The Soft Robotics Laboratory, ETH Zürich

Jonathan Distler

May, 2025 to August, 2025

Special Thanks to Dr. Ronan Hinchet, Dr. Robert Katzschmann, Matthew Fernandez, and Mike Michelis

Support Provided by the Cornell Engineering Undergraduate International Summer Research Grant and the Tatum Career Award Scholarship

Project Goals

During my time at ETH Zurich over the summer of 2025, I primarily worked on hardware setup and simulation for a soft-robotic fish, a direct evolution of the SoFi fish developed by Dr. Robert Katzschmann (SoFi Ted Talk. The goal of this project was to ultimately develop an optimized tail to produce realistic thrust results for a robotic fish that were of the same (or close to) the expected thrust results for a fish exhibiting BCF (Body and/or Caudal Fin Propulsion) locomotion (A Review of Locomotion, Control, and Implementation of Robot Fish, Xenyu, Sept., 2022). The goal of the project was to create a robotic fish in a cost-effective and reproducible manner. Initially, my first projects involved developing and implementing instruments that would be crucial to the fish development at all stages of the testing process and in In-Vitro/untethered testing. These projects were as follows: initial proof of concept; autonomous control coupled with an on-board camera, Raspberry Pi compatibility, and then a physics-based simulation. Following the setup of the static test rig, I was responsible for developing hardware systems to aid in testing and data-collecting, as well as analyzing the data.

Proof of Concept

For the initial proof of concept, we used a scotch-yoke style motor assembly printed with PLA. The scotch yoke was constrained by two vertical constraints, such that the motion of the yoke was almost entirely linear in the plane of the motor's motion. Then, with wire fed through holes at the end of the yoke, the movement of the motor (and subsequent movement of the scotch) would result in a tensioning of the line on the respective side of the yoke that was being pushed. When coupled with the thread being threaded through eyelids on the tail, this produces a bending of the tail.

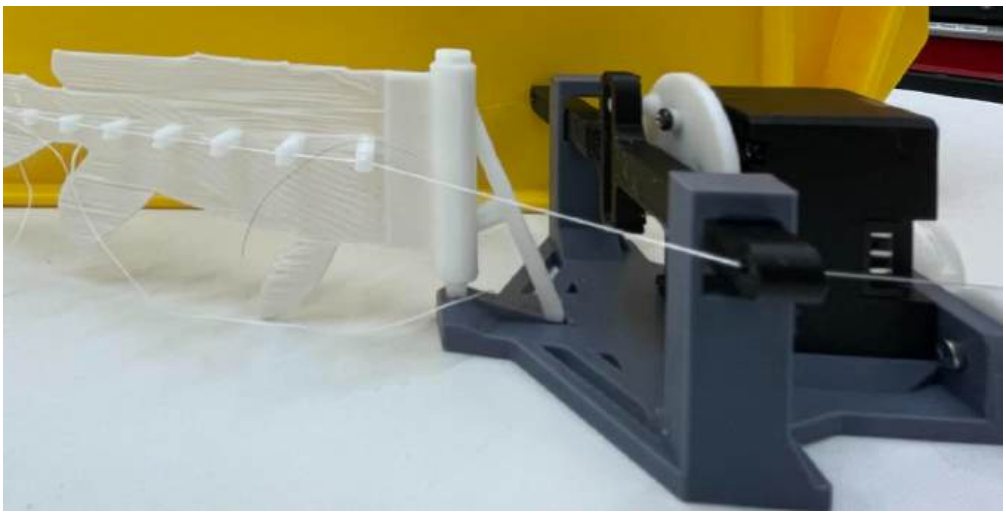

Figure 1: Side View of the Proof of Concept for the Scotch-Yoke Style Tendon Driven Fish

Figure 2: Front View of the Proof of Concept for the Scotch-Yoke Style Tendon Driven Fish

I forked Matthew's Dynamixel motor control class (Motor Control), and when using my specifically designed 3D-printed parts, I was able to produce a relatively large radius-of-curvature (Tail Movement). One issue we noticed at the beginning was that even small perturbations in the tension of the line result in non-uniform bending of the tail out-of-water, which would only be exacerbated in the water. Eventually, after the proof of concept showed that the fish's tail was capable of bending, Matthew designed a larger version of the fish with a gear-box to improve the frequency output range of the Dynamixel, as well as tensioning attachments and a more modular tail assembly.

Autonomous Control

It was thought that autonomous control would be crucial in an ultimately untethered fish, which would then allow the project to be implemented in a plethora of environments and without the constraints associated with tethering. Bluetooth modules weren't a viable option because strong water absorption occurs at the wavelength of Bluetooth modules (2.4 GHz).

Instead, we had to research solutions for radio-waves at a sub-GHz range. This led us to discover a commonly used wavelength at the 433MHz wavelength. Luckily, there were quite a few options at that range, including the TinyCircuit line.

We used the *TinyZero processor board* (Processor Board) with the *433MHz Long-Range Radio Shield* (433 MHz Radio-Shield), which conveniently attaches on top of the processor board with minimal added height. To properly flash the TinyZero boards with the intended code, they can be configured with the Arduino IDE as long as the TinyCircuits SAMD Boards module is downloaded from the IDE.

One board has to be arbitrarily considered the sender (and will be permanently connected to the computer), whereas the other will be considered the receiver (and has to be connected to a power source -in our case, the following: Lithium-Ion Polymer Battery).

