THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

ANALYZING THE EFFECT OF ADDITIONAL COMPLIANT JOINTS ON THE SWIMMING EFFICIENCY OF A LAMINATED ORIGAMI ROBOT

KYLIE MARIE BARBER
SPRING 2024

A thesis
submitted in partial fulfillment
of the requirements
for baccalaureate degrees
in Mechanical Engineering
with honors in Mechanical Engineering

Reviewed and approved* by the following:

Mary Frecker
Professor of Mechanical Engineering
Thesis Supervisor

Margaret Byron
Assistant Professor of Mechanical Engineering
Honors Adviser

*Signatures are on file in the Schreyer Honors College.

Abstract

Compliant origami swimming robots have many applications, from search-and-rescue to underwater discovery. Using manufacturing techniques such as laser cutting and additive manufacturing, compliant origami swimming robots can be manufactured quickly and with smaller expense than typical rigid robots. However, they often experience issues with efficiency and swimming speed. This thesis presents a compliant origami swimming robot which improves its efficiency through the addition of a fin joint. It was determined that the positioning of the fin joint affected the achievable velocity, thrust, and efficiency.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank the many people and organizations that have helped me complete this thesis. First, I would like to thank my co-researcher and twin sister, Kaylie Barber, who helped me develop this swimming robot along with Dr. Temel and the Zoom lab at Carnegie Mellon University. Kaylie both supported me in my research at Carnegie Mellon University and offered valuable advice to me throughout the development of my thesis, and I am grateful for her support.

Additionally, I would like to thank Dr. Frecker, my thesis supervisor, along with the rest of the Engineering Design and Optimization Group, for their support as I learned the research process. Her guidance and support allowed me to substantially improve the robot I designed at Carnegie Mellon, perform testing on this robot, and present my results.

For review of my thesis, I would like to thank Dr. Byron and Dr. O'Connor for their dedicated assistance in helping me format and edit my thesis.

In regards to testing, I would like to thank Kundan Panta in Dr. Cheng's lab, BioRob-InFL at the Pennsylvania State University, for helping to provide me with the resources necessary to test my robot. Kundan provided me with image processing capabilities and testing equipment, helping me to test my robot, and I am grateful for his assistance.

With respect to manufacturing, I would like to thank The Learning Factory, Penn State's engineering makerspace, for providing me with access to laser cutters in order to manufacture my robot.

Lastly, I would like to thank the students who supported me through data processing, writing groups, and deadline checks. These students include Scout Bucks, Conor Savage, William Worrall, Evan Smith, and Dominic Marazita. I am grateful for all of their peer support during my writing and data processing.

Chapter 1

Literature Review

1.1 Overview

Origami, the art of folding paper, has been increasingly applied in engineering. Using origami in engineering allows for unique geometries to be created using novel manufacturing methods which can introduce compliance into designs. Applications of origami in engineering include manufacturing compliant swimming robots. By applying origami engineering to manufacture swimming robots, swimming robots can be created more efficiently and with unique geometry and properties. This thesis examines the designing, prototyping, and testing of a compliant, origami swimming robot created through a laser cutting lamination manufacturing procedure.

1.2 Origami in Engineering

Origami is an ancient art that involves folding paper to create creases which can act as joints for movement [1]. When origami is used for movement, it is considered to be action origami. However, origami can be applied to engineering in addition to art. Origami engineering allows for low cost manufacturing processes and unique geometries. Once folded into configurations, some origami mechanisms are dynamic and can exhibit motion that moves them from their final folded states. [2–4]. These origami mechanisms act as compliant mechanisms which allow motion along the creases of the pattern [5, 6]. Kirigami, a derivative of origami, similarly allows for motion along the creases of the pattern. However, in kirigami, the material can be cut. While the work of this research thesis technically covers kirigami due to cutting being used, the term origami is generally used when discussing origami or kirigami engineering work. Therefore, the broader definition of origami as encompassing kirigami will be used in this thesis.

1.3 Manufacturing Origami Engineering Prototypes

Methods of manufacturing origami engineering prototypes include laser cutting [2, 4], casting [7], and 3D printing [3]. Laser cutting is often an effective manufacturing choice for creating

origami engineering prototypes and products because it allows for low cost manufacturing and rapid prototyping. Laser cutting is a process which uses a high-powered laser to burn through material. It is a two-dimensional process that operates within an x-y axis system following lines sent from a drawing program. Different settings on the laser can be used to adjust speed, power, and frequency depending on the material that is being cut [8]. Within the manufacturing method of laser cutting, origami joints are created by removing material. This material removal can consist of cutting away material from a single sheet as done in [4], or it can involve lamination techniques in which rigid layers of material are adhered to a flexible layer of material which acts as a joint as in [2, 9].

Origami in engineering often requires thickness accommodation for folding. Because the material used in origami engineering is not infinitely thin as it is assumed to be when modeling traditional origami with paper, the material stacks on itself at vertices. When multiple panels meet at a vertex, this thickness interference is heightened. The stacking, or thickness interference, that occurs at vertices prevents the mechanisms from folding completely flat, limiting the motion of the mechanism. Multiple techniques can be applied for thickness accommodation. These techniques can include shifting the joint axis to accommodate angled folds or applying a membrane technique which uses a thinner material at joints [9, 10]. Membrane thickness accommodation is useful because the thinner material used at the joint can often be regarded as negligibly thick compared to the thicker material used for the panels. Therefore, a thicker rigid material can be used on panels, making the origami device more robust, while a thinner flexible material can be used within the joints, allowing the origami device to fold. Depending on the material used as the thin membrane, this thin membrane can add elasticity to the joints, allowing for bistability and spring-like behavior at the joints.

1.4 Swimming Robots

1.4.1 Types of Swimming Robots

Origami engineering has been applied to many different areas of engineering, including swimming robots [11–17]. Origami is well suited for swimming robots because the flexible materials that it is created with are often able to withstand high pressures, such as those at the bottom of the ocean, without showing permanent deformation. Rigid robots, which are made of multiple parts brought together through the use of fasteners, experience high stress-concentration areas at the location of these fasteners. These materials are likely to bend or fracture, causing permanent damage to the robot. Soft materials, such as those that are used when manufacturing origami robots, avoid some of these stress concentrations and permanent deformations.

There are different forms of swimming locomotion that origami robots could be used in. These swimming methods include body/caudal actuation, in which a swimmer uses its body and caudal fins for thrust, and median/paired actuation, in which a swimmer uses its median or paired fins for thrust. These methods are further split into undulatory motion, in which the swimmer uses a wave-like motion, or oscillatory motion, in which the swimmer oscillates its body back and forth. Swimming can be controlled both passively, based on the structure of the object, or actively, based on how appendages or the body are used [18–20]. An additional mode of swimming is jet propulsion. In jet propulsion, a swimmer uses its fins to scoop water and channel that water into a jet which propels the swimmer in the opposite direction of the jet stream's movement. [21, 22]. Choosing the type of swimming motion used for a particular robot involves analyzing the available materials and manufacturing methods as well as the intended application of the robot. Jet propulsion can be especially useful for origami swimming robots because less compliance is needed to create thrust from a jet stream as opposed to creating thrust from undulatory and oscillatory motions.

1.5 Applications of Compliant Swimming Robots

Swimming robots can have many applications, from underwater discovery to search-and-rescue applications. Specifically, compliant swimming robots can be particularly useful because they can be lightweight yet able to withstand high forces, making them particularly useful for high-pressure environments that are found in deep water [7]. While free-swimming, a robot created by researchers at the Zhejiang University in Hangzhou, China was able to withstand the water pressure at a depth of 3,224 m and a depth of 10,500 m when tethered and brought to the seabed [7]. This ability to withstand high pressures was due to the compliant design of the robot, which used adaptive fluid chambers and a dispersed matrix of electronics to decrease the pressure that the electronics experienced [7].

Within areas of application such as search-and-rescue and underwater discovery, it is important to have an efficient robot. These areas of application may require that the robot be untethered and able to carry its own power source so that it can navigate through difficult terrain [7]. Carrying a self-contained power system on a swimming robot can pose many challenges, bringing up issues such as waterproofing and weight. Therefore, creating a robot that is efficient can help to mitigate these issues by reducing the amount of energy that needs to be stored for a given application. Improving the efficiency of swimming robots would greatly increase their ability to be applied in environments external to a lab, which is important for creating robots for actual applications.

1.5.1 Methods of Actuation

There are many different actuation methods for origami swimming robots [11]. These methods include electric motors [12–14], pneumatic actuators [15, 16], and shape memory alloys [17]. Choosing an actuator is often based on the application of the robot, the materials available, and the function that the robot has to perform. For example, having a self-contained electric motor can be easier for transporting than having a pneumatic setup for a robot. These considerations should be considered when designing the actuation system.

1.5.2 Quantification of Swimming

To quantify the quality of different swimming robots, swimming speed and efficiency can be measured and determined. There are various methods that can be used to analyze swimming speed. Methods include image tracking with cameras, laser-based flow velocimetry, and on-board accelerometers. Efficiency is more difficult to measure because it involves analyzing the energy loss of the system. Because it is difficult to measure energy directly, energy loss must often be calculated from measuring other parameters. One method of measuring and quantifying efficiency is comparing the forward progression of a swimming robot as compared to the backward progression of a swimming robot [3] as provided in equation 1.1:

$$\eta = \frac{D_f - D_b}{D_f} \tag{1.1}$$

where η represents efficiency, D_f represents the forward displacement of the robot during the forward stroke, and D_b represents the backwards displacement that occurs when the robot resets for the next forward stroke. Measuring the forward and backward displacements can be done through image processing after using a high speed camera. This definition of efficiency does not take into account any possible lateral motion from the robot or motion that otherwise directs the robot away from its intended path. Further, it does not include any energy loss that occurs through the action of causing the motion. Therefore, while this definition of efficiency allows for a comparison between the effectiveness of the stroke in moving the robot forward and the negative effect of the robot resetting its position to prepare for the next forward stroke, it does not take into account how much energy is input into achieving the stroke as compared to the amount of work that is achieved through that stroke. Nevertheless, measuring efficiency in this way by comparing the distance achieved on the forward stroke and the backward stroke is a way to compare the effectiveness of the swimming stroke and the reset to begin a new stroke, a comparison which can greatly inform the design of different swimming robots' geometry.

In order to be more efficient, origami engineering swimming robots can use bio-inspiration in

their designs. While the list of different bio-inspired designs is extensive for swimming robots, some examples are provided here for reference. For instance, origami robots can be based on fish [7, 23], squids or octopi [2, 24], or other swimming creatures. However, they can also be based on flying creatures such as birds or butterflies because flying, like swimming, involves moving through a fluid and therefore invokes fluid dynamics [25]. Using biology to inspire the design of the robot can provide motivation for the design of geometry, actuation system, and other components of the robot, helping the systems to be more efficient and providing rationale for the design of different systems.

1.6 Prior research

This thesis builds on the work performed by researchers at Carnegie Mellon University in the Zoom Lab [9]. Through that research, a laminated, origami jet propulsion swimming robot, named YoDiFIN, was developed. This robot is shown in Figure 1.1. YoDiFIN is the combination of two different mechanisms, YoFIN, shown in Figure 1.2, and DiamondFIN, shown in Figure 1.3. YoDiFIN swims at a rate of 2.1 cm/s [9]. However, an issue with this robot is its efficiency. Due to the geometry and elasticity of the compliant joints, it was observed that the motion of the robot resetting its stroke causes reverse propulsion, decreasing the efficiency of the robot. My research goal is to improve the efficiency of this robot by adding a compliant joint into the fins of the robot that allows the fins to bend during the reset stroke. A further modification was made to this robot to adjust the closed position of the fins. This adjustment reverses the position of the fins, allowing the robot to provide a larger jet stream when closing and be more efficient when opening.

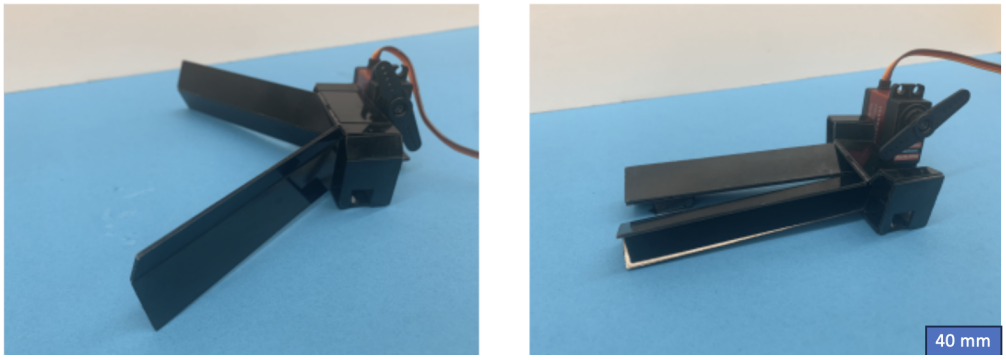

Figure 1.1: YoDiFIN robot.

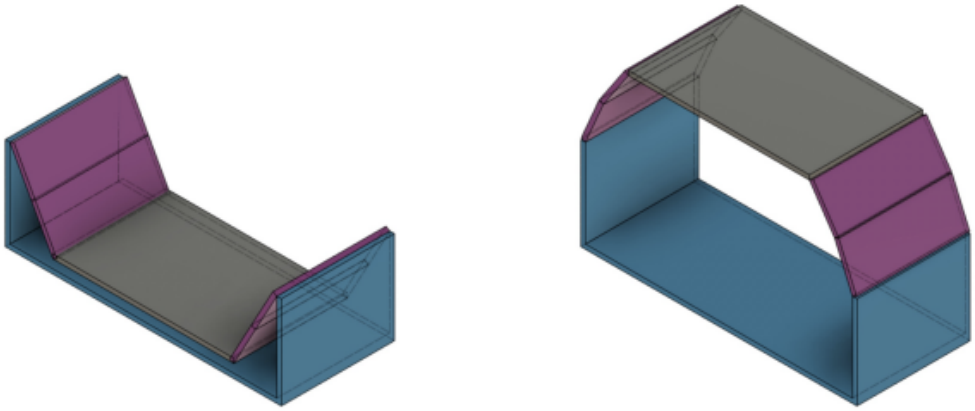

Figure 1.2: YoFIN mechanism.

