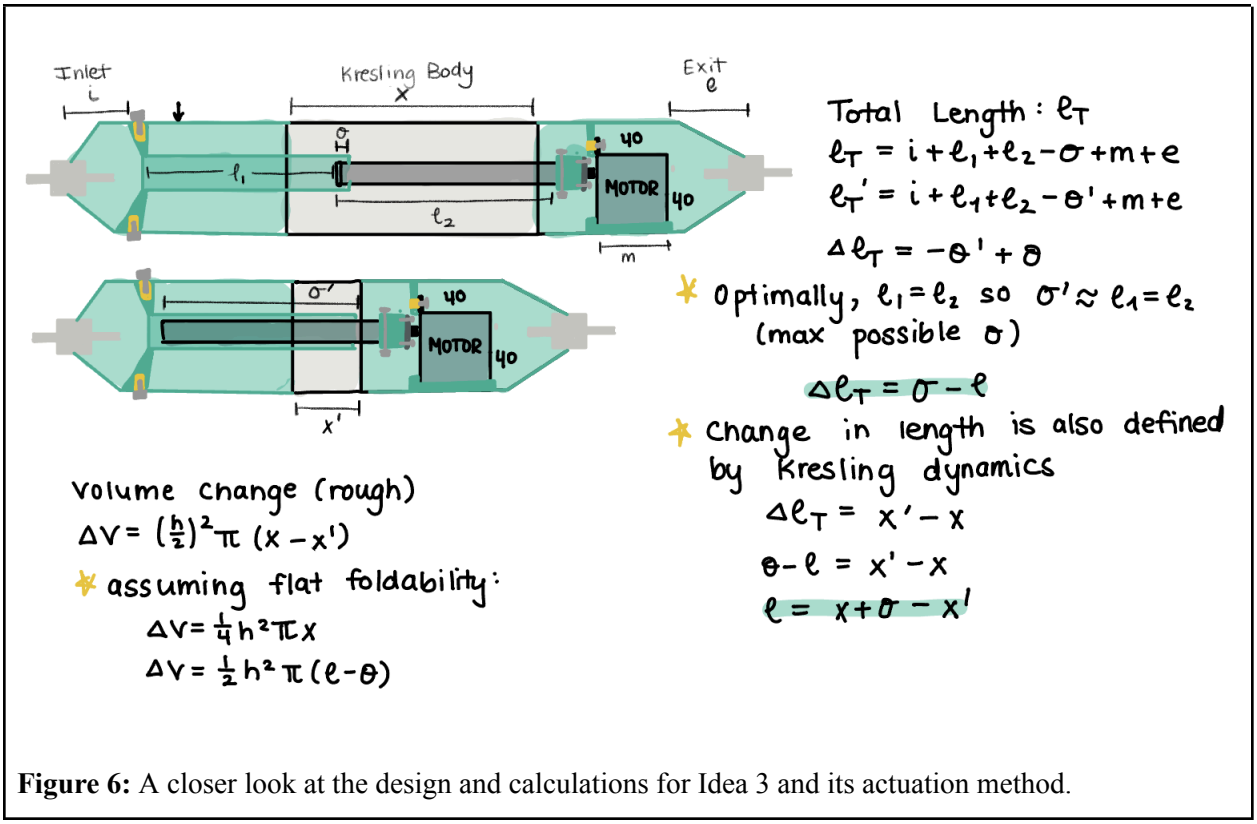


Figure 6: A closer look at the design and calculations for Idea 3 and its actuation method.