- The sender is flashed with the following code: TinyZero Sender Code. The idea is that once a connection occurs between the 'sender' and the 'receiver', the sender can type certain messages. In our case, it was just the letter "W" (the thought was future iterations on the project could use other key-binds like "W-A-S-D" to send specific messages to the receiver for certain motor controls). The letter "W" is sent and decoded by the receiver. If it is properly decoded, the receiver sends a reply along the lines of "Reply from receiver: W". If nothing is sent from the receiver, it is a sign that the connection between the two is temporarily lost. For our testing, the sender was plugged into one of our computers.

- The receiver is flashed with the following code: TinyZero Reciever Code. If it receives "W" and "W" only, then it will send a response saying that it had received the correct message. This message will be received by the 'sender' and will be output to the Arduino IDE for the user to read. For our testing, the 'sender' was plugged into a RaspberryPi.

- Previous testing showed that it was possible for the TinyZero to send directions and remotely control the Dynamixel motor position (and by induction, its velocity and any other of its parameters). The culmination of my efforts was the following script. When the user types "W" into the Arduino IDE, the "sender" sends the radio signal to the "receiver," which was connected to a Raspberry Pi. Then, the following Python script decodes the message received by the receiver and turns the message into a control input for the Dynamixel motor: Autonomous Dynamixel Control

Other tests showed that the TinyCircuits were able to stay connected to each other to a reasonable distance and with reasonable depth underwater. Anecdotally, we were able to send a message from my laptop at a computer 2m away from a water bath with the receiver submerged at a depth of .5m. This testing was never done with the **Dynamixel Control**; however, it shows that the autonomous control would be completely possible underwater. Further testing could incorporate more keybinds and more complex motor movements, as well as further testing to determine the exact point at which the radios disconnect from one another.

Another crucial step in readying the fish for use was the implementation of the TinyZero IMU (using the TinyZero board, which can be bought pre-configured with a 3-Dof IMU), as well as the implementation of a more robust IMU. In order to understand the fish's orientation in 3-D space for live-time testing, it is necessary to have some semblance of on-board orientation, typically an IMU. The TinyCircuit provides the most basic viable setup. First, the TinyCircuit is flashed with the following code: TinyCircuit IMU Code -C++. Then, after the TinyCircuit is flashed with the C++ code, the user should check the Serial Plot to make sure the results make sense intuitively. Then, the user should close the serial-plotting in the Arduino IDE to allow serial communication with the following Python script: TinyCircuit IMU Code -Python. This script streams the IMU data from the TinyZero to the user's command-shell in an easily viewable format.

One issue we found was that the IMU only provided at most 3-DoF, whereas an alternative IMU, The Adafruit BNO-055 module (Adafruit BNo-055 Specification Sheet), can reasonably provide 6-DoF as well as temperature and magnetic-field data, so it was thought that it would be best to move away from the TinyZero IMU in favor of the more robust Adafruit BNo-055 module. However, if it is a goal to package the robot in as small a configuration as possible, then the TinyZero IMU provides enough data to be usable. The code to stream the IMU data from the Adafruit module can be found in this script: Adafruit BNo-055 IMU Streaming. Furthermore, I also created a script that implements the **Dynamixel Control** script with the IMU data streaming: Adafruit BNo-055 IMU Streaming and Dynamixel Control

Future work could implement a closed-loop system where the IMU data is used to directly inform motor control, or to autonomously control the Dynamixel Motor and map the accuracy of the orientation with respect to intended orientations. Additionally, the IMU internally handles the gryo-drift's gravitational bias, but the system is a black-box, where I was unable to see how their system compensates for the IMU aligning with the gravitational field, further testing could work on better understanding the accuracy of the tilt compensation function within the IMU, or developing additional checks (possibly implementing a second IMU, and comparing their readouts) to manually override the gyro-drift.

MuJoCo Tendon Fish Setup

As a secondary task, I was asked to develop a MuJoCo (Multi-Joint dynamics with Contact) simulation involving a tendon-driven fish, similar to our real-life test setup. The goal for the simulation was to use preliminary test data to inform parameters of the simulation. Then, eventually, different tail iterations could be fed into the MuJoCo simulation to test thrust outputs. I followed many of the examples provided by Mike Yan Michelis and linked in his repository: Mike Yan Michelis' `fishCREATE` repository.

Below are images of our real-world static thrust-force test setup. The idea is that a vertical rod is rigidly attached to a position on the robotic fish's head. The vertical rod is then attached to a translationally constrained horizontal rod, such that when the vertical rod is displaced, the horizontal rod rotates. The rotation of the horizontal rod drives a pin in and out of the HX711 load-cell, which tracks the force output of the system. The vertical

rod is constrained such that it can pitch (to a small angle -which was approximated by the small-angle approximation for many of the distance calculations) and yaw (to allow the force to be broken into components parallel and perpendicular to the fish's head). The system was also configured with a USB-camera, and the fish was controlled via the **Dynamixel Control Script**.