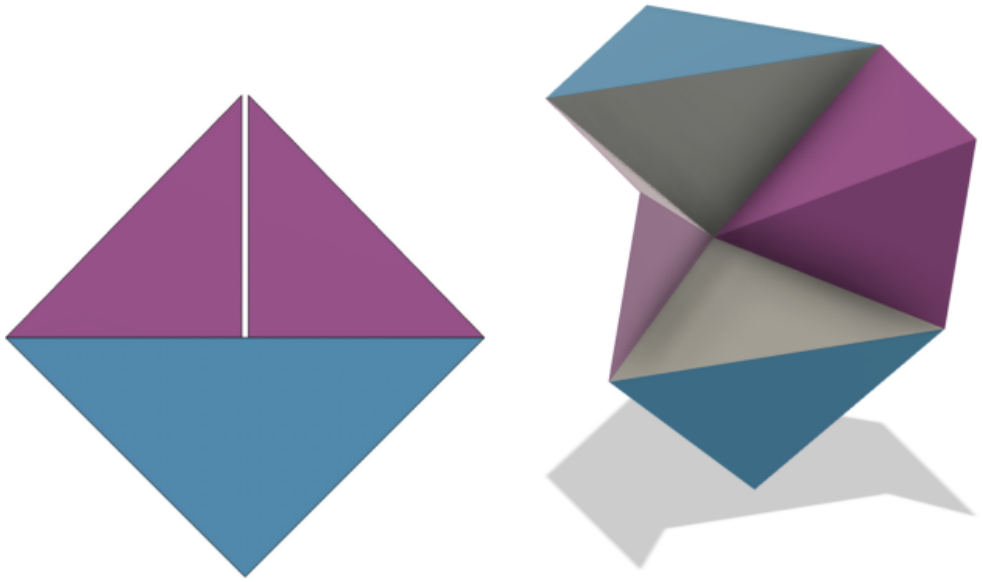

Figure 1.3: DiamondFIN mechanism.

1.7 Identified Research Gaps

This literature review has identified a need for improvements or additions to improve the functionality of swimming robots. Mainly, swimming robots need improvement in efficiency in order to be applied more broadly. Improved efficiency also reduces the power required, which is important for autonomous robots. As reported in the literature, it can be effective to create these swimming robots using soft robotics that incorporate compliant mechanisms and origami in order to benefit from the quick, low-cost manufacturing methods and robustness within high pressure environments. Therefore, this thesis focusses on assessing the impact of a compliant joint design on the efficiency of a laminated origami swimming robot.

Chapter 2

Methology

2.1 Mechanism Design

This section discusses the software and techniques used to create two-dimensional cutting patterns for the origami swimming robot. First, thickness accommodation is discussed followed by computer aided design (CAD) strategies.

Because the panels of the origami robot are created out of thicker materials than paper when prototyped, thickness accommodation should be considered. Thickness accommodation creates joints in the material which allow the robot to fold along the joints without the material interfering with the motion of the robot. In this thesis, thickness accommodation is achieved by separating the design into the layers depicted in Figure 2.1. These layers consist of rigid layers, a flexible layer, and adhesive layers.

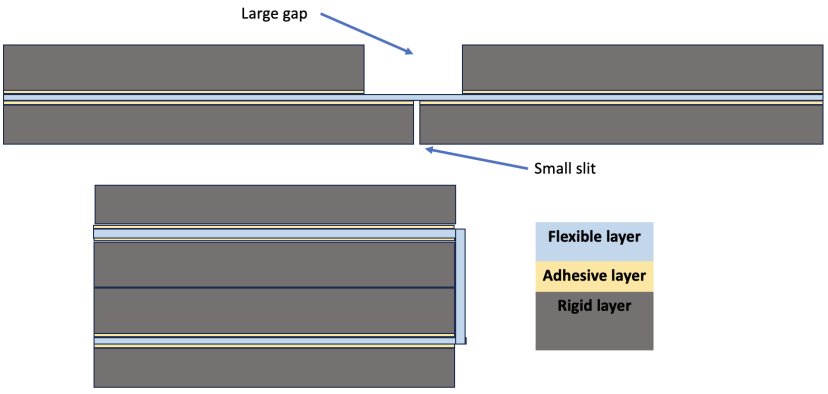

Figure 2.1: Different layers that compose a laminated structure.

The rigid layer provides the robot with structure and actuation surfaces, while the flexible layer provides the robot with compliant joints that allow for folding without bending the rigid panels. Holding these layers together is the adhesive layer, which can be a thin layer of glue or a layer of double-sided tape. Gaps in the rigid layer allow for the rigid layers to fold onto the other rigid layers. Using twice the thickness of the rigid layer as the spacing, the robot is manufactured so that the rigid panels have the ability to fold 180°. Because the flexible layer and adhesive layers are very thin compared to the rigid layers, these layers can be approximated as infinitely thin and therefore

do not require thickness accommodation. There are no gaps in the flexible layer across the surface of the robot, so the flexible layer holds the rigid layers together. To control the direction of each fold, different sized gaps are used in the rigid panels. A small slit is used to introduce flexibility in the rigid material on the outside of the fold, and a larger gap is used to allow the rigid panels to fold towards that larger gap on the inside of the fold.

Converting the crease pattern, shown in Figure 2.2, to cutting patterns is done by putting the larger gaps at valley folds and the smaller gaps at mountain folds. To implement these different gap sizes for folding patterns, CAD is used to sketch the lines and create cutting files. Different sketches within the same CAD file allow each cutting layer of the robot to update as changes are made. There are three different cutting passes on the rigid material that are represented by four sketches: a reference sketch, an inside sketch, an outside sketch, and a final sketch. Three different designs were considered: no fin joint, close fin joint, and far fin joint. The reference sketch, shown in Figures 2.3, 2.4, and 2.5, contains all the lines across the different layers, allowing the other sketches to refer to a common sketch. Therefore, when the design is updated, only the reference sketch is changed to update the rest of the sketches appropriately. Using a reference sketch improves design efficiency and layer consistency because the reference sketch can be adjusted and all layer sketches automatically update to that adjustment. Across all of the cutting line figures, the colors used in these sketches are not significant for determining the types of lines.

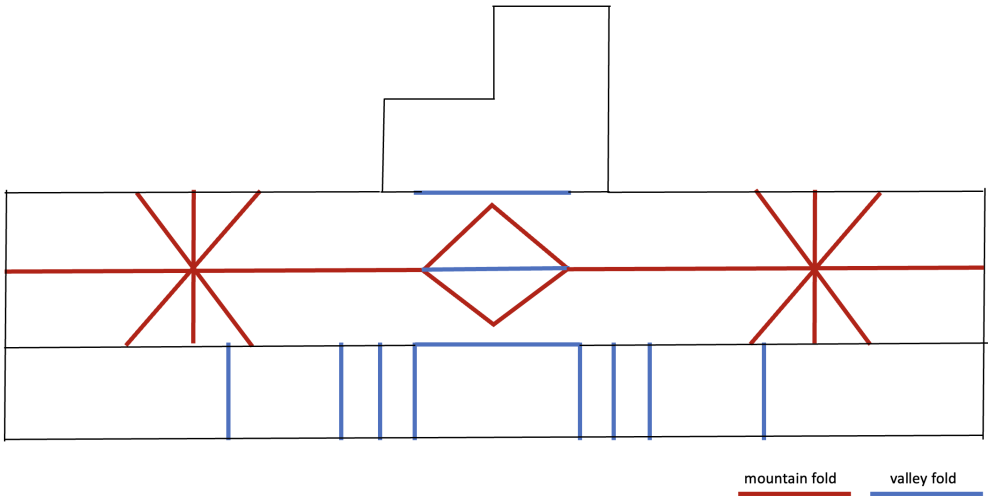

Figure 2.2: Origami pattern for one of the robots created.

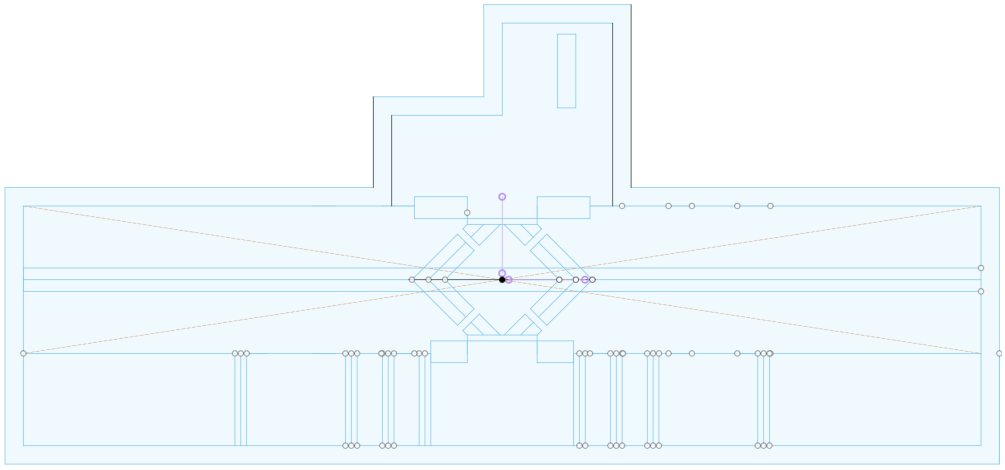

Figure 2.3: Reference sketch for robot with no fin joint.

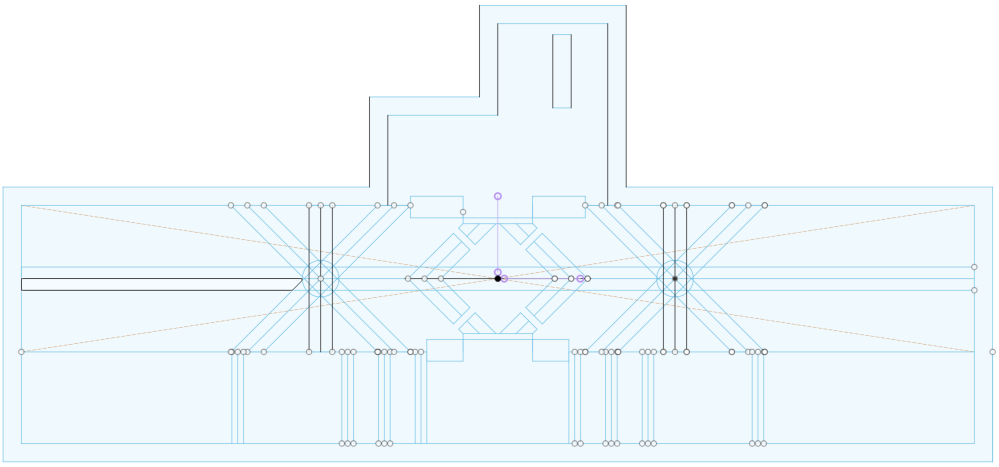

Figure 2.4: Reference sketch for robot with close fin joint.

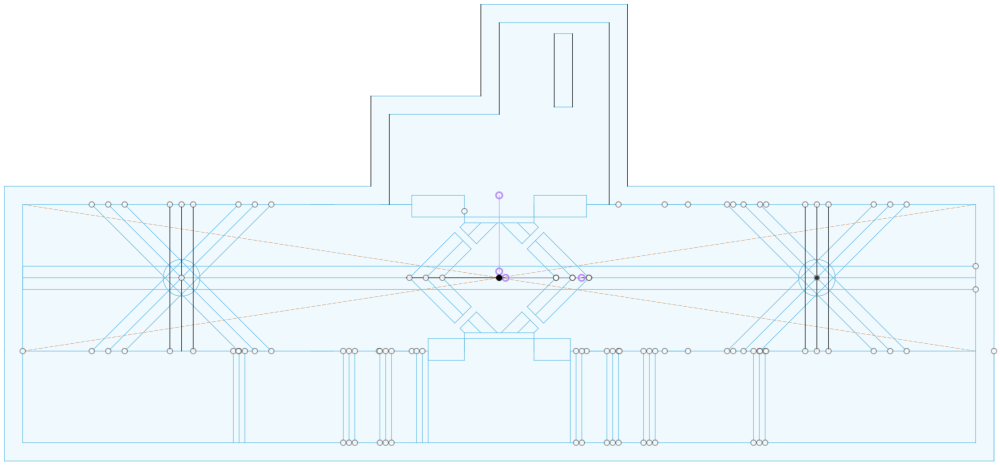

Figure 2.5: Reference sketch for robot with far fin joint.

Next, the inside sketch and outside sketch, shown in Figure 2.6, 2.7, and 2.8, contain the cutting lines for the inner and outer rigid layers respectively before the robot is assembled. These three sketches, reference, inner, and outer, all contain excess offset material outside the joints to hold the rigid panels together and allow for assembly. After assembly, the robot requires an additional final cut to release the joints from this extra material, with these final cutting lines provided in the final sketch as shown in Figure 2.9, 2.10, and 2.11.

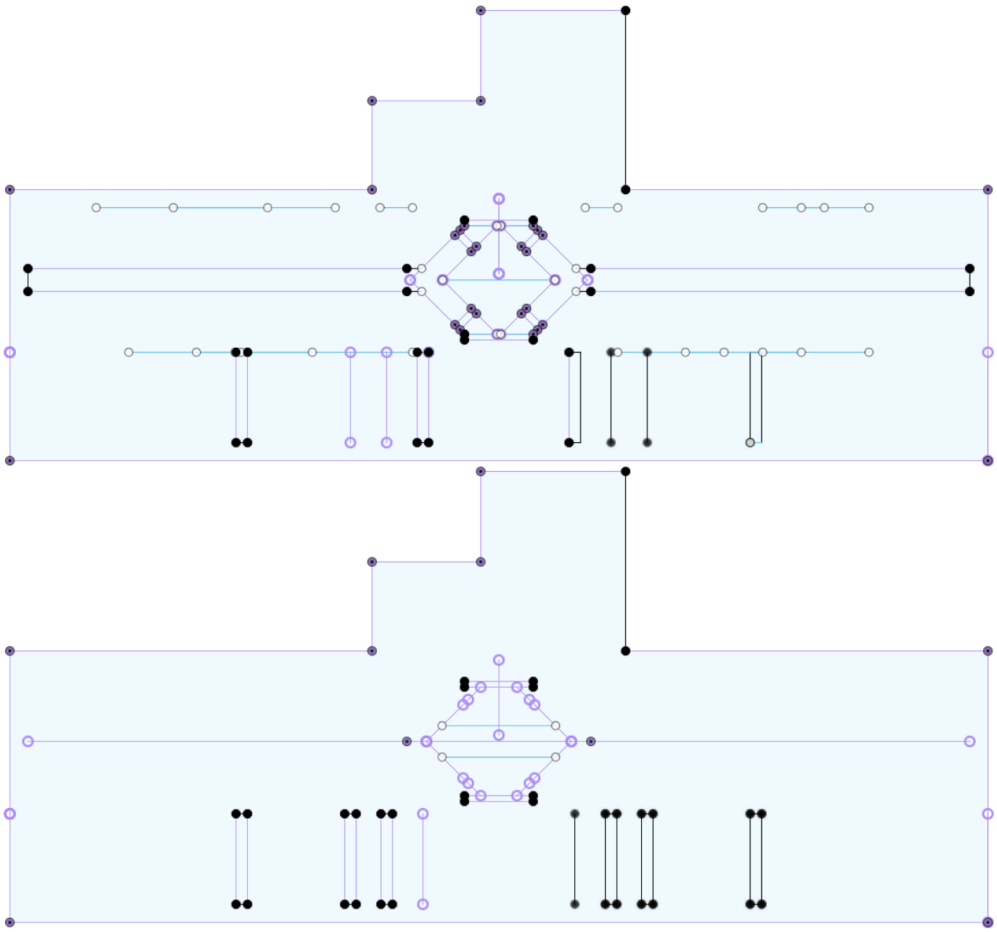

Figure 2.6: Inside and outside sketch for robot with no fin joint.