The green pieces of the figure are 3D printed parts. These 3D printed caps at the beginning and end serve to hold all the pieces together: the end caps hold one-way valves at either end that limit the flow of water, allowing thrust to be produced. The green and grey shafts slide together as the Kresling expands and contracts. Both shafts are D-shafts, with a flat surface interrupting the circular shape. This means that as

the motor turns the grey D-shaft, it forces the blue D-shaft to turn. Since this forces the end cap on the left to turn, which is rigidly connected to the Kresling body, one end of the Kresling body is forced to rotate in relation to the other, creating linear motion. The motor is mounted to the walls of the right end cap, and the grey shaft is rigidly connected to the motor output. Calculations for the linear motion and size constraints of the Kresling are shown in the figure above. Once this design was finalized, a CAD model was created, see below:

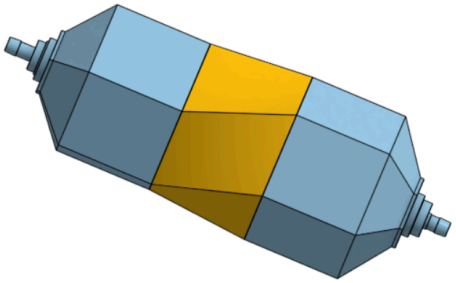


Figure 7: The CAD model visualization of our salp-inspired design.

Since the design has some mechanical nuance, the electronics were made as simply as possible. The motor selected was a 25 kg waterproof servo. This allowed the motor to exist within the system without needing a special waterproof case. The servo's power and signal cable are routed out of the end cap through watertight tubing to an Arduino and battery, which exist above the surface of the water.

Manufacturing and Assembly

Kresling Body

The initial plans involved making the Kresling Body out of silicone provided by the origami lab. To make a silicone model, a mold was created into which the silicone could be poured. The mold followed the cut pattern proposed in the Kresling theory section and included ridges where folds were supposed to be in order to create weaker sections in the Kresling body. This mold was 3D printed for ease of manufacturing.



Figure 8:
From left to right: Ella with our 3D printed Kresling “crease pattern” mold; George pouring the epoxy mix into the mold.

However, when the silicone was cast, there were several problems: firstly, the silicone created weird bubbled sections, which meant that the layers were uneven. Secondly, a 2 mm thickness was poured, and the silicone at that thickness was far too flexible. Both of these problems meant that the silicone would not work as a Kresling. Although the silicone could be re-poured thicker to try and mitigate these problems, a thicker silicone would cause issues when attempting to flat-fold the Kresling during the robot’s range of motion.

Thus, we pivoted to PET plastic, a very thin plastic that is stiffer than paper but can still warp as needed for the Kresling design. To cut the Kresling, a laser cutter was used, as PET does not release toxic fumes. The laser cutter was set to engrave the fold lines for ease of assembly. The PET performed far better than the silicone and was chosen as the final material.

We wanted to test out the different thicknesses of the PET.
Although many thicknesses of PET were cut for various Kreslings, we used the 0.0075” thick (transparent matte color), which we found was stiff enough to hold shape and flexible enough to fold as needed.

Assembly
The assembly process involved many components. Most of the components were 3D printed for the ease of manufacturing and prototyping capabilities. The end caps, sliding D shafts, and interfaces between the end caps and Kresling were all 3D printed. We replaced the 3D printed inner shaft with one made of metal, since the D shafts kept catching on one another, likely due to the tiny ridges of the 3D printed material. The one-way valves and motor were bought from Amazon. All of the components were assembled according to the CAD model created. Where possible, screws were used to affix pieces, as the goal was to make it easily prototypable. For the Kresling and the interface between the Kresling and the end caps, a 2-part epoxy was used to ensure a stiff connection. Additionally, to waterproof the assembly, a layer of silicone was applied after the epoxy dried on all connections.

Further Work

There are many ideas we have for further work. The ideas fall under three main topics: manufacturing, Kresling, and electronics.

There are a couple of key improvements that could be made to the manufacturing processes used to make the Salp. Firstly, several parts were made from 3D prints due to time constraints, and should be made from other materials. This will prevent binding between the two D shafts as they slide, as it will reduce the friction. Additionally, this will involve creating a new connector from the inner D shaft to the motor which can also be made from metal. Along these lines, the arm of the servo (coupler from the output shaft of the motor to turning application), came with the servo and was made of a cheap, thin, and delicate plastic. In future iterations of the project, a different arm would be used. Additionally, as part of the manufacturing process, a better waterproofing method should be explored. The silicone worked, but between the epoxy and silicone, the process of adhering and waterproofing was time intensive and the final product still had leaks. Different waterproofing processes shall be explored in future work.