Figure 3: Back View of the Static Thrust-Force Assembly

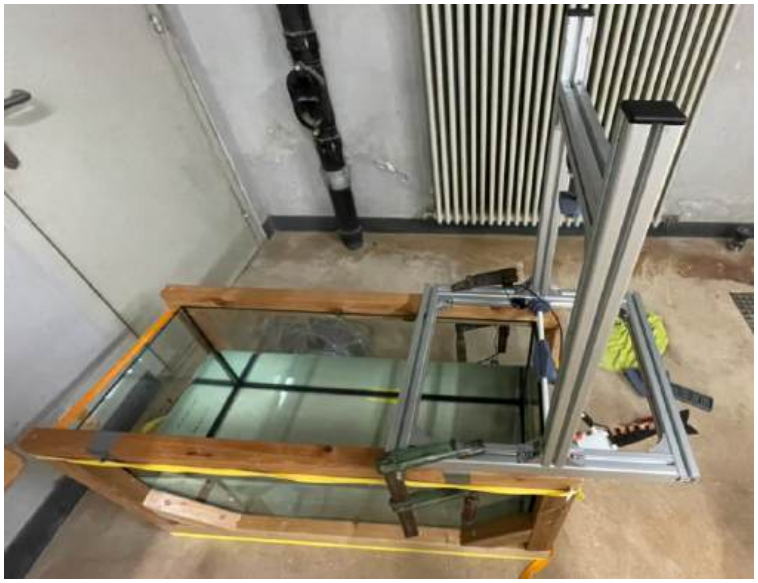

Figure 4: Side View of the Static Thrust-Force Assembly

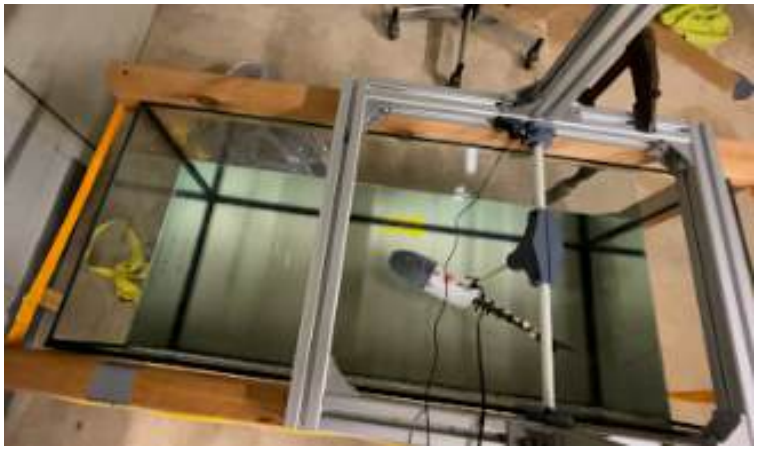

Figure 5: Orthgraphic View of the Static Thrust-Force Assembly

The core of the simulation involves creating an XML file that defines the fish model using MuJoCo's hierarchical body structure and geometry definitions. To visualize the XML model in MuJoCo, I navigated to the MuJoCo binary folder via terminal:

```
cd "C:\Users\15405\OneDrive\Desktop\Career
\ETHZ\ETHZ Work\mujoco-3.3.2-windows-x86_64\bin"
```

Then, I executed the simulation using the following command (with the appropriate path to my XML file):

```
./simulate "C:\Users\15405\OneDrive\Desktop\Career\
ETHZ\ETHZ Work\xmlfish.xml"
```

This will load the XML file, and if you define features like an actuator, then they can be manually controlled from the MuJoCo environment. I followed many of Mike's examples, while trying to integrate our fish's design into the overall fabric of the XML code, such as integrating STL files and trying to build an accurate test environment. I also developed many different actuation types (some more analogous to the scotch-yoke than others) to better mimic the movement of the MuJoCo fish.

At the time of writing, I have two useful MuJoCo simulation bodies, with a plethora of other bodies kept in the following repository, where they showcase some unique aspect of a tendon-driven fish (ie, different actuation methods, different sizes, different force setups): MuJoCo Fish Iterations.

The first MuJoCo body that I found useful was the following file: MuJoCo Fish First Iteration. This XML file builds a tendon-driven fish and static-force test setup that mirrors that of our real-world setup visually; however, I changed some of the joint types along the force-amplification system so that they weren't necessarily constrained the same way as in real-world testing (the two-rod system turning translational fish-movement into rotational rod movement and producing a force on the force-sensor). I found this result produced force-outputs at the same order of magnitude and the same visual appearance (sinusoidally varying between compression and tension with roughly equal magnitudes among both) as the real-world testing.