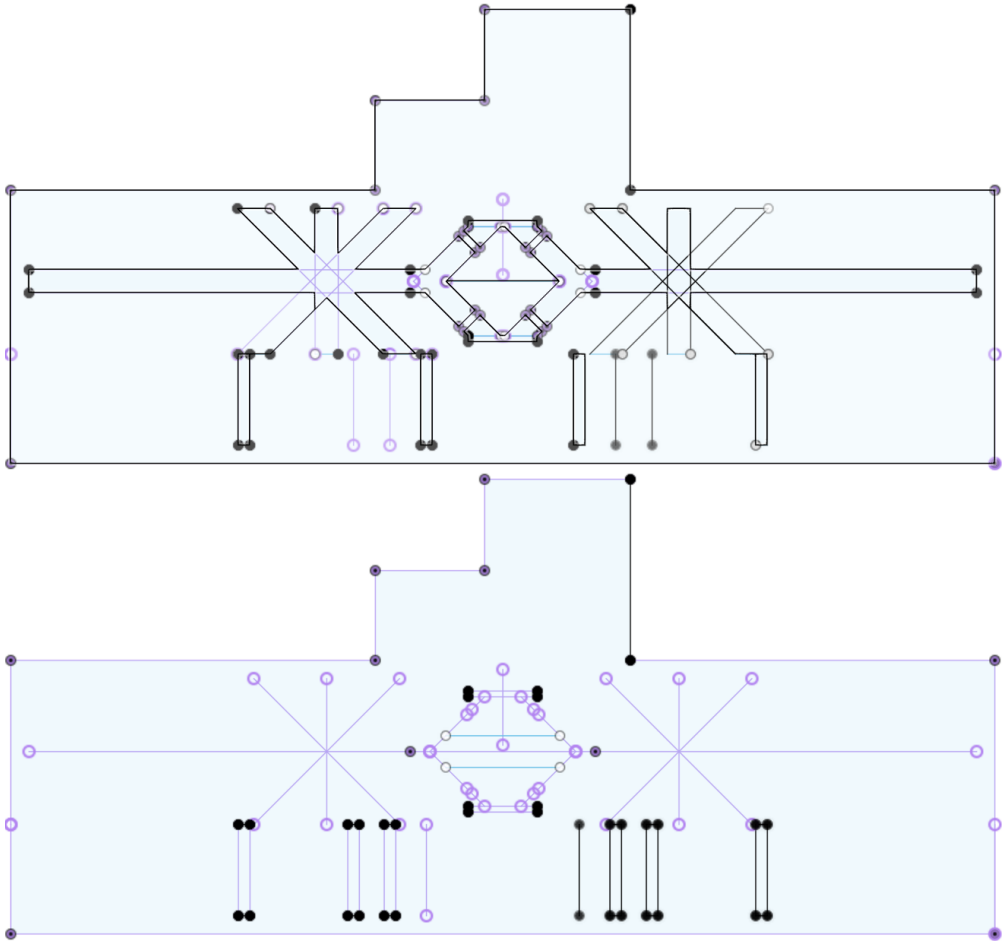

Figure 2.7: Inside and outside sketch for robot with close fin joint.

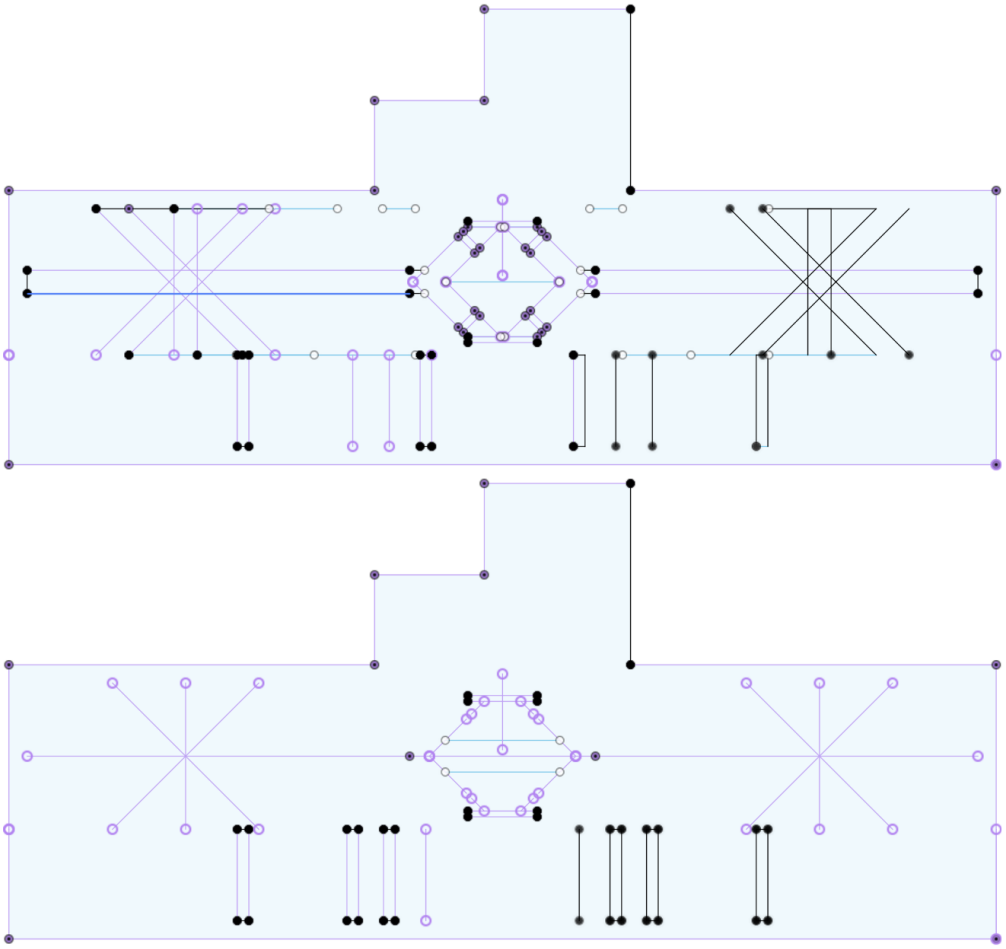

Figure 2.8: Inside and outside sketch for robot with far fin joint.

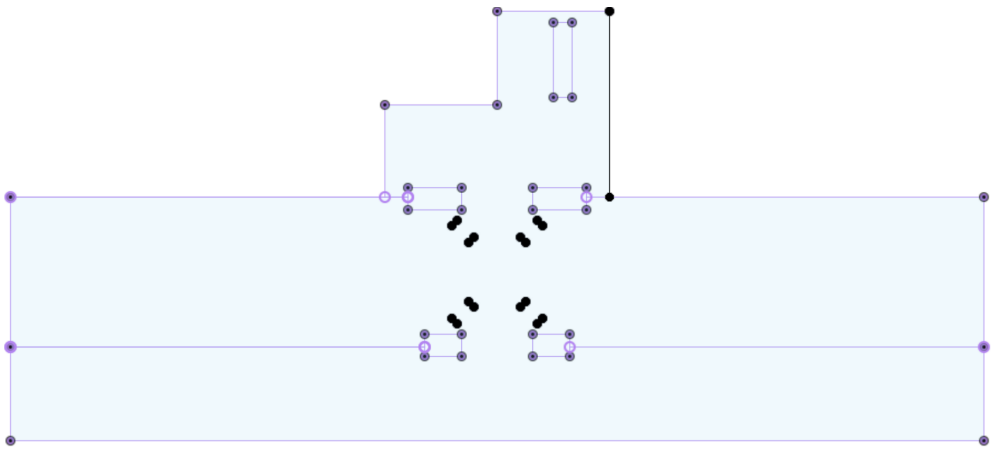

Figure 2.9: Final sketch for robot with no fin joint.

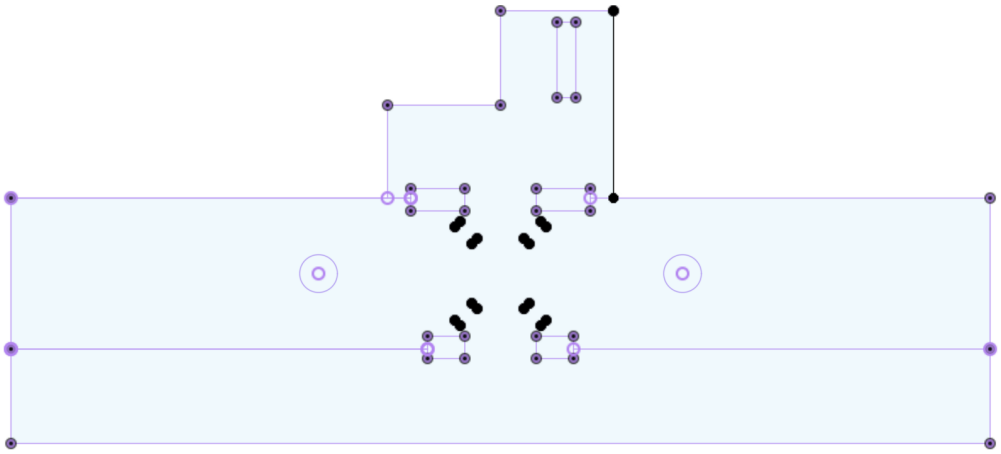

Figure 2.10: Final sketch for robot with close fin joint.

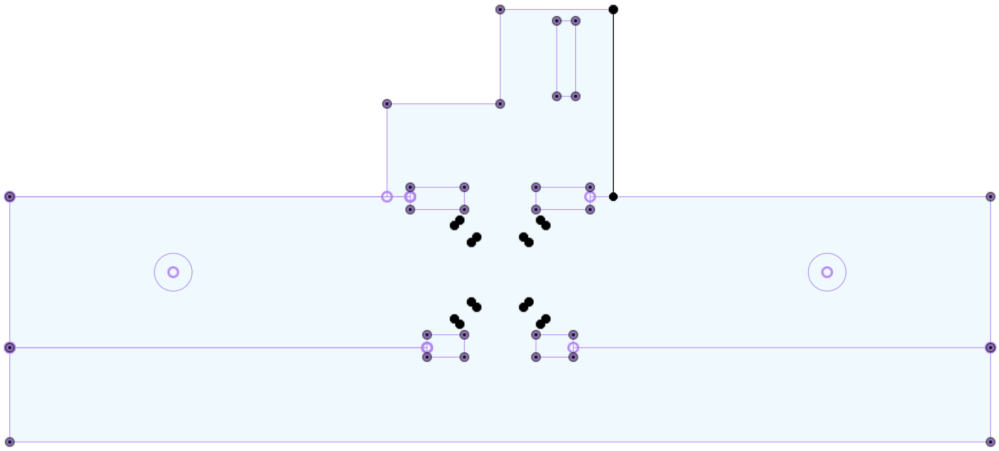

Figure 2.11: Final sketch for robot with far fin joint.

Due to manufacturing constraints, these cutting patterns are two-dimensional and can often be difficult to visualize within the CAD program. Because it can be difficult to determine the location and size of each joint when sketching in CAD, it is useful to fold and unfold the design in paper to observe and measure the location of the folds.

2.2 Mechanism Fabrication

This section details the fabrication process used to create prototypes, which includes laser cutting and assembly.

Crease patterns sketched in CAD are converted to an applicable file format of .ai and sent to laser cutters for manufacturing. To cut the folding patterns, an 80 Watt Epilog Fusion Pro Laser Cutter was used. All layers are cut separately and assembled before taking a final pass on the laser cutter, shown in Figure 2.12. This final pass cuts away the extra material that is holding the joints together, allowing the joints to be flexible.

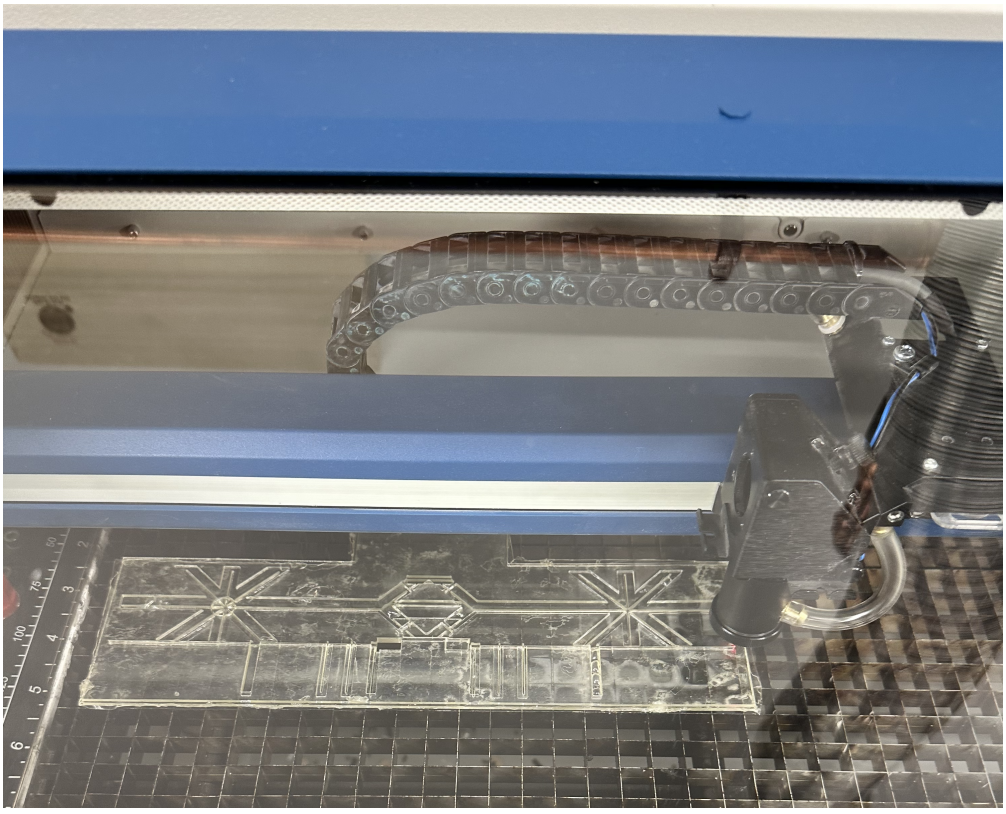

Figure 2.12: Laser cutter with robot.