There is future work that could be conducted involving the Kresling. Although the theory pointed to a crease pattern and PET plastic worked very well in terms of material, there are many thicknesses of PET and none of our theory points to a specific thickness, as it all assumes the Kresling is made out of infinitesimally thin panels. While we were able to test out a couple PET thicknesses, further work will be conducted on a wider range to better assess how it impacts the salps movement.

Finally, the electronics are currently floating at the top of the water, which could affect the flow of the water around the salp, and thus the motion of the salp robot. Thus, in future iterations, the electronics should be stored within the salp. This will require careful packaging to ensure it is both as small as possible and totally waterproof.

After all of these improvements have been made, we hope to progress the development and scope of the project by including the ability for the robot to change its direction, allowing a more complete bio-inspired swimmer. This might include, but is not limited to, autonomous navigation with vision tracking, navigation of currents, and multiple robots being able to connect and form a bigger swimming robot, as salps do in the wild.

Appendix

Below is the code that we developed to create the dxf files of the kresling cutting pattern and plot volume vs twisting angle.

```
# Code for simulating kresling parameters
# And producing DXF files of the kresling pattern

import numpy as np
import matplotlib.pyplot as plt
import ezdxf
```

```
from ezdxf import units
import os
import cv2
from scipy.integrate import quad


def generateNGon(n, b, in3D = False):
    """
    Generates the vertices of a regular ngon

    :param n: Number of edges
    :param b: Edge length
    :param in3D: Whether to return 3d vectors or 2d
    """
    internal_angle = np.pi-(n-2)*np.pi/n
    current_angle = 0
    points = np.zeros((n, 2))
    for i in range(1, n):
        points[i] = points[i-1]+np.array([np.cos(current_angle), np.sin(current_angle)])*b
        current_angle += internal_angle

    center = np.average(points, axis = 0)
    points -= center
    if in3D:
        points3D = np.zeros((n, 3))
        points3D[:, :2] = points
        points = points3D
    return points

def planar_rotation(vector, theta):
    """
    Rotates the vector by amount theta

    :param vector: Vector to rotate
    :param theta: Rotation amount (radians)
    """
    rotation_matrix = np.array([[np.cos(theta), -1*np.sin(theta), 0], [np.sin(theta),
np.cos(theta), 0], [0, 0, 1]])
    return rotation_matrix @ vector

class Kresling_Volume_All:
    """
    Class for calculating the volume of an array of kresling configurations
    """
    def __init__(self, Kresling_Tube, twisting_angles):
        """
        Initialization

        :param Kresling_Tube: A Kresling object to calculate volume for
        :param twisting_angles: Array of twisting angles (degrees) for calculation
        """
        self.twisting_angles = twisting_angles
        self.all_volumes = np.zeros(len(self.twisting_angles))
        for i in range(len(self.all_volumes)):
            Kresling_Volume_Obj = Kresling_Volume_Single(Kresling_Tube, twisting_angles[i])
            self.all_volumes[i] = Kresling_Volume_Obj.calculate_volume()
        self.all_volumes_cm3 = self.all_volumes*2.54**3

class Kresling_Volume_Single:
    """
    Class for calculating the volume of a kresling for a single twisting angle
    """
    def __init__(self, Kresling_Tube, twisting_angle = None):
        """
        Initialization

        :param Kresling_Tube: A Kresling object to calculate volume for
        :param twisting_angle: Single twisting angle (degrees) for the calculation
        """