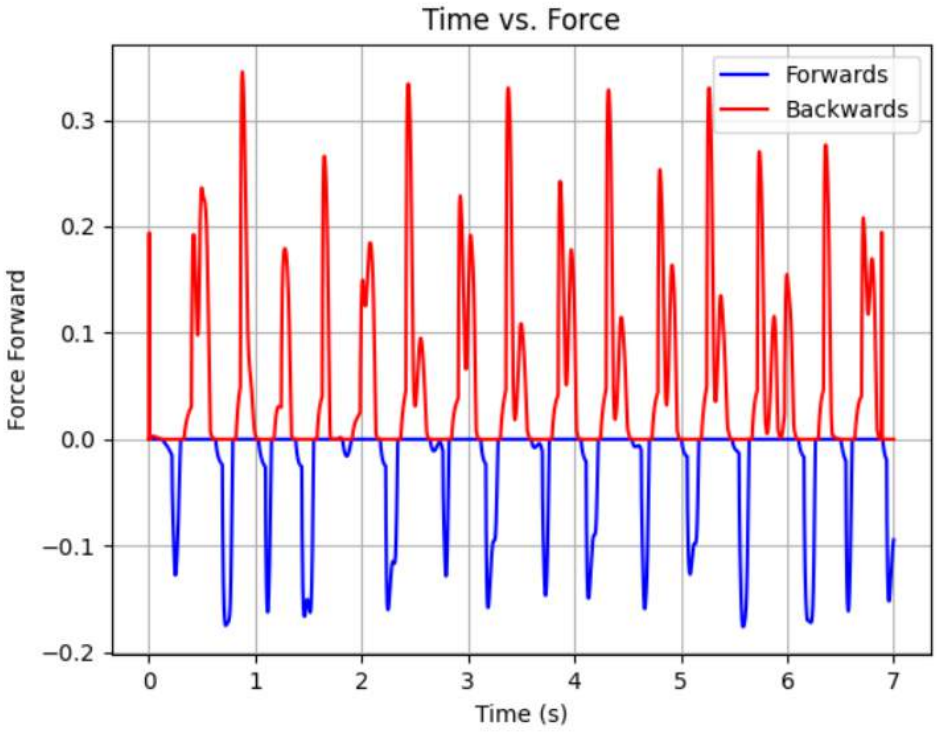

Figure 6: MuJoCo Thrust-Force vs. Time Graph

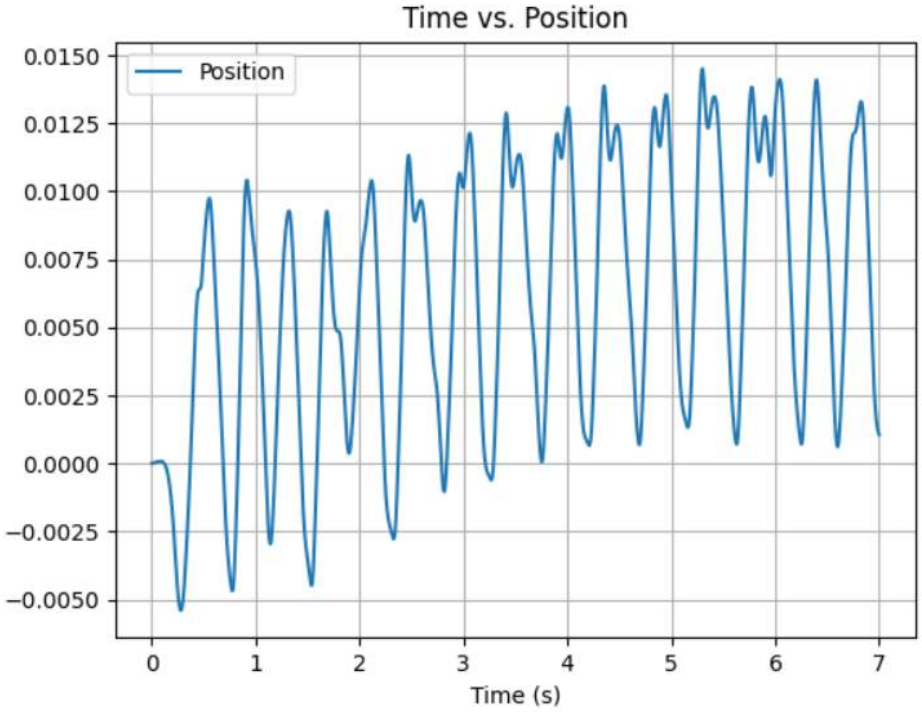

Figure 7: MuJoCo Position vs. Time Graph

To run this file and produce the test results, I used the following Python script to view

and plot the results: MuJoCo Fish Follower - First Iteration, and the following script to record and plot the results: MuJoCo Fish Videography - First Iteration

The second MuJoCo body that I found useful was the following file: MuJoCo Fish Second Iteration. This XML file also builds a tendon-drive fish and a static-force test setup, but it uses realistic constraints to almost exactly reproduce the testing setup of the real world. The main issue is that the results aren't as idealized as in the first case, with there being little to no sinusoidal motion among the tension and compression. However, once I begin to optimize this XML file using a Nelder-Mead optimization scheme, I believe that this will produce as good, if not better, results as the first iteration, and it will be more useful in long-term testing. To run this particular file and produce its test results, I used the following Python script to view and plot the results: MuJoCo Fish Follower - Second Iteration, and the following script to record and plot the results: MuJoCo Fish Videography - Second Iteration

In order to optimize my XML scripts, I followed Mike's advice to use a Nelder-Mead Optimization. Without going too much into the proof of the optimization technique, it pretty much takes the parameters you want to vary (for example, in my case, I varied the tendon stiffness and damping coefficients, as well as the fluid-coefficients) and a function you want to optimize (for example, I built scripts maximizing the force-output, maximizing the distance traveled in 1 s, and following a set equation to minimize the difference in maximum amplitudes within the front and back-stroke portions of the swim). Then, it iteratively changes the parameters with your set tolerances in order to find a minimum value, or in the case of trying to maximize a function, it tries to minimize the negative version of the value.

I plan on using this algorithm to change the second iteration of the XML tendon fish in order to try to match the real-world testing as close to 1:1 as possible; however, that relies on better and more accurate real-world testing, which as of now, has been pigeon-holed by shipping and ordering delays. Another issue I came across was the motor control. I couldn't find a real-world analog to the motor control behaviors in the MuJoCo simulation, so I mostly operated the motor at a constant angular velocity (in most cases at a rate of 20 units/2 -with no pre-defined units in MuJoCo it can be difficult to develop real-world analogs with their given control parameters). In the future, with better real-world results, I would like to compare different motor control versions (ie. constant torque, constant angular velocity, sinusoidally varying position, etc.), as well as possibly implementing an IsaacGym simulation to help develop better control parameters for more accurate swimming behaviors. As an aside, I was using smaller magnitudes of damping and stiffness (first order of magnitude for both, whereas, Mike had used stiffness and damping 20,000). This may have been one of the issues in the second test not producing the intended behaviors. Additionally, the smaller values for stiffness and damping led to chaotic behavior where a small tweak in the stiffness or damping could completely disrupt the simulation. A workaround in the Nelder-Mead Optimization was to implement a time-out, such that when the simulation has lasted longer in real-time than in simulation time (meaning the simulation has reset at a minimum of once), the steps returns a very large number to completely avoid future steps in that direction, and to force the simulation to step in a direction away from crashes. Below are four different versions of my Nelder-Mead Optimizations:

- 1) This version of the optimization is set to maximize the force-output of the fish simulation in a 1 s time-frame, varying the fluid-coefficients and the motor control-speed: Nelder-Mead Force Optimization Script
- 2) This version of the optimization is set to maximize the distance traveled from the starting position (in the arbitrarily defined negative x-direction) in a 1 s time-frame, varying the fluid-coefficients and the motor control-speed. However, given that it isn't bounded by a constraint negatively, I had to change the force-test setup to remove the constraints, so it requires a slightly different XML file: Nelder-Mead Position Optimization Script and Nelder-Mead Position Optimization XML File
- 3) This version of the optimization is set to minimize the average difference between the forward and backward strokes during a 1.5 s interval. The parameters being varied are the force-sensor locations and the head constraint positions: Nelder-Mead Average Force Difference
- 4) This version of the optimization is set to maximize the number of force-readings in a 5 s time-frame, as well as trying to maintain a semi-even spread between the number of force-readings from the forward and backward sensors, however, in the future I think it would be best to do one or the other rather than both because they are seemingly antagonistic towards each other. The parameters being varied are the stiffness and damping coefficients of the tendons and of the force-amplification rod: Nelder-Mead Force Readings

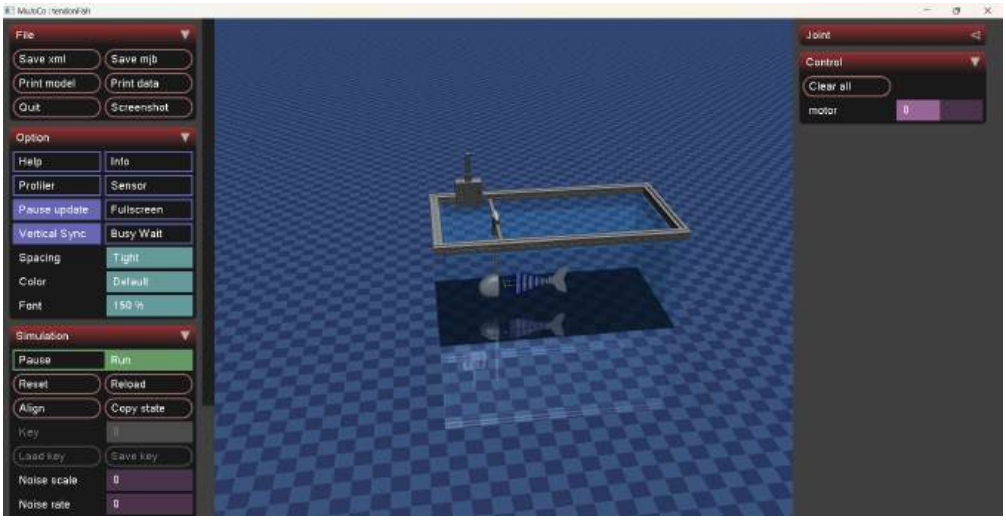

Figure 8: MuJoCo Testing Environment with Realistic Fish *(Iteration 2)* and Realistic Test Setup (*Iteration 2)*

Overall, the simulation was supposed to use many of the parameters and constraints of the real-world static testing setup Matthew and I had been using to inform many of the user-defined parameters in simulation (ie, hydrodynamic forces). Then, once the simulation was producing similar results to the real-world testing, it would be very easy to iterate over

different tail designs and quickly illustrate which types of tail designs produce the most thrust and which tail design optimizes any given number of success criteria. Now that real-world testing has produced results, one has to decide how to implement the results into the simulation. I thought it would be useful to try to minimize the difference between the maximum force output of the MuJoCo simulation and the real-world test, as well as to minimize the number of "peaks" (or relative maxima) between the MuJoCo simulation and the real-world tests. Additionally, the head-angle could be minimized to match the angle in simulation. The different matches between simulation and real-world really depend on the use-case for the simulation. For the tail iterations, radius of curvature and/or force-output would be the most important factors to consider. However, first the real-world tests need to be verified first before even considering implementation into the MuJoCo environment.

Hardware Testing

Hardware Setup

As for the hardware setup, the goal of our experiment was to streamline all of our data together in an easy-to-digest format that would provide information on the components of thrust generated by the fish, as well as the respective motor inputs that lead to desired outputs. First and foremost, I had to configure the load-cell (HX-711). There was code online for the load-cell, but I found that occasionally the input and output pins had to be reversed physically and remain the same in the code to work (and vice versa). Below is the reliable test setup I found to work [Red-wire to 5V, Black-wire to Ground, Blue-wire to Pin-2, Green-wire to Pin-3]. I also found it helpful to use alligator clips to attach to the cables and then use M-M jumpers, rather than having to solder every time the force output became too great and a new load-cell had to be used. I also found that the Arduino AVR Board module had to be downloaded on the Arduino IDE (for the case of our particular Arduino). One of the issues we came across was damage of the board during testing. De-soldering was very difficult, so in the future, it could be helpful to have a few different Load-Cell Amp boards on hand.