When aligning the layers, care must be taken to achieve alignment of joints. Imperfections in alignment can cause asymmetries in the mechanism behavior, leading to difficulty in control. Further, misaligned joints can create material interference which counteracts the thickness accommodation discussed earlier in Section 1.3 and Section 2.1. One option for more easily aligning these panels is to use an alignment board. By cutting holes into excess offset material on each layer and using dowel rods to position the layers on the board, layers can be more easily aligned correctly. However, this process requires cutting alignment holes for the dowel rods into the flexible and adhesive layers as well, increasing manufacturing time and material waste. Therefore,

while the alignment board was used for some initial prototypes, the final prototypes were manufactured by visually aligning the joints and then pressing them together freely by hand.

Once the layers are assembled and the final pass is made with the laser cutter to release the joints, the mechanism is folded into its three-dimensional configuration. Once in this configuration, the motor and flotation devices can be attached to the robot. These flotation devices are made out of polyethylene foam, which can be easily cut to affect the buoyancy of the robot. Approximately 1.4 g of polyethylene foam was attached to the robot, shown in Figure 2.13, to allow the robot to float at the surface of the water.

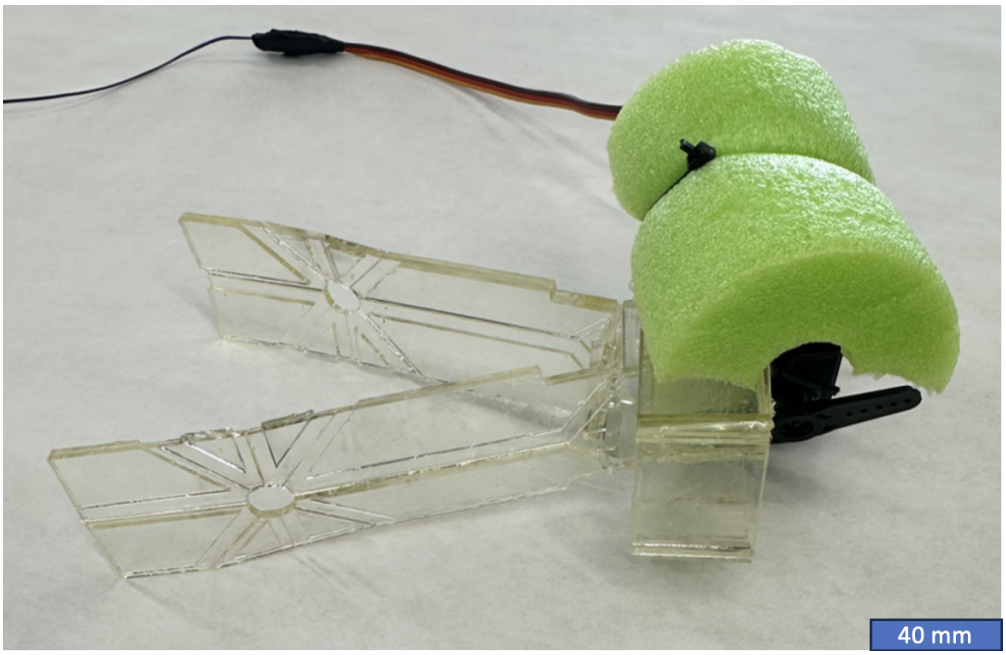

Figure 2.13: Robot set up to run in tank.

2.3 Prototypes

This section discusses the different prototypes that were created for comparison. Three different prototypes were manufactured. One prototype contains no extra joint in the fin, one prototype contains an extra joint in the fin close to the main body of the robot, and the last prototype contains an extra joint in the fin close to the ends of the fins. These robots are shown in Figures 2.14, 2.15, 2.16. When the fin joints were added, a circular hole was included in the center of

the joint to relieve tension in that joint. Fin joint placement was done so that the new joint was as close to the main body of the robot and far from the main body of the robot as possible in order to test both extremes. How close to an edge the joint could be placed was determined by manufacturing constraints because there needed to be enough rigid material to hold the robot together at that joint during manufacturing.

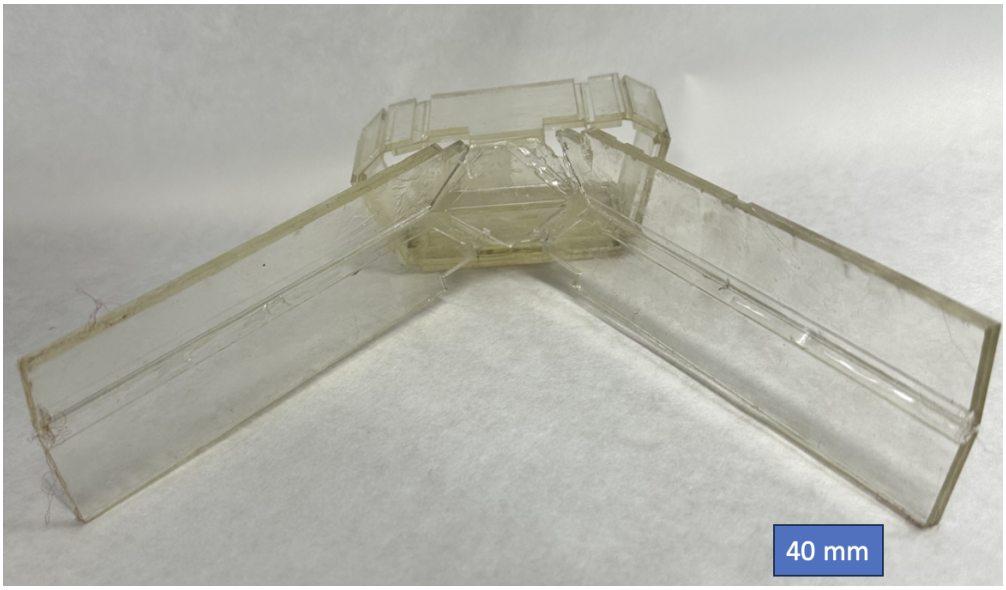

Figure 2.14: Robot with no fin joint in relaxed state.

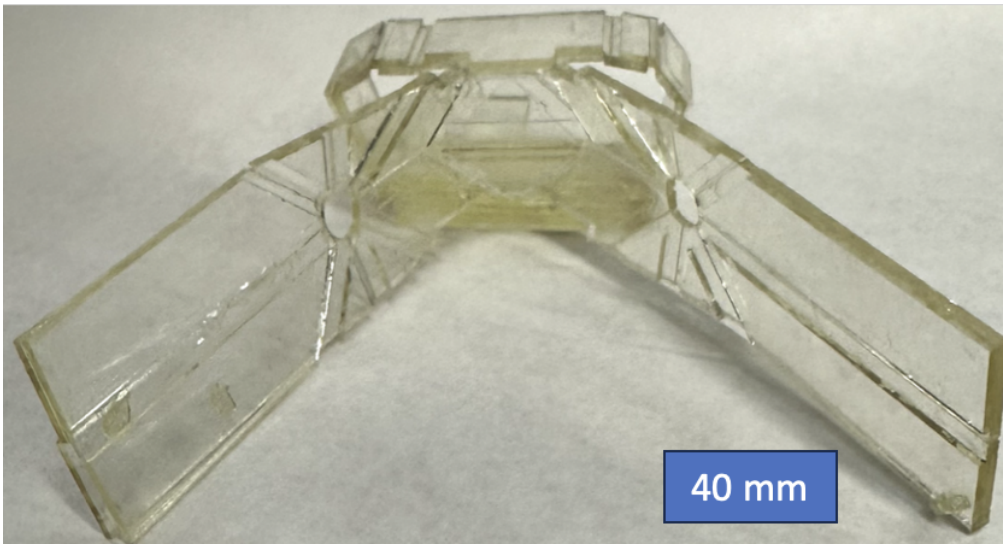

Figure 2.15: Robot with close fin joint in relaxed state.

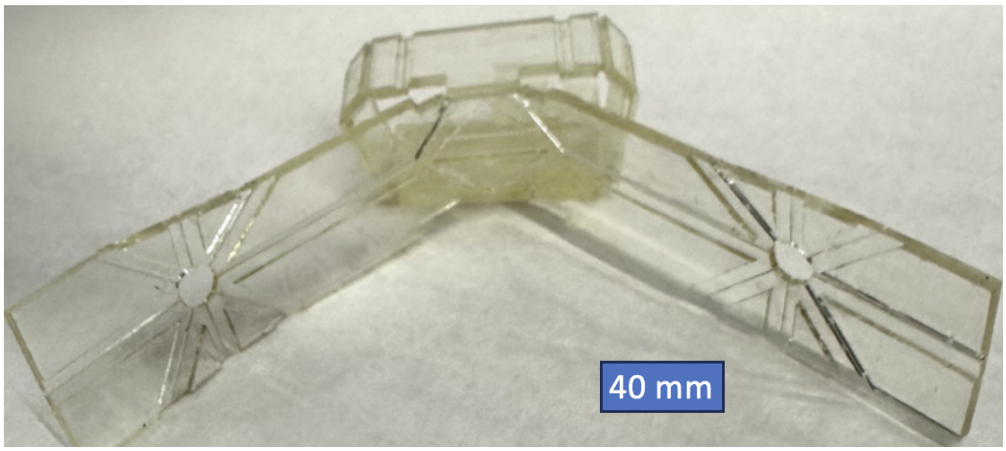

Figure 2.16: Robot with far fin joint in relaxed state.

2.4 Experimental Setup

This section discusses the experimental setup used to conduct experiments and collect data on the robots. Figure 2.17 provides a visual of the experimental setup. A water tank, of dimensions 300 cm by 53 cm with a water depth of 25 cm, is used for the robot to swim in. This tank size is large enough to allow room for the robot to swim without contacting the sides of the tank. IR lights are shown on the water to reflect off of the robot. For capturing data, a Basler acA2000-165umNIR IR camera, operated at 50 Hz, is used to record the robot swimming. This IR camera produces black and white images based on whether or not the object in view reflects light back to the lens. Because the robot has brightly colored foam on it, the robot reflects light back to the lens and can be tracked as it moves across the tank. To capture position data of the robot, an algorithm is run in MATLAB to track the robot as it progresses across the frames of the video. To process the data, a MATLAB script was created to take derivatives of the position data. MATLAB scripts for each dataset are provided in the appendix.

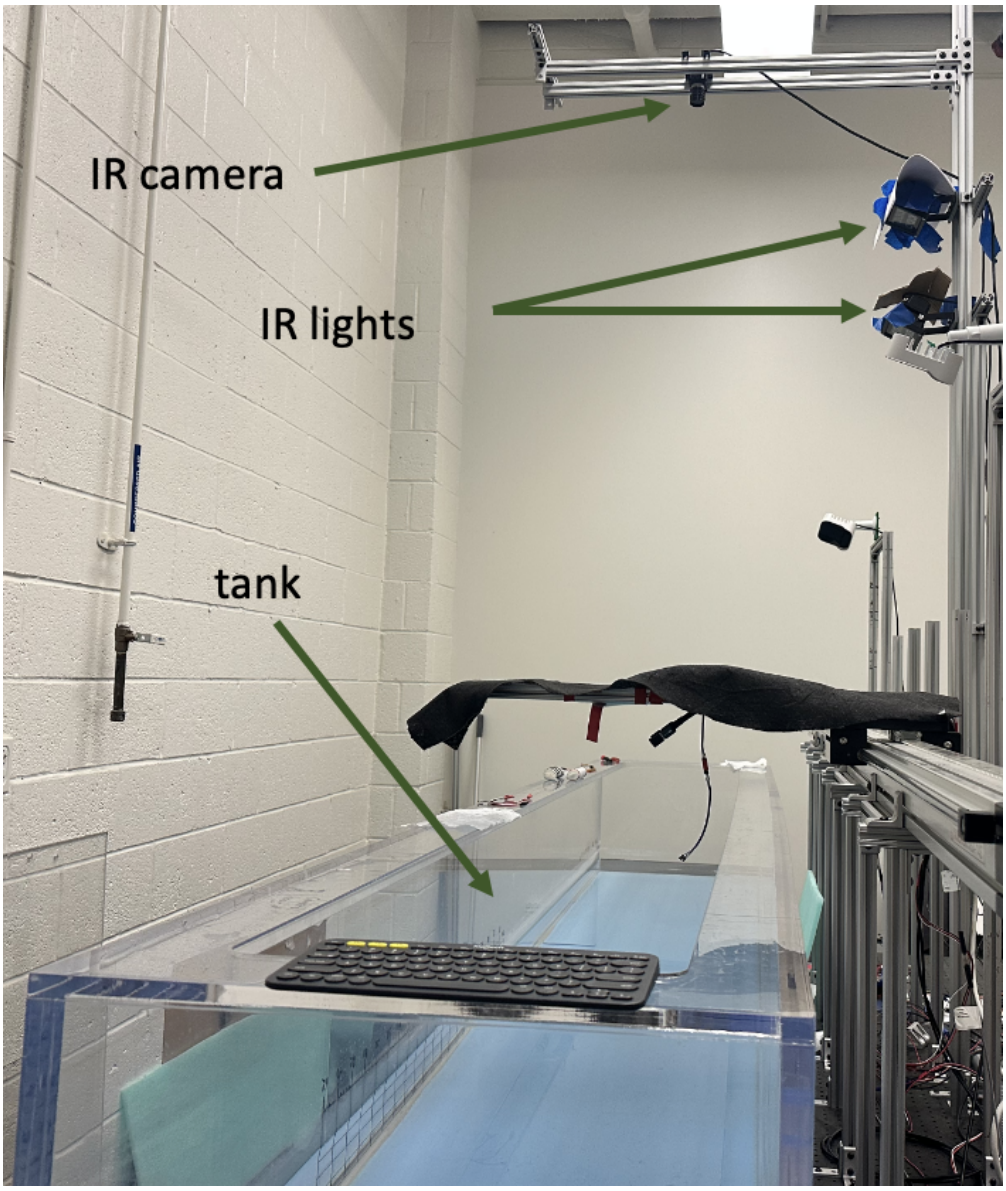

Figure 2.17: Experimental setup showing tank, IR camera, and IR lights.

Chapter 3

Results and Discussion

3.1 Speed

This section provides and discusses the results obtained during speed testing.

Position, velocity, and acceleration plots were created for each different robot based off of the averages of three trials. These plots also show plus and minus one standard deviation. Shown in Figures 3.1, 3.2, and 3.3 are the resultant graphs for the robot with no fin joint. Figures 3.4, 3.5, and 3.6 show the resultant graphs for the robot with a close fin joint. Lastly, Figures 3.7, 3.8, and 3.9 show the resultant graphs for the robot with a far fin joint. For the far fin joint, a large amount of standard deviation is seen due to experimental error in which the robot was not tracked in certain frames. These frames were excluded from the data, decreasing the size of the dataset. The average and maximum speed for each robot as well as the maximum acceleration are shown in Table 3.1. A consideration for these maximum values is that spikes in data are included as relevant. However, it is not known whether these spikes are due to noise or whether these spikes are significant. As shown by the provided data, the robot with the close fin joint is fastest, followed by the robot with no fin joint and then the robot with the far fin joint.