        self.Kresling_Tube = Kresling_Tube
```

```
        if twisting_angle is None:
            self.twisting_angle = (Kresling_Tube.psi2-Kresling_Tube.psi1)*180/np.pi
        else:
            self.twisting_angle=twisting_angle

        self.H =
self.Kresling_Tube.calculate_height(self.twisting_angle*np.pi/180+self.Kresling_Tube.psi1)

    def calculate_volume(self):
        """
        Returns the volume of the kresling for this twisting angle
        """
        if self.H < 1e-8: return 0
        return quad(self.calculate_area, 1e-8, self.H)[0]

    def calculate_area(self, h):
        """
        Returns the cross-sectional area at height h

        :param h: Height of cross-section
        """
        ngon_area =
self.Kresling_Tube.n*self.Kresling_Tube.b**2/(4*np.tan(np.pi/self.Kresling_Tube.n))
        returnVal =
ngon_area-self.Kresling_Tube.n*self.Kresling_Tube.b*(self.Kresling_Tube.r*np.cos(np.pi/self.
Kresling_Tube.n)-self.Kresling_Tube.calculate_eff_r(self.twisting_angle, h/self.H))
        return returnVal


class Kresling:
    """
    Object that holds kresling parameters and computes simulations
    """
    def __init__(self, n, b, unit_count, H, H0, chirality = 'A'):
        """
        Initialization

        :param n: N-gon amount
        :param b: Edge length
        :param unit_count: Number of kresling units
        :param H: The extended stable height
        :param H0: The retracted stable height
        :param chirality: String of 'A' and 'B' representing the chirality of the kresling
        """
        self.n = n
        self.b = b
        self.H0 = H0/unit_count
        self.H = H/unit_count
        self.unit_count = unit_count
        self.chiralities = self.generate_chiralities(chirality)
        self.x1 = self.calculate_x1()
        self.x2 = self.calculate_x2()

        if np.isnan(self.x1) or np.isnan(self.x2):
            raise ValueError("Invalid configuration")

        self.alpha = self.calculate_alpha()
        self.a = self.calculate_a()
        self.c = self.calculate_c()

        self.psi1 = 2*np.atan(self.x1)
        self.psi2 = 2*np.atan(self.x2)

        self.r = self.b/(2*np.sin(np.pi/n))
        self.l = np.sqrt(self.H**2-2*self.r**2*np.cos(self.psi1)+2*self.r**2)

        self.layers = self.generate_3d()

    def generate_chiralities(self, chirality):
        """
```

```
        Converts the chirality string into an array of booleans, where true is a CCW twist

        :param chirality: String of 'A' and 'B' representing the chiralities of the kresling
        """
        chirality_tessilated = chirality
        while len(chirality_tessilated) < self.unit_count:
            chirality_tessilated += chirality

        chirality_tessilated = chirality_tessilated[:self.unit_count]

        chiralities = []
        for char in chirality_tessilated:
            chiralities.append(char == 'A')

        return np.array(chiralities)

    def calculate_x1(self):
        """
        Calculates the x1 parameter for the kresling
        """
        rn = np.pi/self.n
        rH = self.H/self.b
        rH0 = self.H0/self.b
        num =
2*np.sin(rn)*(np.sin(rn)*np.sqrt((1/np.tan(rn))**2*(1/np.sin(rn))**2-(rH**2-rH0**2)**2)-np.c
os(rn))
        den = 1 + rH**2-rH0**2+(1-rH**2+rH0**2)*np.cos(2*rn)
        return num/den

    def calculate_x2(self):
        """
        Calculates the x2 parameter for the kresling
        """
        rn = np.pi/self.n
        rH = self.H/self.b
        rH0 = self.H0/self.b
        num =
2*np.sin(rn)*(np.sin(rn)*np.sqrt((1/np.tan(rn))**2*(1/np.sin(rn))**2-(rH**2-rH0**2)**2)-np.c
os(rn))
        den = 1 - rH**2 + rH0**2 + (1 + rH**2 - rH0**2)*np.cos(2*rn)
        return num/den

    def calculate_alpha(self):
        """
        Calculates the alpha of the kresling