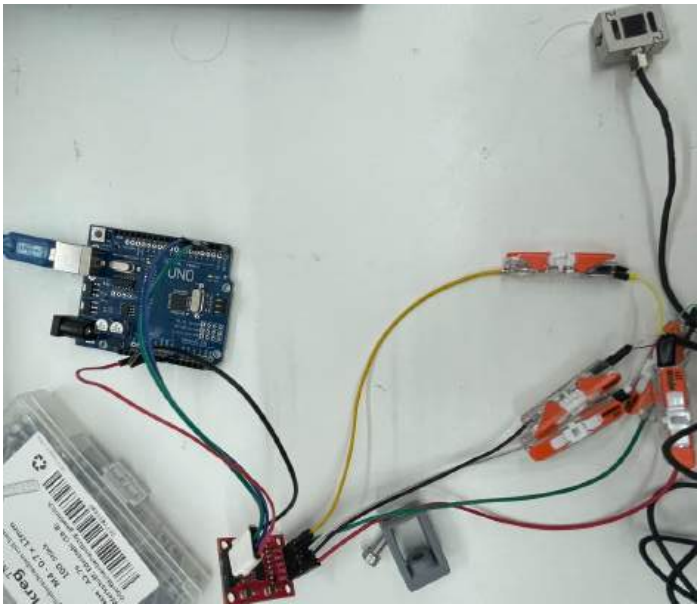

Figure 9: HX711 Load Cell Setup with Red-wire to 5V, Black-wire to Ground, Blue-wire to Pin-2, and Green-wire to Pin-3

Load-Cell

For the load cell, first, it needed to be calibrated. This came down to using pre-measured weights along with the calibration code (Calibration Code) to change the coefficient associated with the particular load-cell. It ended up being different for every load-cell I used. I found it was helpful to measure using 4 significant figures, and to do so for a minimum of three different pre-measured weights. Then, I either interpolated between the load-cell calibration factors (Interpolation Script) or I just purely averaged the calibration factors (Average Script). Once the load-cell was calibrated, I then tested a previously unused mass to see whether or not my calibration factor worked for the load-cell output (Output Script). For the 20-kg cell, I found that the calibration factor was roughly

I found that typically the output code doesn't produce a completely accurate result (usually within .25% to .5% of the expected value), but it does provide a very good bound. For the 10-kg load cell we used in our test setup, I found that a calibration factor of -355,000 was sufficiently accurate, with accuracy up to 13%. Once the Arduino is outputting force values (in kg), a simple conversion by the moment-arm factor and the gravitational constant results in an accurate force output in N, however, the moment-arm factor changes from fish-setup to fish-setup, so this does introduce some specificity in the results from system to system. For the 20-kg load cell, I found that the calibration factor was -217500, although the calibration itself wasn't super accurate given the allowed tolerances of the load-cell.

Camera

After the load-cell was calibrated, we also needed to gain visual feedback for the form of the fish, the fish's angle concerning the origin, and the tail radius-of-curvature. Firstly,

we decided on a fish-eye lens because it is more analogous to real-world fish-sight, partially because it captures a wider field of view. Our group purchased the IMX-335 USB-camera with the fish-eye lens (Camera Specifications). This is the code I developed to stream video output from the camera to save the video locally: Camera Control. I found that writing a specific-resolution (in our case, the highest possible resolution according to the camera specifications -2592x1944) caused no issue with the streaming output. The real issue came from writing a specific FPS. On its own, the camera is able to stream at a desired rate up to 30-FPS, but when using other functions (such as the load-cell and dynamixel motor control), the camera can't physically keep up with the desired rate.

However, because we are writing the camera at that particular rate, it has to condense multiple seconds of frames into one-second intervals to keep up with the desired FPS, so when using all of the functions at once, it ends up being 2 times as fast as in real-time (for example, if we're writing the video at 30 FPS and the camera can only stream at 15 FPS because the other functions are causing a streaming delay, then it will write 2 seconds of live-video into a 1 second output-video). One solution was to post-process the video with both the camera-calibration (to revert to a normal viewing angle rather than fish-eye lens) and to slow down the video at a rate proportional to the FPS and actual time of the video. Below are the different links to each calibration. I found that the distorted camera-view worked well for computer-vision purposes and just general streaming, but for real distances (rather than pixel distances), it is important to have an undistorted camera, although the edge-effects on the camera aren't necessarily super accurate outside of the pixels directly beneath the camera.

- This calibration uses the chessboard from the F2 room at a distance of 74.2 cm (measured distance from the back of the fish in the test-rig at the time to the lens of the camera): Camera Calibration - Chessboard

- This calibration uses an asymmetric chessboard printed out, also at a distance of 74.2 cm. Many people suggest using this method because it is more versatile when it works; however, it doesn't produce a natural saddle-point like the normal chessboard (at its respective corners), so it isn't always guaranteed to work. I found this calibration matrix to provide better results than the normal chessboard method: Camera Calibration - Assymetric Chessboard

- This uses predetermined matrices from either calibration to parse through a video frame by frame and undistort each image: Calibrated Camera - Frame Parse

- This code takes a pre-measured object (in my case, the image was a line drawn to roughly 10 cm long) at the specified distance of the camera-calibrations (74.2 cm in my case, again mirroring the symmetry of our test setup) and it determines the real-distance per pixel (in mm/pixel). Then, test data can be parsed over, and it can be understood how far different objects have moved in real distance: Pixel Distance

The latter were all issues that came from the usage of the USB-Camera. However, using the Oak-D Camera, the camera provides much more functionality to maintain a given

frame-rate throughout testing while keeping real-speed. This involved some core allocation in the combined test platform, however, the issue of having to post-process was overcome by changing cameras. Although, going to a higher frame rate than 30-FPS did change the video from the true time (as had previously occurred), but not to such a high-degree, and it maintained a constant FPS, which hadn't necessarily been the case previously, so one can still determine the true time based on the frame and the constant FPS of the video. The new script involves setting up a data pipeline, and can be found here: Oak-D Camera Setup. The lens wasn't a fish-eye, so the output was distorted on the edges, so we avoided having to do any undistorted post-processing. Overall, the USB-Camera provides a more affordable option for testing, with similar results following the postprocessing and calibration.