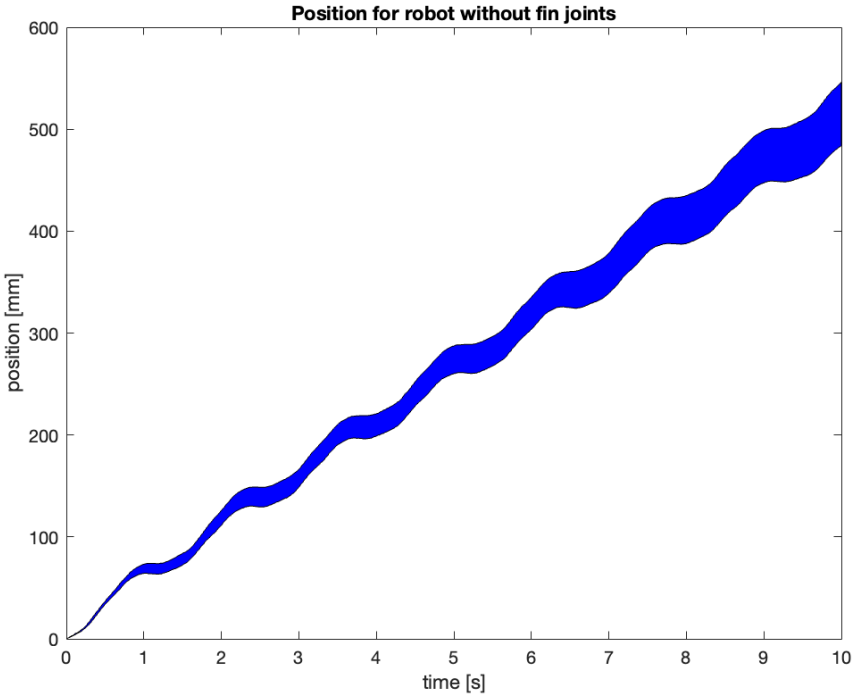

Figure 3.1: Position graph for robot with no fin joint.

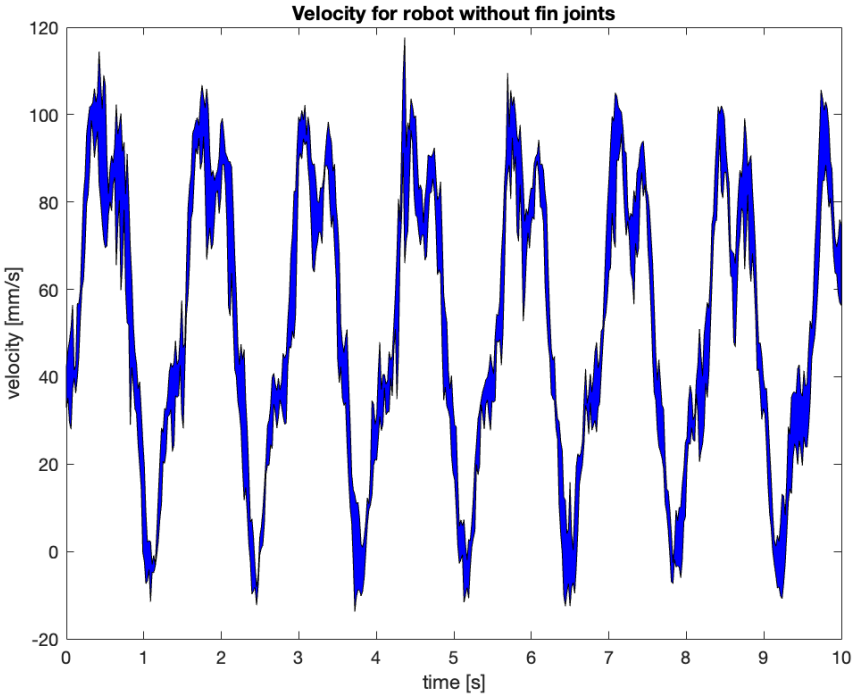

Figure 3.2: Velocity graph for robot with no fin joint.

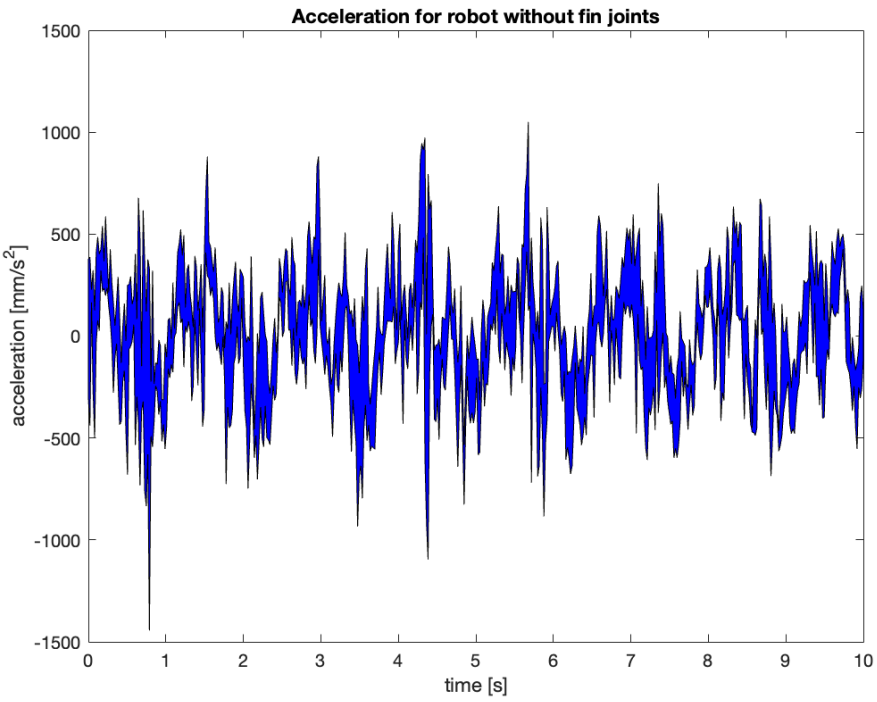

Figure 3.3: Acceleration graph for robot with no fin joint.

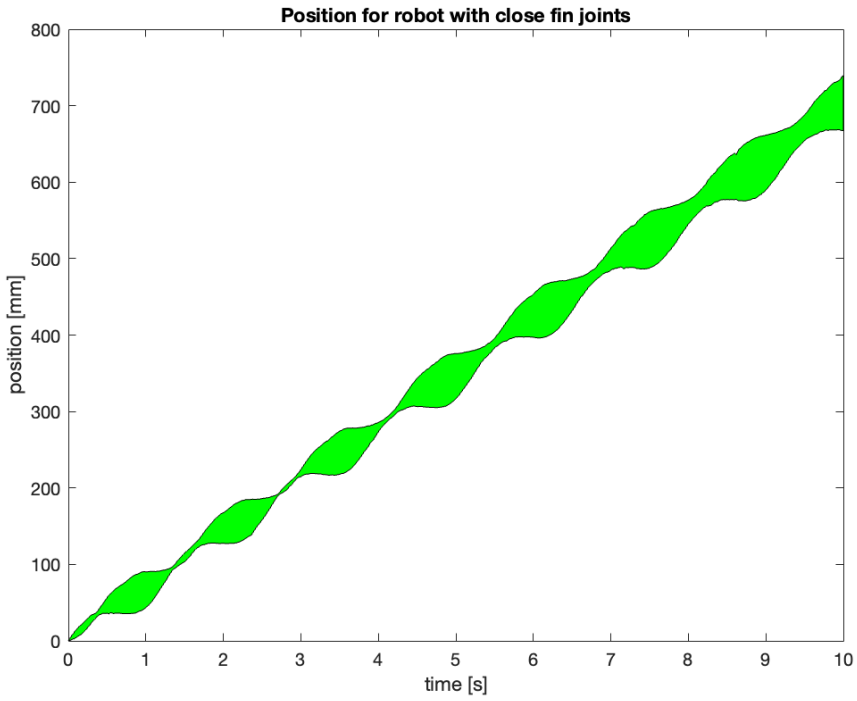

Figure 3.4: Position graph for robot with close fin joint.

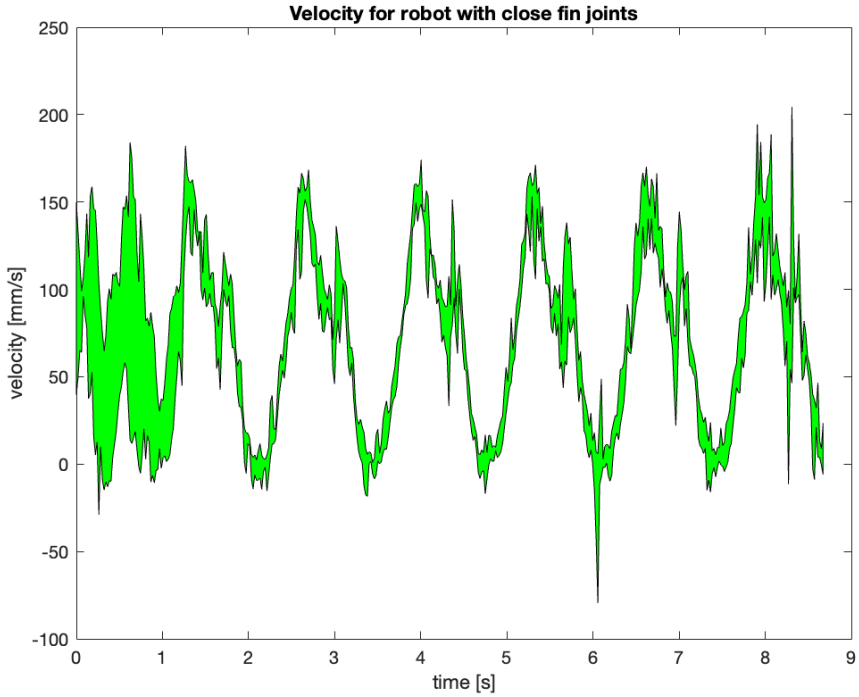

Figure 3.5: Velocity graph for robot with close fin joint.

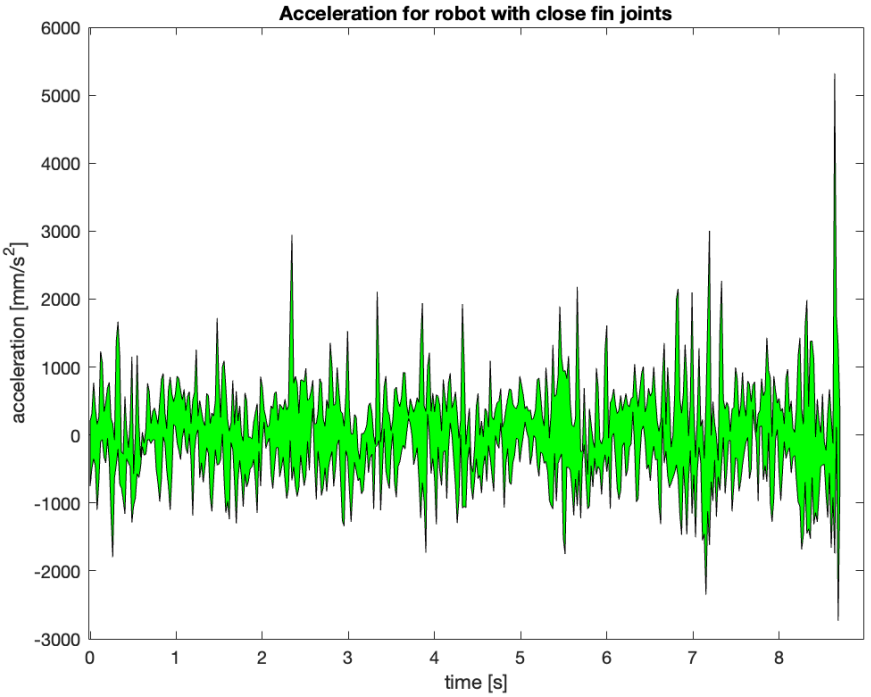

Figure 3.6: Acceleration graph for robot with close fin joint.

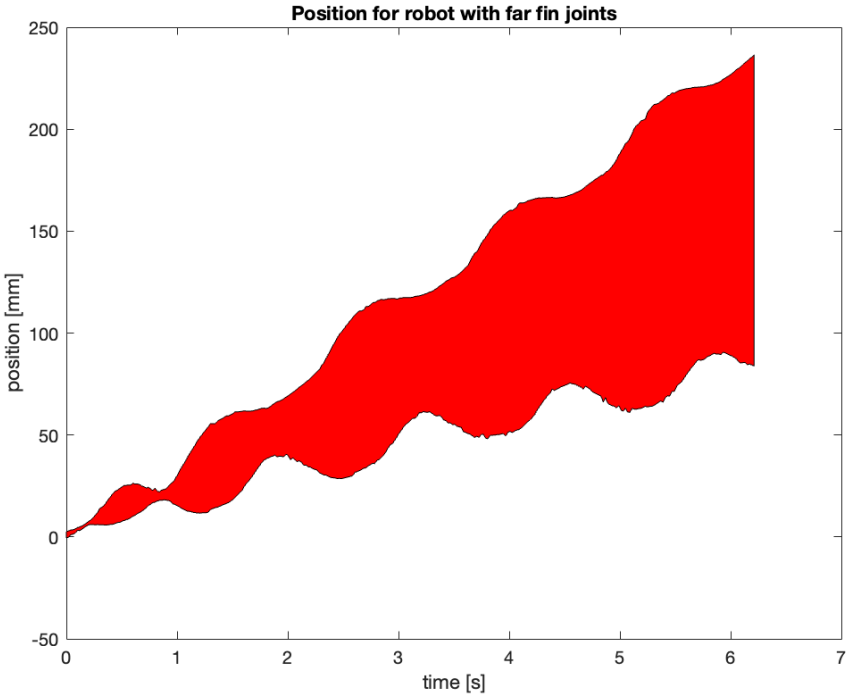

Figure 3.7: Position graph for robot with far fin joint.