        """
        rn = np.pi/self.n
        rH0 = self.H0/self.b
        num = self.x2*(self.x2-1/np.tan(rn))
        den = np.sqrt((self.x2**2+1)*(rH0**2*(self.x2**2+1)+self.x2**2*(1/np.sin(rn))**2))
        return np.acos(num/den)

    def calculate_a(self):
        """
        Calculates a for the kresling
        """
        rn = np.pi/self.n
        rH0 = self.H0/self.b
        return self.b*np.sqrt(rH0**2+self.x2**2/((np.sin(rn))**2*(self.x2**2+1)))

    def calculate_c(self):
        """
        Calculates c for the kresling
        """
        rn = np.pi/self.n
        rH0 = self.H0/self.b
        num =
self.b*np.sqrt(rH0**2*(self.x2**2+1)**2+self.x2**3*1/np.tan(rn)*(self.x2*1/np.tan(rn)+2)+sel
f.x2**2)
        return num/(self.x2**2+1)
```

```
    def calculate_eff_r(self, twisting_angle = None, ratio = 1/2):
        """
        Calculates the minimum distance of the kresling from the center at a specific height

        :param twisting_angle: The angle the kresling is twisted at (degrees) from rest.
Must be less than psi2-psi1
        :param ratio: The height at which this cross-section is calculated. 1/2 is default
as this is the minimum effective r for any given configuration
        """
        if twisting_angle is None:
            phi = (self.psi2)
        else:
            phi = twisting_angle*np.pi/180 + self.psi1
        gamma = (2*np.pi/self.n)/2
        theta = 2*gamma+phi

        return
self.r*(np.cos(ratio*phi)*((1-ratio)*np.cos(gamma)+ratio*(np.cos(gamma)*np.cos(theta)+np.sin
(theta)*np.sin(gamma)))+np.sin(ratio*phi)*(-1*(1-ratio)*np.sin(gamma)+ratio*(np.cos(gamma)*n
p.sin(theta)-np.sin(gamma)*np.cos(theta))))


    def calculate_twisting_angle(self, h):
        """
        Calculates the twisting angle (radians) for any arbitrary kresling height between H0
and H
        This value will fall between psi1 and psi2
        Equation source:
https://www.sciencedirect.com/science/article/pii/S2095034924000576#
        l^2 = h^2 - 2r^2cos(phi)+2r^2
        h^2 + 2r^2 - l^2 = 2r^2cos(phi)
        phi = arccos(h^2/(2r^2)+1-l^2/(2r^2))

        :param h: Height for twisting angle calculation
        """
        return np.acos(1+(h**2-self.l**2)/(2*self.r**2))

    def calculate_height(self, psi):
        """
        Calculates the height of the kresling based on a twisting angle

        :param psi: Twisting angle (radians). Should be between psi1 and psi2
        """
        return np.sqrt(self.l**2 + 2*self.r**2*np.cos(psi)-2*self.r**2)



    def cutting_pattern(self, draw_funct, space):
        """
        Generates a cutting pattern of the kresling

        :param draw_funct: Function that draws the cutting pattern based on points. This
currently supports matplotlib and ezdxf
        :param space: The draw space. For matplotlib it is an axs, for ezdxf it is a
modelspace
        """
        base_points = np.array([[x*self.b, 0] for x in range(0, self.n+1)])
        translation = np.array([self.a*np.cos(np.pi-self.alpha),
self.a*np.sin(np.pi-self.alpha)])
        internal_angle = (np.pi-(self.n-2)*np.pi/self.n)/2
        tab_length = self.b/2
        tabs = np.zeros((3*(self.n)+1, 2))
        ind = 0
        for i in range(0, self.n):
            tabs[ind] = base_points[i]
            ind += 1
            tabs[ind] = [base_points[i, 0]+tab_length*np.cos(internal_angle),
-1*tab_length*np.sin(internal_angle)]
            ind += 1
```

```
            tabs[ind] = [base_points[i+1, 0]-tab_length*np.cos(internal_angle),
-1*tab_length*np.sin(internal_angle)]
            ind += 1
        tabs[ind] = base_points[-1]



        for i in range(0, self.unit_count):
            draw_funct(space, base_points[:, 0], base_points[:, 1], color = 1)
            draw_funct(space, tabs[:, 0], tabs[:, 1], color = 0)