Dynamixel Control

After the camera was calibrated and fitted with distortion matrices, it was important to figure out how to control the Dynamixel motor. Our first test setup used a version of the Dynamixel motor; however, it appears as though we will be switching to a brushless-DC motor in the future, so much of the motor-control can be disregarded. In short, one has to first fork Matthew's Dynamixel control-class: Dynamixel Control Class, then, using the Dynamixel-Wizard app, they determine and/or write the ID of the Dynamixel motor they're using (in this case, it would be ID=1). Then, using the following script, Dynamixel Control, the USB port is automatically configured, and the user can specify a desired frequency to run the Dynamixel motor at. I found it easier to parse over every frequency between 1 and 4 by incrementing .2 Hz at a time in the script; however, it does maintain its utility with a singular desired frequency. One issue was that trying to get the motor to run at higher frequencies, such as 4 hZ, isn't physically possibly, so it reaches a maximum limit (for our case it was 1.5 hZ). However, it was useful to see the different form factors of the tail at different intended frequencies up to 4 hZ, even if the motor only realistically reached up to 1.5 hZ.

Combined Testing

To run all scripts in conjunction, allowing the user to read in the force-output of the fish, see the fish-swimming recording, and control the motor, a terminal script had to be developed. This script starts the camera first (because it takes the longest time to begin to render). Once the camera is started, it sends a signal to the working directory (from which the other two functions are looking) to begin. Once the motor reaches its goal position, then it will send another signal (like the first) which begins to shut down the other functions. I found that at repeated frequency intervals it was sometimes the case that the motor would hold onto previous data, in an attempt to flush the data, I return the motor back to its original starting position (this seemed to alleviate the problem), which is why the motor doesn't shut-off at the same time as the other functions: Combined Test Platform (as an aside these are the two commands to run this script in terminal: ***Set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass*** and then ***./MultiTestAgent.ps1***). Alternatively, I developed an "all-in-one" Python script that does all of the tasks in one Python script; however, its output rate is as high as that of the lowest rate of any one of the individual

components, so it completely disregards a great deal of data that comes with faster rates, so it is preferable to run each function separately: All-in-One Script.

After some computer issues, it was decided to turn the Powershell script into a Python script to modularize everything in one platform. The script (Combined Test Platform - Python Version also allots CPU cores and affinities in order to prioritize the camera streaming, such that there was a high (and more importantly a consistent) frame rate across all videos. It functions much like the Powershell script, writing text files to the folder of the scripts to turn them on or off at opportune moments. It implements the position control script, the force data serial communication script, and the camera script (with capabilities for the USB-Camera and the Oak-D camera). The final versions of all of the relative scripts are linked below. It should be noted that given Matthew's class setup, it was found that the **Position Control** script should be one folder below the working directory:

- This code starts the camera script and then sends out a signal to the other scripts to initialize them, this is a result of the camera taking a few seconds to properly initialize: Oak-D Camera Setup
- This code serially streams the force data from the load-cell. With the 20-kg load cell there is expected to be a lot of uncertainty (.1% accuracy in 20kg has a much higher uncertainty than .1% of 5 or 10kg). It involves running the **Output Script** from the Arduino IDE with the console-output closed. The code occasionally doesn't stream. I found it helpful to flash the Arduino Uno board multiple times, as well as unplugging and then plugging back in the board: Load-Cell Setup
- This code controls the position of the motor using a Dynamixel Script, forked from Matthew's class. Occasionally the motor will encounter a hardware issue, in that case reboot the motor; this is usually a result of a loose part or the strings being too tight: Dynamixel Control Script

Computer Vision

After the testing had been completed, all of the data had been imported to my computer locally. I was tasked with determining the component of force in the direction of the head, as well as the tail radius-of-curvature. I used a computer-vision script recommended to me by Mike that tracks the pixels of a bounding-box for the duration of the video. I also developed a function that changes the contrast of each frame to make the pixels more distinct and easier to view (I found an alpha value of 1 and a beta value of 45 were very good for the test setting in the MLA A22.2 room). One issue we struggled with was how to develop "markers" for more differentiated pixel tracking. It was recommended to use UV glue with some embedded reflector. We used red-sharpie (the color red is known for high-contrast), but often bled into the water. Below are different versions of the computer-vision program and their intended use case:

- This script determines the radius of curvature of the tail, as well as the angle difference between each tail-segment and the subsequent tail-segment, as well as all with respect

to the fish-head. This is very useful for determining how the radius-of-curvature influences thrust values. This also outputs raw pixel distances between each segment in time, which can be used along with the **Pixel-Distance** function to output real-world distances between each segment. To use the code, simply draw a bounding box from the end of the tail (box-by-box) in the direction of the head: Relative Tail Angle

- This script requires a bounding box around the pivot-arm of the test-setup (as a static world-body), then a bounding box around some marker on the fish's head. This script then tracks the angle of the fish head with respect to the angle of the bounding box's center. This shows how much the fish's head is yawing with time, and this can be broken into components and used alongside an interpolation script (Interpolated Thrust Forces) to determine the component of thrust in the x and y-directions produced by the fish. The current script parses through all of the videos at once, but again, it doesn't lose utility with just one video at a time: Head Angle.

Below is an example of the computer vision output. The script also writes the force-output, time, and motor goal-position when used in conjunction with video produced by the combined testing. This is without the undistorted camera, as can be seen by the edges of the frame. I found that because the camera was directly above the fish, it was accurate without calibration, and when using calibration, the video quality produces unrealistic views of the test setup away from the vertical projection of the camera, this could be alleviated by lowering the camera on the test-rig or just using the raw camera-output.