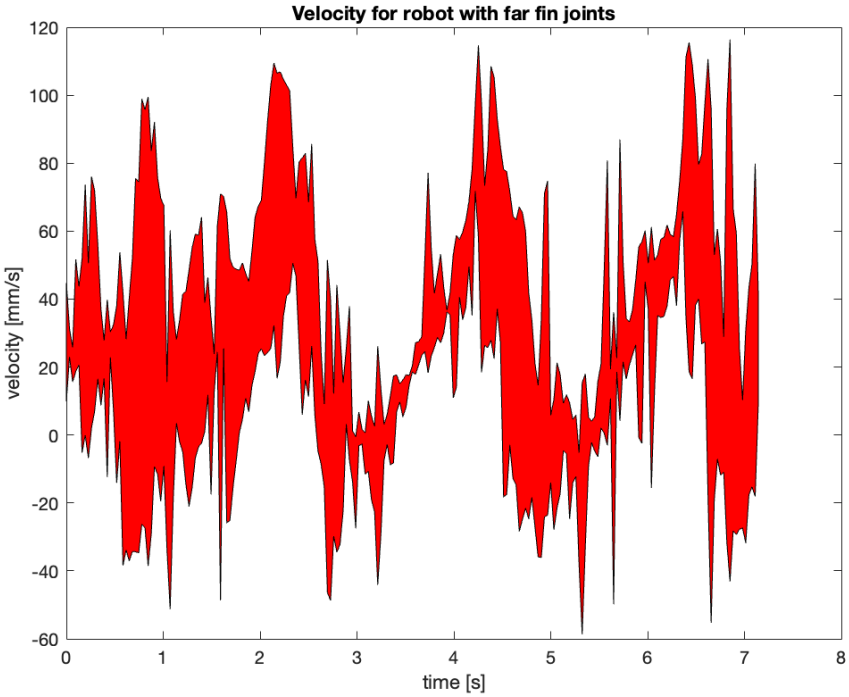

Figure 3.8: Velocity graph for robot with far fin joint.

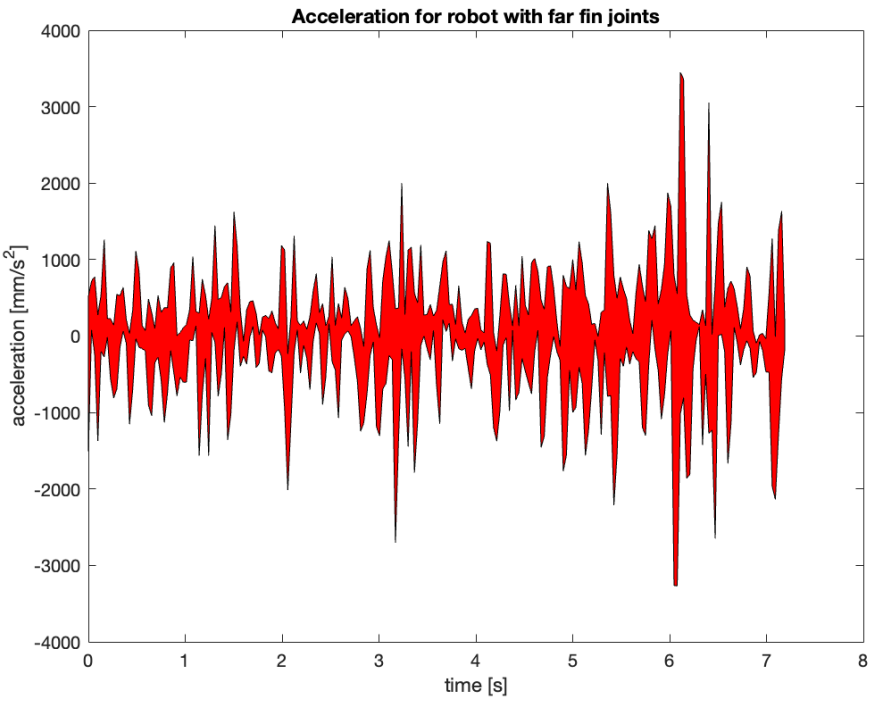

Figure 3.9: Acceleration graph for robot with far fin joint.

Robot	**Max velocity** (mm/s)	**Average velocity** (mm/s)	**Max acceleration** (mm/s^2)
No fin joint	101	52	697
Close fin joint	161	71	1790
Far fin joint	86	26	1280

Table 3.1: Table of velocity and acceleration for all robots tested.

As shown in the position graphs and velocity graphs, some amount of negative displacement and negative velocity occurs. Average acceleration centers around zero for all trials. The comparison between all these results is reasonable because the robot with the close fin joint has less negative displacement than the other robots during its return stroke because the fin bends due to the added joint. Additionally, because the robot with the far fin joint does not fold as much, it has less negative displacement when the robot resets its stroke. However, because water is able to flow through the far joint due to the hole without the joint being able to fold as much, the robot with the far joint has less forward displacement during the forward stroke. This less forward displacement

makes the robot with the far joint slower than the robot without a fin joint.

3.2 Thrust

This section provides and discusses the results of thrust for all three YoDiFIN versions.

Using the calculated acceleration data from the measured position, the resultant force is determined via Newton's Second Law. These maximum resultant forces are provided in Table 3.2. Resultant forces are a combination of two main forces acting on the robot: thrust produced by the robot and drag acting on the robot as it swims. A mass of 144 g is measured and used for calculating this resultant force. As shown in Table 3.2, the robot with a close fin joint has the highest resultant force, followed by the robot with a far fin joint and then the robot with no fin joint. These results make sense because it would be expected that the robots with holes would have less drag. Additionally, the robots that swim faster would be expected to have a higher thrust force.

Robot	**Resultant force** (mN)
No fin joint	100
Close fin joint	258
Far fin joint	184

Table 3.2: Table of resultant force for all robots tested.

3.3 Efficiency

This section discusses the efficiency improvement from the robot discussed in [9]. Because speed is greatly increased while using the same input from the same motor, the efficiency of the robot therefore increases with increasing speed. Consequently, the geometry changes to the robot in this thesis greatly improve the efficiency of the robot discussed in [9]. Efficiency is calculated based on the equation provided in Section 3.3, which compares forward displacement with backward displacement. These results are provided in Table 3.3. As can be seen in that table, the robot with the close fin joint is the most efficient, followed by the robot with no fin joint and then the

robot with the far fin joint.

Robot	**Efficiency** $(\%)$
No fin joint	99.6
Close fin joint	99.7
Far fin joint	96.4

Table 3.3: Table of efficiency for all robots tested.

Chapter 4

Conclusion

In conclusion, this thesis has presented the design, manufacture, and testing of a compliant origami swimming robot. As discussed in this thesis, adding a fin joint close to the body of the robot allows the robot to move faster with more thrust. However, using a fin joint in a non-ideal position, such as far from the fins, can cause more inefficiencies with swimming. Therefore, further research should be performed to optimize this joint placement. By addressing inefficiencies in compliant origami swimming robots, these robots become closer to being used in application areas such as search-and-rescue and underwater discovery.