            translation_dir = 1 if self.chiralities[i] else -1
            temp_translation = translation * np.array([translation_dir, 1])
            new_points = base_points+temp_translation
            tab_point = (base_points[-1]+new_points[-1])/2
            tab_offset_dir = new_points[-1]-base_points[-1]
            tab_offset_dir = [tab_offset_dir[1],
-1*tab_offset_dir[0]]/np.sqrt(tab_offset_dir[0]**2+tab_offset_dir[1]**2)
            tab_point = tab_point + self.b/2*tab_offset_dir
            draw_funct(space, [base_points[-1, 0], tab_point[0], new_points[-1, 0]],
[base_points[-1, 1], tab_point[1], new_points[-1, 1]], color = 0)
            for n in range(0, self.n+1):
                if n != 0:
                    edge_color = 1
                else:
                    edge_color = 0
                draw_funct(space, [base_points[n, 0], new_points[n, 0]], [base_points[n, 1],
new_points[n, 1]], edge_color)
                if not n == self.n:
                    if self.chiralities[i]:
                        draw_funct(space, [base_points[n, 0], new_points[n+1, 0]],
[base_points[n, 1], new_points[n+1, 1]], color = 2)
                    else:
                        draw_funct(space, [base_points[n+1, 0], new_points[n, 0]],
[base_points[n+1, 1], new_points[n, 1]], color = 2)
            base_points = np.copy(new_points)
        draw_funct(space, base_points[:, 0], base_points[:, 1], color = 1)
        tabs[:, 1] = tabs[:, 1]*-1
        tabs = tabs+base_points[0]
        draw_funct(space, tabs[:, 0], tabs[:, 1], color= 0)


    def plot_cutting_pattern(self, saveName = None):
        """
        Function for plotting the cut pattern with matplotlib

        :param saveName: Save file name
        """
        def plot_funct(axs, x, y, color):
            colors = [(0, 0, 0), (1, 0, 0), (0, 0, 1)]
            axs.plot(x, y, color = colors[color])
        fig, axs = plt.subplots()
        draw_funct = plot_funct
        self.cutting_pattern(draw_funct, axs)
        axs.set_aspect('equal')
        axs.axis('off')
        if not saveName is None:
            plt.savefig(saveName, format="svg", bbox_inches='tight', transparent=True)

    def dxf_cutting_pattern(self, saveName):
        """
        Function for creating a dxf of the cut pattern with ezdxf

        :param saveName: Save file name
        """
        def dxf_funct(msp, x, y, color):
            colors = [7, 1, 5]
            for i in range(0, len(x)-1):
                msp.add_line((x[i], y[i]), (x[i+1], y[i+1]), dxfattribs={"color":
colors[color]})
        doc = ezdxf.new("R2018")
```

```
        doc.units = units.IN
        msp = doc.modelspace()
        draw_funct = dxf_funct
        self.cutting_pattern(draw_funct, msp)
        doc.saveas(saveName)

    def generate_3d(self):
        """
        Generates an array of Layers of a kresling. This is useful for plotting in 3D
        """
        layers_3d = []
        angle_sum = 0
        for i in range(self.unit_count+1):
            layers_3d.append(Layer(self, i, angle_sum))
            if not i == self.unit_count:
                angle_sum += self.psi1 if self.chiralities[i] else -1*self.psi1
        return layers_3d

    def plot_3d_angle(self, twist_angle, Volume_Obj = None, saveName = None):
        """
        Plots the kresling based on a twist angle

        :param twist_angle: Twist angle from rest
        :param Volume_Obj: An object for plotting the volume of the kresling relative to
twist angle
        :param saveName: Name for saving the frames
        """
        fig = plt.figure(figsize=(12, 6))
        fig.suptitle("Kresling Folding Simulation")
        ax_count = 111 if Volume_Obj is None else 121
        ax1 = fig.add_subplot(ax_count, projection='3d')