Appendix

4.1 MATLAB Code

This section provides the MATLAB code used for data processing, with the joint with no fin joints processed first, followed by the robot with close fins joints and then the robot with far fin joints.

```
%% Setup
clc, clear, clf, close all;


%% Position Data filtered with smoothdata command on velocity
load 20240228_131501.mat % load dataset 1
% position
t_pos = 0:10/(length(p_world_t)-1):10; % set time stamp
pos1 = (abs(p_world_t(:,1) - max(p_world_t(:,1)))); % normalize
   position data


% velocity
tb2 = t_pos(3) - t_pos(1); % calculate the time between two
   datapoints for linear interpolation
vel1 = (pos1(3:length(pos1)) - pos1(1:(length(pos1)-2)))/tb2; %
   linear interpolation for determining velocity
vel1_fit = smoothdata(vel1, 'SmoothingFactor',0.1);
vel1 = vel1'; % change from row vector to column vector
vel1_fit = vel1_fit';
t_vel = 0:10/(length(vel1)-1):10; % calculate time vector for
   velocity
t_vel_fit = 0:10/(length(vel1_fit)-1):10; % calculate time vector
    for velocity
```

```
% acceleration
acc1 = (vel1(3:length(vel1)) - vel1(1:(length(vel1)-2)))/tb2; %
   linear interpolation for determining acceleration
acc1_fit = (vel1_fit(3:length(vel1_fit)) - vel1_fit(1:(length(
   vel1_fit)-2)))/tb2; % linear interpolation for determining
   acceleration
t_acc = 0:10/(length(acc1)-1):10; % calculate time vector for
   acceleration
t_acc_fit = 0:10/(length(acc1_fit)-1):10; % calculate time vector
    for acceleration

% plot data
figure(1)
subplot (3,1,1)
plot(t_pos, pos1)
legend('unfiltered')
title('X-position versus time')
xlabel('time [s]')
ylabel('position [mm]')

subplot (3,1,2)
plot(t_vel,vel1, t_vel_fit, vel1_fit)
legend('unfiltered', 'smoothed')
title('X-velocity versus time')
xlabel('time [s]')
ylabel('velocity [mm/s]')
```

```
subplot(3,1,3)
plot(t_acc,acc1, t_acc_fit, acc1_fit)
legend('unfiltered', 'smoothed')
title('X-acceleration versus time')
xlabel('time [s]')
ylabel('acceleration [mm/s^2]')


%% Position Data filtered with smoothdata command on velocity
load 20240228_131832.mat % load dataset 2
% position
t_pos = 0:10/(length(p_world_t)-1):10; % set time stamp
pos2 = (abs(p_world_t(:,1) - max(p_world_t(:,1)))); % normalize
   position data

% velocity
tb2 = t_pos(3) - t_pos(1); % calculate the time between two
   datapoints for linear interpolation
vel2 = (pos2(3:length(pos2)) - pos2(1:(length(pos2)-2)))/tb2; %
   linear interpolation for determining velocity
vel2_fit = smoothdata(vel2, 'SmoothingFactor',0.1);
vel2 = vel2'; % change from row vector to column vector
vel2_fit = vel2_fit';
t_vel = 0:10/(length(vel2)-1):10; % calculate time vector for
   velocity
```

```
t_vel_fit = 0:10/(length(vel2_fit)-1):10; % calculate time vector
    for velocity

% acceleration
acc2 = (vel2(3:length(vel2)) - vel2(1:(length(vel2)-2)))/tb2; %
   linear interpolation for determining acceleration
acc2_fit = (vel2_fit(3:length(vel2_fit)) - vel2_fit(1:(length(
   vel2_fit)-2)))/tb2; % linear interpolation for determining
   acceleration
t_acc = 0:10/(length(acc2)-1):10; % calculate time vector for
   acceleration
t_acc_fit = 0:10/(length(acc2_fit)-1):10; % calculate time vector
    for acceleration

% plot data
figure(2)
subplot (3,1,1)
plot(t_pos, pos2)
legend('unfiltered')
title('X-position versus time')
xlabel('time [s]')
ylabel('position [mm]')

subplot (3,1,2)
plot(t_vel,vel2, t_vel_fit, vel2_fit)
legend('unfiltered', 'smoothed')
title('X-velocity versus time')
```

```
xlabel('time [s]')
ylabel('velocity [mm/s]')

subplot(3,1,3)
plot(t_acc,acc2, t_acc_fit, acc2_fit)
legend('unfiltered', 'smoothed')
title('X-acceleration versus time')
xlabel('time [s]')
ylabel('acceleration [mm/s^2]')

%% Position Data filtered with smoothdata command on velocity
load 20240228_132014.mat % load dataset 3
% position
t_pos = 0:10/(length(p_world_t)-1):10; % set time stamp
p_world_t(33) = (p_world_t(32) + p_world_t(34))/2; %% line for
   jumped data on dataset 3
pos3 = (abs(p_world_t(:,1) - max(p_world_t(:,1)))); % normalize
   position data

% velocity
tb2 = t_pos(3) - t_pos(1); % calculate the time between two
   datapoints for linear interpolation
vel3 = (pos3(3:length(pos3)) - pos3(1:(length(pos3)-2)))/tb2; %
   linear interpolation for determining velocity
vel3_fit = smoothdata(vel3, 'SmoothingFactor',0.1);
vel3 = vel3'; % change from row vector to column vector
vel3_fit = vel3_fit';
```

```
t_vel = 0:10/(length(vel3)-1):10; % calculate time vector for
   velocity
t_vel_fit = 0:10/(length(vel3_fit)-1):10; % calculate time vector
    for velocity

% acceleration
acc3 = (vel3(3:length(vel3)) - vel3(1:(length(vel3)-2)))/tb2; %
   linear interpolation for determining acceleration
acc3_fit = (vel3_fit(3:length(vel3_fit)) - vel3_fit(1:(length(
   vel3_fit)-2)))/tb2; % linear interpolation for determining
   acceleration
t_acc = 0:10/(length(acc3)-1):10; % calculate time vector for
   acceleration
t_acc_fit = 0:10/(length(acc3_fit)-1):10; % calculate time vector
    for acceleration

% plot data
figure(3)
subplot (3,1,1)
plot(t_pos, pos3)
legend('unfiltered')
title('X-position versus time')
xlabel('time [s]')
ylabel('position [mm]')

subplot (3,1,2)
plot(t_vel,vel3, t_vel_fit, vel3_fit)
```

```
legend('unfiltered', 'smoothed')
title('X-velocity versus time')
xlabel('time [s]')
ylabel('velocity [mm/s]')

subplot(3,1,3)
plot(t_acc,acc3, t_acc_fit, acc3_fit)
legend('unfiltered', 'smoothed')
title('X-acceleration versus time')
xlabel('time [s]')
ylabel('acceleration [mm/s^2]')

%% Plot averages
% plot data
figure(4)
subplot (3,1,1)
plot(t_pos, (pos1 + pos2 + pos3)/3)
legend('unfiltered')
title('X-position versus time')
xlabel('time [s]')
ylabel('position [mm]')

subplot (3,1,2)
plot(t_vel, (vel1 + vel2 + vel3)/3, t_vel_fit, (vel1_fit +
   vel2_fit + vel3_fit)/3)
legend('unfiltered', 'smoothed')
title('X-velocity versus time')
```

```
xlabel('time [s]')
ylabel('velocity [mm/s]')

subplot(3,1,3)
plot(t_acc, (acc1 + acc2 + acc3)/3, t_acc_fit, (acc1_fit +
   acc2_fit + acc3_fit)/3)
legend('unfiltered', 'smoothed')
title('X-acceleration versus time')
xlabel('time [s]')
ylabel('acceleration [mm/s^2]')

posmax = max((pos1 + pos2 + pos3)/3)
velmax = max((vel1 + vel2 + vel3)/3)
velavg = mean((vel1 + vel2 + vel3)/3)
accmax = max((acc1 + acc2 + acc3)/3)



%%
% position
posvals = [pos1 pos2 pos3];
for i = 1:length(pos1)
    stdpos(i) = std(posvals(i, :));
end
stdpos = stdpos';
posavg = (pos1 + pos2 + pos3)/3;
stdhpos = posavg + stdpos;
stdlpos = posavg - stdpos;
```

```
figure(5)
plot(t_pos, posavg, t_pos, stdhpos, t_pos, stdlpos)
patch([t_pos fliplr(t_pos)], [stdlpos' fliplr(stdhpos')], 'b')
title('Position for robot without fin joints')
xlabel('time [s]')
ylabel('position [mm]')

% velocity
velvals = [vel1' vel2' vel3'];
for i = 1:length(vel1)
    stdvel(i) = std(velvals(i, :));
end
stdvel = stdvel';
velavg = ((vel1 + vel2 + vel3)/3)';
stdhvel = velavg + stdvel;
stdlvel = velavg - stdvel;
figure(6)
plot(t_vel, velavg, t_vel, stdhvel, t_vel, stdlvel)
patch([t_vel fliplr(t_vel)], [stdlvel' fliplr(stdhvel')], 'b')
title('Velocity for robot without fin joints')
xlabel('time [s]')
ylabel('velocity [mm/s]')

% acceleration
accvals = [acc1' acc2' acc3'];
for i = 1:length(acc1)
    stdacc(i) = std(accvals(i, :));
```

```
end
stdacc = stdacc';
accavg = ((acc1 + acc2 + acc3)/3)';
stdhacc = accavg + stdacc;
stdlacc = accavg - stdacc;
figure(7)
plot(t_acc, accavg, t_acc, stdhacc, t_acc, stdlacc)
patch([t_acc fliplr(t_acc)], [stdlacc' fliplr(stdhacc')], 'b')
title('Acceleration for robot without fin joints')
xlabel('time [s]')
ylabel('acceleration [mm/s^2]')

%% Setup
clc, clear, clf, close all;

%% Position Data filtered with smoothdata command on velocity
load 20240228_145240.mat % load dataset 1
% position
t_pos = 0:10/(length(p_world_t)-1):10; % set time stamp
p_world_t(469) = (p_world_t(468) + p_world_t(470))/2; %% line for
    jumped data
p_world_t(479) = (p_world_t(478) + p_world_t(480))/2; %% line for
    jumped data
p_world_t(481) = (p_world_t(480) + p_world_t(482))/2; %% line for
    jumped data
pos1 = p_world_t(:,1) - min(p_world_t(:,1)); % normalize position
    data
```

```
% velocity
tb2 = t_pos(3) - t_pos(1); % calculate the time between two
   datapoints for linear interpolation
vel1 = (pos1(3:length(pos1)) - pos1(1:(length(pos1)-2)))/tb2; %
   linear interpolation for determining velocity
vel1_fit = smoothdata(vel1, 'SmoothingFactor',0.1);
vel1 = vel1'; % change from row vector to column vector
vel1_fit = vel1_fit';
t_vel = 0:10/(length(vel1)-1):10; % calculate time vector for
   velocity
t_vel_fit = 0:10/(length(vel1_fit)-1):10; % calculate time vector
    for velocity

% acceleration
acc1 = (vel1(3:length(vel1)) - vel1(1:(length(vel1)-2)))/tb2; %
   linear interpolation for determining acceleration
acc1_fit = (vel1_fit(3:length(vel1_fit)) - vel1_fit(1:(length(
   vel1_fit)-2)))/tb2; % linear interpolation for determining
   acceleration
t_acc = 0:10/(length(acc1)-1):10; % calculate time vector for
   acceleration
t_acc_fit = 0:10/(length(acc1_fit)-1):10; % calculate time vector
    for acceleration

% plot data
figure(1)
```

```
subplot (3,1,1)
plot(t_pos, pos1)
legend('unfiltered')
title('X-position versus time')
xlabel('time [s]')
ylabel('position [mm]')

subplot (3,1,2)
plot(t_vel,vel1, t_vel_fit, vel1_fit)
legend('unfiltered', 'smoothed')
title('X-velocity versus time')
xlabel('time [s]')
ylabel('velocity [mm/s]')

subplot(3,1,3)
plot(t_acc,acc1, t_acc_fit, acc1_fit)
legend('unfiltered', 'smoothed')
title('X-acceleration versus time')
xlabel('time [s]')
ylabel('acceleration [mm/s^2]')


%% Position Data filtered with smoothdata command on velocity
load 20240228_145408.mat % load dataset 2
% position
t_pos = 0:10/(length(p_world_t)-1):10; % set time stamp
```

```
p_world_t(489) = (p_world_t(487) + p_world_t(490))/2; %% line for
    jumped data on dataset 3
pos2 = p_world_t(:,1) - min(p_world_t(:,1)); % normalize position
    data

% velocity
tb2 = t_pos(3) - t_pos(1); % calculate the time between two
   datapoints for linear interpolation
vel2 = (pos2(3:length(pos2)) - pos2(1:(length(pos2)-2)))/tb2; %
   linear interpolation for determining velocity
vel2_fit = smoothdata(vel2, 'SmoothingFactor',0.1);
vel2 = vel2'; % change from row vector to column vector
vel2_fit = vel2_fit';
t_vel = 0:10/(length(vel2)-1):10; % calculate time vector for
   velocity
t_vel_fit = 0:10/(length(vel2_fit)-1):10; % calculate time vector
    for velocity

% acceleration
acc2 = (vel2(3:length(vel2)) - vel2(1:(length(vel2)-2)))/tb2; %
   linear interpolation for determining acceleration
acc2_fit = (vel2_fit(3:length(vel2_fit)) - vel2_fit(1:(length(
   vel2_fit)-2)))/tb2; % linear interpolation for determining
   acceleration
t_acc = 0:10/(length(acc2)-1):10; % calculate time vector for
   acceleration
```

```
t_acc_fit = 0:10/(length(acc2_fit)-1):10; % calculate time vector
    for acceleration

% plot data
figure(2)
subplot (3,1,1)
plot(t_pos, pos2)
legend('unfiltered')
title('X-position versus time')
xlabel('time [s]')
ylabel('position [mm]')

subplot (3,1,2)
plot(t_vel,vel2, t_vel_fit, vel2_fit)
legend('unfiltered', 'smoothed')
title('X-velocity versus time')
xlabel('time [s]')
ylabel('velocity [mm/s]')

subplot(3,1,3)
plot(t_acc,acc2, t_acc_fit, acc2_fit)
legend('unfiltered', 'smoothed')
title('X-acceleration versus time')
xlabel('time [s]')
ylabel('acceleration [mm/s^2]')

%% Position Data filtered with smoothdata command on velocity
```

```
load 20240228_145557.mat % load dataset 3
% position
t_pos = 0:10/(length(p_world_t)-1):10; % set time stamp
pos3 = p_world_t(:,1) - min(p_world_t(:,1)); % normalize position
    data

% velocity
tb2 = t_pos(3) - t_pos(1); % calculate the time between two
   datapoints for linear interpolation
vel3 = (pos3(3:length(pos3)) - pos3(1:(length(pos3)-2)))/tb2; %
   linear interpolation for determining velocity
vel3_fit = smoothdata(vel3, 'SmoothingFactor',0.1);
vel3 = vel3'; % change from row vector to column vector
vel3_fit = vel3_fit';
t_vel = 0:10/(length(vel3)-1):10; % calculate time vector for
   velocity
t_vel_fit = 0:10/(length(vel3_fit)-1):10; % calculate time vector
    for velocity

% acceleration
acc3 = (vel3(3:length(vel3)) - vel3(1:(length(vel3)-2)))/tb2; %
   linear interpolation for determining acceleration
acc3_fit = (vel3_fit(3:length(vel3_fit)) - vel3_fit(1:(length(
   vel3_fit)-2)))/tb2; % linear interpolation for determining
   acceleration
t_acc = 0:10/(length(acc3)-1):10; % calculate time vector for
   acceleration
```

```
t_acc_fit = 0:10/(length(acc3_fit)-1):10; % calculate time vector
    for acceleration

% plot data
figure(3)
subplot (3,1,1)
plot(t_pos, pos3)
legend('unfiltered')
title('X-position versus time')
xlabel('time [s]')
ylabel('position [mm]')

subplot (3,1,2)
plot(t_vel,vel3, t_vel_fit, vel3_fit)
legend('unfiltered', 'smoothed')
title('X-velocity versus time')
xlabel('time [s]')
ylabel('velocity [mm/s]')

subplot(3,1,3)
plot(t_acc,acc3, t_acc_fit, acc3_fit)
legend('unfiltered', 'smoothed')
title('X-acceleration versus time')
xlabel('time [s]')
ylabel('acceleration [mm/s^2]')

%% Plot averages
```

```
% plot data
n1 = 66;
s1 = n1-1;
for k1 = 1:s1
    vel1(k1) = [];
end

n2 = 18;
s2 = n2-1;
for k2 = 1:s2
    vel2(k2) = [];
end

n3 = 56;
s3 = n3-1;
for k3 = 1:s3
    vel3(k3) = [];
end

lengthvec = [length(vel1) length(vel2) length(vel3)];
lastval = min(lengthvec);
if length(vel1) >= lastval
    vel1(lastval:end) = [];
    acc1(lastval:end) = [];
end

if length(vel2) > lastval
```

```
    vel2(lastval:end) = [];
    acc2(lastval:end) = [];
end

if length(vel3) > lastval
    vel3(lastval:end) = [];
    acc3(lastval:end) = [];
end
t_vel(lastval:end) = [];
t_acc(lastval:end) = [];

figure(4)
subplot (3,1,1)
plot(t_pos, (pos1 + pos2 + pos3)/3)
legend('unfiltered')
title('X-position versus time')
xlabel('time [s]')
ylabel('position [mm]')

subplot (3,1,2)
plot(t_vel, (vel1 + vel2 + vel3)/3)
legend('unfiltered')
title('X-velocity versus time')
xlabel('time [s]')
ylabel('velocity [mm/s]')

subplot(3,1,3)
```

```
plot(t_acc, (acc1 + acc2 + acc3)/3)
legend('unfiltered')
title('X-acceleration versus time')
xlabel('time [s]')
ylabel('acceleration [mm/s^2]')

posmax = max((pos1 + pos2 + pos3)/3)
velmax = max((vel1 + vel2 + vel3)/3)
velavg = mean((vel1 + vel2 + vel3)/3)
accmax = max((acc1 + acc2 + acc3)/3)



%%
% position
posvals = [pos1 pos2 pos3];
for i = 1:length(pos1)
    stdpos(i) = std(posvals(i, :));
end
stdpos = stdpos';
posavg = (pos1 + pos2 + pos3)/3;
stdhpos = posavg + stdpos;
stdlpos = posavg - stdpos;
figure(8)
plot(t_pos, posavg, t_pos, stdhpos, t_pos, stdlpos)
patch([t_pos fliplr(t_pos)], [stdlpos' fliplr(stdhpos')], 'g')
title('Position for robot with close fin joints')
xlabel('time [s]')
```

```
ylabel('position [mm]')

% velocity
velvals = [vel1' vel2' vel3'];
for i = 1:length(vel1)
    stdvel(i) = std(velvals(i, :));
end
stdvel = stdvel';
velavg = ((vel1 + vel2 + vel3)/3)';
stdhvel = velavg + stdvel;
stdlvel = velavg - stdvel;
figure(9)
plot(t_vel, velavg, t_vel, stdhvel, t_vel, stdlvel)
patch([t_vel fliplr(t_vel)], [stdlvel' fliplr(stdhvel')], 'g')
title('Velocity for robot with close fin joints')
xlabel('time [s]')
ylabel('velocity [mm/s]')

% acceleration
accvals = [acc1' acc2' acc3'];
for i = 1:length(acc1)
    stdacc(i) = std(accvals(i, :));
end
stdacc = stdacc';
accavg = ((acc1 + acc2 + acc3)/3)';
stdhacc = accavg + stdacc;
stdlacc = accavg - stdacc;
```

```
figure(10)
plot(t_acc, accavg, t_acc, stdhacc, t_acc, stdlacc)
patch([t_acc fliplr(t_acc)], [stdlacc' fliplr(stdhacc')], 'g')
title('Acceleration for robot with close fin joints')
xlabel('time [s]')
ylabel('acceleration [mm/s^2]')

%% Setup
clc, clear, clf, close all;

%% Position Data filtered with smoothdata command on velocity
load 20240228_150457.mat % load dataset 1
% position
t_pos = 0:10/(length(p_world_t)-1):10; % set time stamp
p_world_t(67) = (p_world_t(68) + p_world_t(66))/2; %% line for
   jumped data
p_world_t(151:157) = (p_world_t(150) + p_world_t(158))/2; %% line
    for jumped data
p_world_t(261) = (p_world_t(260) + p_world_t(262))/2; %% line for
    jumped data
p_world_t(400:402) = (p_world_t(399) + p_world_t(403))/2; %% line
    for jumped data
p_world_t(405:406) = (p_world_t(404) + p_world_t(407))/2; %% line
    for jumped data
p_world_t(201:202) = (p_world_t(200) + p_world_t(203))/2; %% line
    for jumped data
```

```
p_world_t(269) = (p_world_t(268) + p_world_t(270))/2; %% line for
    jumped data
p_world_t(409:410) = (p_world_t(408) + p_world_t(411))/2; %% line
    for jumped data
pos1 = abs(p_world_t(:,1) - min(p_world_t(:,1))); % normalize
   position data

% velocity
tb2 = t_pos(3) - t_pos(1); % calculate the time between two
   datapoints for linear interpolation
vel1 = (pos1(3:length(pos1)) - pos1(1:(length(pos1)-2)))/tb2; %
   linear interpolation for determining velocity
vel1_fit = smoothdata(vel1, 'SmoothingFactor',0.1);
vel1 = vel1'; % change from row vector to column vector
vel1_fit = vel1_fit';
t_vel = 0:10/(length(vel1)-1):10; % calculate time vector for
   velocity
t_vel_fit = 0:10/(length(vel1_fit)-1):10; % calculate time vector
    for velocity

% acceleration
acc1 = (vel1(3:length(vel1)) - vel1(1:(length(vel1)-2)))/tb2; %
   linear interpolation for determining acceleration
acc1_fit = (vel1_fit(3:length(vel1_fit)) - vel1_fit(1:(length(
   vel1_fit)-2)))/tb2; % linear interpolation for determining
   acceleration
```

```
t_acc = 0:10/(length(acc1)-1):10; % calculate time vector for
   acceleration
t_acc_fit = 0:10/(length(acc1_fit)-1):10; % calculate time vector
    for acceleration

% plot data
figure(1)
subplot (3,1,1)
plot(t_pos, pos1)
legend('unfiltered')
title('X-position versus time')
xlabel('time [s]')
ylabel('position [mm]')

subplot (3,1,2)
plot(t_vel,vel1, t_vel_fit, vel1_fit)
legend('unfiltered', 'smoothed')
title('X-velocity versus time')
xlabel('time [s]')
ylabel('velocity [mm/s]')

subplot(3,1,3)
plot(t_acc,acc1, t_acc_fit, acc1_fit)
legend('unfiltered', 'smoothed')
title('X-acceleration versus time')
xlabel('time [s]')
ylabel('acceleration [mm/s^2]')
```

```
%% Position Data filtered with smoothdata command on velocity
load 20240228_143744.mat % load dataset 2
% position
t_pos = 0:10/(length(p_world_t)-1):10; % set time stamp
%p_world_t(489) = (p_world_t(487) + p_world_t(490))/2; %% line
   for jumped data on dataset 3
pos2 = p_world_t(:,1) - min(p_world_t(:,1)); % normalize position
    data

% velocity
tb2 = t_pos(3) - t_pos(1); % calculate the time between two
   datapoints for linear interpolation
vel2 = (pos2(3:length(pos2)) - pos2(1:(length(pos2)-2)))/tb2; %
   linear interpolation for determining velocity
vel2_fit = smoothdata(vel2, 'SmoothingFactor',0.1);
vel2 = vel2'; % change from row vector to column vector
vel2_fit = vel2_fit';
t_vel = 0:10/(length(vel2)-1):10; % calculate time vector for
   velocity
t_vel_fit = 0:10/(length(vel2_fit)-1):10; % calculate time vector
    for velocity

% % smooth velocity %
% vel2 = vel2(find(vel2 < 100));
% tnvel2 = 0:10/(length(vel2)-1):10;
```

```
% % %

% acceleration
acc2 = (vel2(3:length(vel2)) - vel2(1:(length(vel2)-2)))/tb2; %
   linear interpolation for determining acceleration
acc2_fit = (vel2_fit(3:length(vel2_fit)) - vel2_fit(1:(length(
   vel2_fit)-2)))/tb2; % linear interpolation for determining
   acceleration
t_acc = 0:10/(length(acc2)-1):10; % calculate time vector for
   acceleration
t_acc_fit = 0:10/(length(acc2_fit)-1):10; % calculate time vector
    for acceleration

% plot data
figure(2)
subplot (3,1,1)
plot(t_pos, pos2)
legend('unfiltered')
title('X-position versus time')
xlabel('time [s]')
ylabel('position [mm]')

subplot (3,1,2)
%plot(tnvel2,vel2)
plot(t_vel_fit, vel2_fit)
%plot(t_vel,vel2, t_vel_fit, vel2_fit)
legend('unfiltered', 'smoothed')
```

```
title('X-velocity versus time')
xlabel('time [s]')
ylabel('velocity [mm/s]')

subplot(3,1,3)
plot(t_acc_fit, acc2_fit)
%plot(t_acc,acc2, t_acc_fit, acc2_fit)
legend('unfiltered', 'smoothed')
title('X-acceleration versus time')
xlabel('time [s]')
ylabel('acceleration [mm/s^2]')

%% Position Data filtered with smoothdata command on velocity
load 20240228_143904.mat % load dataset 3
% position
t_pos = 0:10/(length(p_world_t)-1):10; % set time stamp
t_pos((500 - 148 - 40):end) = [];
pos3 = p_world_t(:,1) - min(p_world_t(:,1)); % normalize position
    data
pos3(1:148+41) = [];
for l = 2:length(pos3)
    if pos3(l) == NaN;
        pos3(l) = (pos3(l+1) + pos3(l-1))/2;
    end
end

% velocity
```

```
tb2 = t_pos(3) - t_pos(1); % calculate the time between two
   datapoints for linear interpolation
vel3 = (pos3(3:length(pos3)) - pos3(1:(length(pos3)-2)))/tb2; %
   linear interpolation for determining velocity
vel3_fit = smoothdata(vel3, 'SmoothingFactor',0.1);
vel3 = vel3'; % change from row vector to column vector
vel3_fit = vel3_fit';
t_vel = 0:10/(length(vel3)-1):10; % calculate time vector for
   velocity
t_vel_fit = 0:10/(length(vel3_fit)-1):10; % calculate time vector
    for velocity

% acceleration
acc3 = (vel3(3:length(vel3)) - vel3(1:(length(vel3)-2)))/tb2; %
   linear interpolation for determining acceleration
acc3_fit = (vel3_fit(3:length(vel3_fit)) - vel3_fit(1:(length(
   vel3_fit)-2)))/tb2; % linear interpolation for determining
   acceleration
t_acc = 0:10/(length(acc3)-1):10; % calculate time vector for
   acceleration
t_acc_fit = 0:10/(length(acc3_fit)-1):10; % calculate time vector
    for acceleration

% plot data
figure(3)
subplot (3,1,1)
plot(t_pos, pos3)
```

```
legend('unfiltered')
title('X-position versus time')
xlabel('time [s]')
ylabel('position [mm]')

subplot (3,1,2)
plot(t_vel,vel3) %, t_vel_fit, vel3_fit)
legend('unfiltered', 'smoothed')
title('X-velocity versus time')
xlabel('time [s]')
ylabel('velocity [mm/s]')

subplot(3,1,3)
plot(t_acc,acc3, t_acc_fit, acc3_fit)
legend('unfiltered', 'smoothed')
title('X-acceleration versus time')
xlabel('time [s]')
ylabel('acceleration [mm/s^2]')

%% Plot averages
% plot data
n1 = 51;
s1 = n1-1;
for k1 = 1:s1
    vel1(k1) = [];
end
```

```
n2 = 15;
s2 = n2-1;
for k2 = 1:s2
    vel2_fit(k2) = [];
end

n3 = 88;
s3 = n3-1;
for k3 = 1:s3
    vel3(k3) = [];
end

lengthvec = [length(vel1) length(vel2_fit) length(vel3)];
lastval = min(lengthvec);
if length(vel1) >= lastval
    vel1(lastval:end) = [];
    acc1(lastval:end) = [];
end

if length(vel2_fit) > lastval
    vel2_fit(lastval:end) = [];
    acc2(lastval:end) = [];
end
%valu = lastval-1;
if length(vel3) >= lastval
    vel3(lastval:end) = [];
    acc3(lastval:end) = [];
```

```
end
t_vel(lastval:end) = [];
t_acc(lastval:end) = [];

pos1(length(pos3)+1:end) = [];
pos2(length(pos3)+1:end) = [];
% t_posn = 0:10/(length(pos3)-1):10;
figure(4)
subplot (3,1,1)
plot(t_pos, (pos1 + pos2 + pos3)/3)
legend('unfiltered')
title('X-position versus time')
xlabel('time [s]')
ylabel('position [mm]')

subplot (3,1,2)
plot(t_vel, (vel1 + vel2_fit + vel3)/3)
legend('unfiltered')
title('X-velocity versus time')
xlabel('time [s]')
ylabel('velocity [mm/s]')

subplot(3,1,3)
plot(t_acc, (acc1 + acc2 + acc3)/3)
legend('unfiltered')
title('X-acceleration versus time')
xlabel('time [s]')
```

```
ylabel('acceleration [mm/s^2]')

posmax = max((pos1 + pos2 + pos3)/3)
velmax = max((vel1 + vel2_fit + vel3)/3)
velavg = mean((vel1 + vel2_fit + vel3)/3)
accmax = max((acc1 + acc2 + acc3)/3)



%%
% position
posvals = [pos1 pos2 pos3];
for i = 1:length(pos1)
    stdpos(i) = std(posvals(i, :));
end
stdpos = stdpos';
posavg = (pos1 + pos2 + pos3)/3;
stdhpos = posavg + stdpos;
stdlpos = posavg - stdpos;
figure(5)
plot(t_pos, posavg, t_pos, stdhpos, t_pos, stdlpos)
patch([t_pos fliplr(t_pos)], [stdlpos' fliplr(stdhpos')], 'r')
title('Position for robot with close fin joints')
xlabel('time [s]')
ylabel('position [mm]')

% velocity
velvals = [vel1' vel2_fit' vel3'];
```

```
for i = 1:length(vel1)
    stdvel(i) = std(velvals(i, :));
end
stdvel = stdvel';
velavg = ((vel1 + vel2_fit + vel3)/3)';
stdhvel = velavg + stdvel;
stdlvel = velavg - stdvel;
figure(6)
plot(t_vel, velavg, t_vel, stdhvel, t_vel, stdlvel)
patch([t_vel fliplr(t_vel)], [stdlvel' fliplr(stdhvel')], 'r')
title('Velocity for robot with far fin joints')
xlabel('time [s]')
ylabel('velocity [mm/s]')

% acceleration
accvals = [acc1' acc2' acc3'];
for i = 1:length(acc1)
    stdacc(i) = std(accvals(i, :));
end
stdacc = stdacc';
accavg = ((acc1 + acc2 + acc3)/3)';
stdhacc = accavg + stdacc;
stdlacc = accavg - stdacc;
figure(7)
plot(t_acc, accavg, t_acc, stdhacc, t_acc, stdlacc)
patch([t_acc fliplr(t_acc)], [stdlacc' fliplr(stdhacc')], 'r')
title('Acceleration for robot with far fin joints')
```

```
xlabel('time [s]')
ylabel('acceleration [mm/s^2]')
```

Bibliography

[1] RJ Lang. Origami design secrets: Mathematical methods for an ancient art vol. 27. *Wellesley, MA: Springer: Springer*, 2005.

[2] Zhiyuan Yang, Dongsheng Chen, David J. Levine, and Cynthia Sung. Origami-inspired robot that swims via jet propulsion. *IEEE Robotics and Automation Letters*, 6(4):7145–7152, 2021.

[3] Dongting Li, Sichuan Huang, Yong Tang, Hamidreza Marvi, Junliang Tao, and Daniel M. Aukes. Compliant fins for locomotion in granular media. *IEEE Robotics and Automation Letters*, 6(3):5984–5991, 2021.

[4] Jared Butler, Landen Bowen, Eric Wilcox, Adam Shrager, Mary I Frecker, Paris von Lockette, Timothy W Simpson, Robert J Lang, Larry L Howell, and Spencer P Magleby. A model for multi-input mechanical advantage in origami-based mechanisms. *Journal of Mechanisms and Robotics*, 10(6):061007, 2018.

[5] Landen A Bowen, Clayton L Grames, Spencer P Magleby, Robert J Lang, and Larry L Howell. An approach for understanding action origami as kinematic mechanisms. In *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, volume 55942, page V06BT07A044. American Society of Mechanical Engineers, 2013.

[6] Holly C Greenberg. *The Application of Origami to the Design of Lamina Emergent Mechanisms (LEMs) with Extensions to Collapsible, Compliant and Flat-Folding Mechanisms*. Brigham Young University, 2012.

[7] Guorui Li, Xiangping Chen, Fanghao Zhou, Yiming Liang, Youhua Xiao, Xunuo Cao, Zhen Zhang, Mingqi Zhang, Baosheng Wu, Shunyu Yin, et al. Self-powered soft robot in the mariana trench. *Nature*, 591(7848):66–71, 2021.

[8] Serope Kalpakjian, Stephen Schmid, et al. *Manufacturing, engineering and technology SI 6th edition-serope kalpakjian and stephen schmid: manufacturing, engineering and technology*. Digital Designs, 2006.

[9] Kaylie Barber and Kylie Barber. Foldable compliant origami swimmer. *Carnegie Mellon Robotics Institute Summer Scholars Working Papers Journal*, 10, 2022.

[10] Jared Buter, Nathan Pehrson, and Spencer Magleby. Folding of thick origami through regionally sandwiched compliant sheets. *Journal of Mechanisms and Robotics*, 12(1), 2020.

[11] Hankun Deng, Patrick Burke, Donghao Li, and Bo Cheng. Design and experimental learning of swimming gaits for a magnetic, modular, undulatory robbot. *IROS*, pages 9562–9568, 2021.

[12] J. Zhu, C. White, D. K. Wainwright, V. Di Santo, G. V. Lauder, and H. Bart-Smith. Tuna robotics: A high-frequency experimental platform exploring the performance space of swimming fishes. *Science robotics*, 4(34), 2019.

[13] Junzhi Yu, Min Tan, Shuo Wang, and Chen Erkui. Development of a biomimetic robotic fish and its control algorithm. *IEEE Transactions on Systems, Man, and Cybernetics*, 34(4):1798 – 1810, 2004.

[14] Mathieu Porez, Frédéric Boyer, and Auke Lispeert. Improved lighthill fish swimming model for bio-inspired robots: Modeling, computational aspects and experimental comparisons. *The International Journal of Robotics Research*, 33(10), 2014.

[15] Robert Katschmann, Joseph Delpreto, Robert Maccurdy, and Daniela Rus. Exploration of

underwater life with an acoustically controlled soft robotic fish. *Science robotics*, 3(16), 2018.

[16] Koichi Suzumori, Satoshi Endo, Takefumi Kanda, Naommi Kato, and Hiroyoshi Suzuki. A bending pneumatic rubber actuator realizing soft-bodied manta swimming robot. *IEEE International Conference on Robotics and Automation*, pages 4975 – 4980, 2007.

[17] Byungkyu Kim, Deok-Ho Kim, Jaehoon Jung, and Jong-Oh Park. A biomimetic undulatory tadpole robot using ionic polymer–metal composite actuators. *Smart Material Strutures*, 14:1579 – 1585, 2005.

[18] P. W. Webb. Body form, locomotion and foraging in aquatic vertebrates. *Integrative and Comparative Biology*, 24(1):107–120, 1984.

[19] Zane Wolf and George V. Lauder. A fish-like soft-robotic model generates a diversity of swimming patterns. *Integrative and Comparative Biology*, 62(3):735–748, 2022.

[20] F. E. Fish and G. V. Lauder. Passive and active flow control by swimming fishes and mammals. *Annual Review of Fluid Mechanics*, 38:193–224, 2006.

[21] Won-Shik Chu, Kyung-Tae Lee, Sung-Hyuk Song, Min-Woo Han, Jang-Yeob Lee, Hyung-Soo Kim, Min-Soo Kim, Yong-Jai Park, Kyu-Jin Cho, and Sung-Hoon Ahn. Review of biomimetic underwater robots using smart actuators. *International journal of precision engineering and manufacturing*, 13:1281–1292, 2012.

[22] Yi Li, Yuteng Xu, Zhenguo Wu, Lei Ma, Mingfei Guo, Zhixin Li, and Yanbiao Li. A comprehensive review on fish-inspired robots. *International Journal of Advanced Robotic Systems*, 19(3):17298806221103707, 2022.

[23] Mohammad Sharifzadeh, Yuhao Jiang, Amir Salimi Lafmejani, Kevin Nichols, and Daniel Aukes. Maneuverable gait selection for a novel fish-inspired robot using a cma-es-assisted workflow. *Bioinspiration & Biomimetics*, 16(5):056017, 2021.

[24] Dohgyu Hwang, Edward J Barron III, ABM Tahidul Haque, and Michael D Bartlett. Shape morphing mechanical metamaterials through reversible plasticity. *Science robotics*, 7(63):eabg2171, 2022.

[25] Yinding Chi, Yaoye Hong, Yao Zhao, Yanbin Li, and Jie Yin. Snapping for high-speed and high-efficient butterfly stroke–like soft swimmer. *Science Advances*, 8(46):eadd3788, 2022.