        ax1.set_title("Kresling View")
        ax1.view_init(elev=45, azim=30)
        ax1.set_axis_off()

        activated_units = self.chiralities
        backwards = False
        if twist_angle < 0:
            backwards = True
            activated_units = np.logical_not(activated_units)
        activated_units = np.where(activated_units)[0]
        individual_angle = np.abs((twist_angle)/len(activated_units)*np.pi/180)
        if individual_angle > (self.psi2-self.psi1):
            raise ValueError(f"Twist angle is too large: {twist_angle} is greater than max
twist angle {(self.psi2-self.psi1)*180/np.pi*len(activated_units)}")
        h = self.calculate_height(individual_angle + self.psi1)
        accumulated_angle = 0
        accumulated_height = 0
        for i in range(self.unit_count+1):
            base_points = self.layers[i].twisted(accumulated_angle, accumulated_height)
            base_points_closed = np.append(base_points, [base_points[0]], axis = 0)
            ax1.plot(base_points_closed[:, 0], base_points_closed[:, 1],
base_points_closed[:, 2], color = 'k')
            if i == self.unit_count: break
            if i in activated_units:
                accumulated_angle += individual_angle if not backwards else
-1*individual_angle
                accumulated_height += h
            else:
                accumulated_height += self.H
            top_points = self.layers[i+1].twisted(accumulated_angle, accumulated_height)

            for n in range(len(base_points)):
                ax1.plot([base_points[n][0], top_points[n][0]], [base_points[n][1],
top_points[n][1]], [base_points[n][2], top_points[n][2]], color = 'k')


            for n in range(len(base_points)):
                if self.chiralities[i]:
                    next_ind = n+1
```

```
                    if next_ind >= len(base_points):
                        next_ind = 0
                else:
                    next_ind = n-1
                    if next_ind < 0:
                        next_ind = len(base_points)-1
                ax1.plot([base_points[n][0], top_points[next_ind][0]], [base_points[n][1],
top_points[next_ind][1]], [base_points[n][2], top_points[next_ind][2]], color = 'k',
linestyle = '--')


        ax1.set_xlim(-1*self.r, self.r)
        ax1.set_ylim(-1*self.r, self.r)
        ax1.set_zlim(0, self.H*self.unit_count)
        ax1.set_aspect('equal')

        if not Volume_Obj is None:
            ax2 = fig.add_subplot(122)
            ax2.set_title("Kresling View")
            ax2.set_xlabel("Twist Angle from Rest (deg)")
            ax2.set_ylabel("Volume (cm3)")
            ax2.grid()
            ax2.scatter([Volume_Obj.twisting_angles[0], Volume_Obj.twisting_angles[-1]],
[Volume_Obj.all_volumes_cm3[0], Volume_Obj.all_volumes_cm3[-1] if not
np.isnan(Volume_Obj.all_volumes_cm3[-1]) else 0], color = 'k', marker = 'o', label =
'Volume')
            ax2.plot(Volume_Obj.twisting_angles, Volume_Obj.all_volumes_cm3, color = 'k',
linestyle = '--', label = 'Volume (Approximate)')
            ax2.scatter(twist_angle,
Volume_Obj.all_volumes_cm3[np.where(Volume_Obj.twisting_angles == twist_angle)[0][0]],
marker = 'o', color = 'b', label = 'Current Volume')
            ax2.legend()




        if not saveName is None:
            plt.savefig(saveName)

        plt.close()

    def plot_twisting_video(self, save_dir_name, time, fraction_collapse = 1, bounce =
True):
        """
        Creates a full twisting video

        :param save_dir_name: Directory to save the frames and result
        :param time: How long the video should be
        :param fraction_collapse: What fraction of the full kresling collapse should be made
        :param bounce: Whether to append all frames in reverse at the end of the video
        """
        save_dir = save_dir_name
        copies = 0
        while (os.path.exists(os.path.join(os.getcwd(), save_dir))):
            copies += 1
            save_dir = f"{save_dir_name}({copies})"

        frameDirFull = os.path.join(os.getcwd(), save_dir, 'frames')

        os.makedirs(frameDirFull)

        fps = 32

        save_name = "Kresling_Twist"

        frame_count = fps*time

        digit_count = int(np.log10(frame_count))+1

        angle_start = 0
```

```
        angle_end =
(self.psi2*np.sum(self.chiralities)*fraction_collapse-self.psi1*np.sum(self.chiralities))*18
0/np.pi

        angles = np.linspace(angle_start, angle_end, frame_count)
        theVolumes = Kresling_Volume_All(self, angles)

        for frame_index in range(frame_count):

            fileName = save_name+str(frame_index).zfill(digit_count)+".png"

            filePath = os.path.join(os.getcwd(), frameDirFull, fileName)
            self.plot_3d_angle(angles[frame_index], theVolumes, saveName=filePath)

        video_name = os.path.join(save_dir, f'{save_name}_video.mp4')

        images = [img for img in os.listdir(frameDirFull) if img.endswith(".png")]
        images_all = images
        if bounce:
            for i in range(len(images)-2, -1, -1):
                images_all.append(images[i])

        frame = cv2.imread(os.path.join(frameDirFull, images_all[0]))
        height, width, layers = frame.shape

        video = cv2.VideoWriter(video_name, 0, fps, (width,height))

        for image in images_all:
            video.write(cv2.imread(os.path.join(frameDirFull, image)))

        cv2.destroyAllWindows()
        video.release()


class Layer:
    """
    Object for storing a Layer of a kresling tube. Each layer is an n-gon which can be
connected to layers above and below
    This is used for plotting in 3D
    """
    def __init__(self, Kresling_Tube, ind, base_angle):
        """
        Initialization

        :param Kresling_Tube: Kresling object
        :param ind: What layer index this layer should be
        :param base_angle: How rotated the layer is when at rest
        """
        self.points = generateNGon(Kresling_Tube.n, Kresling_Tube.b, in3D=True)
        self.ind = ind
        self.base_angle = base_angle
        # self.points = self.points + np.array([0, 0, Kresling_Tube.H*ind])

        self.points = planar_rotation(self.points.T, self.base_angle).T

    def twisted(self, angle, height):
        return planar_rotation(self.points.T, angle).T + np.array([0, 0, height])


# Inches
H = 2.25
b = 1.85

# Plotting all of the possible H values
all_Hs = np.linspace(2, 3.30, 30)
all_radii = np.zeros(30)
all_alphas = np.zeros(30)
for i in range(0, len(all_Hs)):
    theKresling = Kresling(6, b, 1, all_Hs[i], 0, 'A')
    all_radii[i] = theKresling.calculate_eff_r()
```

```
    all_alphas[i] = theKresling.alpha*180/np.pi

# Converts to cm
plt.plot(all_Hs*2.54, all_radii*2.54, color = 'k')
plt.vlines(5.72, np.min(all_radii)*2.54, np.max(all_radii)*2.54, color = "k", linestyles =
':', label = "H = 5.72 cm")
plt.vlines(b*2.54/np.tan(np.pi/6), np.min(all_radii)*2.54, np.max(all_radii)*2.54, color =
"k", linestyles = '--', label = "H = b/tan(pi/n)")

plt.ylim(0, np.max(all_radii)*2.54)
plt.legend()
plt.suptitle("Minimum Effective Radius (cm) vs Extended Length H (cm)")
plt.xlabel("Extended Length H (cm)")
plt.ylabel("Minimum Effective Radius (cm)")
plt.grid()



theKresling = Kresling(6, b, 1, H, 0, 'A')

print(theKresling.alpha*180/np.pi)
print(theKresling.calculate_eff_r()*25.4*2)
print(f"a = {theKresling.a}, c = {theKresling.c} alpha = {theKresling.alpha*180/np.pi}")

theKresling.dxf_cutting_pattern('final_Pattern.dxf')
theKresling.plot_twisting_video("Final Volume Demonstration", 2)
```

Appendix 1: Code used to generate theory