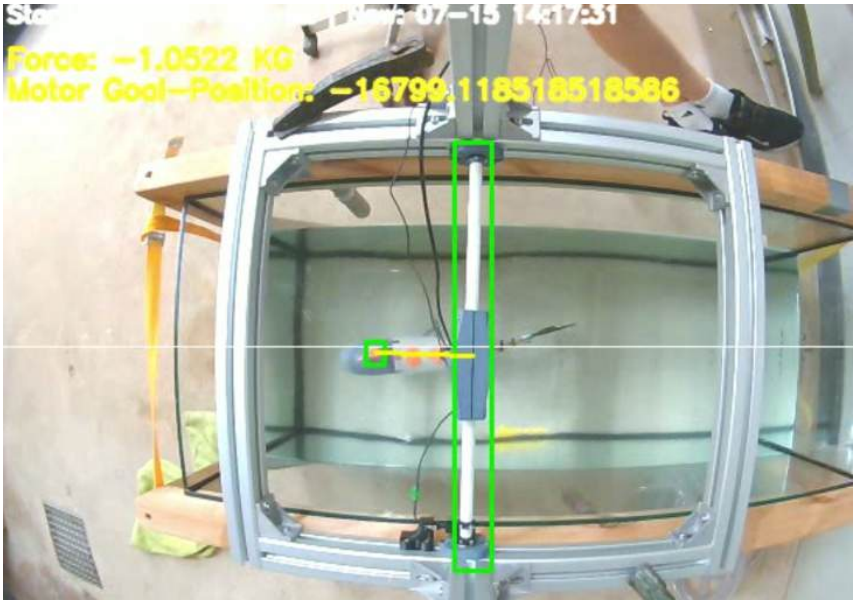

Figure 10: Example Computer-Vision Head-Tracking Output With User-Defined Bounding Boxes

Future testing could look into a wider range of contrast values and light settings; however, I think these are sufficient for the smaller scale testing being conducted at the moment, producing intuitive and reliable results.

Data Analysis

Time vs. Thrust-Force

Once testing was conducted and data was derived from the combined testing and computer vision processes, I was also responsible for analyzing the data. The first analysis I did was purely a Time vs. Thrust force. Using the Load-Cell data plotted versus time, I was able to quickly derive the time vs. thrust-force relationship. The Matlab scripts I used for this data analysis are: Maximum Force and Force Parse. The first script sets a for-loop, for which it is assumed that the frequency tests follow a consistent increment (ie, every test increases by .2 Hz from the previous test). If not the case, then simply copy and paste over the for-loop changing the boundaries to suit the data. The second script again parses over each CSV file and determines the maximum force output in each case. This was especially helpful in determining a proper magnitude of load-cell. Both scripts are helped by the fact that I write my CSVs' names as their desired frequency (ie, .../2.0.csv) during the load-cell serial output script, so all of them have very consistent names to parse through.

The graphs are linked below in the "Resources" section. As is made immediately clear, there are two separate sections. The first shows a high-frequency oscillation followed by a taper towards equilibrium, followed by another high-frequency oscillation. The first high-frequency oscillation is the real data from live-motor control, whereas the second high-frequency oscillation set is a result of the motor returning to its start position to flush away its positional data for iterative frequency tests. The graphs were for the original testing in the smaller fish tank, however, there were significant edge effects, so the data needs to be taken with a grain of salt.

Time vs. Force Components Relative to Head Angle

My next task was to develop a script that took the CSV output of the **Head Angle** script, which outputs the angle of the head relative to a stationary object for all of the recording, and determines the fish's thrust force component relative to the stationary object (taken to be the origin). I started by converting the frame values of the tracking-video outputs of the **Head Angle** script into real-time values, then I interpolated between those time values and the time values of the **Force Parse** data. Then, taking the angle from the **Head Angle** and the force output from the **Force Parse** data, I was able to use simple trigonometric relationships to determine the thrust generated by the fish in component form (with +x running parallel to the initial nose position of the fish and +y following the right-hand rule relative to the +x direction). The video shuts off before the motor returns to its starting position, so it completely disregards the unnecessary section high-frequency oscillation section. The script is linked below: Interpolated Force Components. The graphs showing the force-components versus time are linked underneath the "Resources" section. I also linked a newer version of the **Head Angle** script that writes a CSV file showing the mean force magnitudes for each frequency, it also has a simpler structure: Interpolated Force Components Updated

Frequency vs. Averaged Maximum Force

My final task was to develop graphs for the tested frequencies vs. their averaged maximum force components in the +x and +y directions, as well as the total averaged force components. This data provides a baseline to determine the most successful desired-frequencies to run the motor at in order to maximize efficiency (force-output in the +x direction), as well as what frequencies to avoid because they are made up of too large a proportion of "unhelpful" force-output (output in the +y direction). The script is linked below: Averaged Maximum Force Components. Again, the graphs showing the frequency versus averaged maximum force components are linked underneath the "Resources" section. I also included a second script that includes a more up-to-date logic sequence to read the sum of the magnitudes from a previously written CSV file, with an additional benefit that it is in Python like most of the other scripts: Averaged Maximum Force Components -Python

Future Work

More Legitimized Testing

Testing Procedure

Our testing procedure was as follows. First, we calibrated the load-cell with the previously mentioned **Calibration** script for the HX-711 load cell. We measured to masses and the average calibration value for our particular force-amplification chip came out to be -217500, which was then set as a variable in the **Output** script. One issue I came across was for the particular load-cell we ended up using, rated for 20kg, the calibration factor's outputted value slid; for example, for a given calibration factor it would output a correllating mass, but soon that mass would incrementally increase. To counter this issue, we found the first instance of when our mass was produced by a calibration factor and took that initial calibration factor as the true calibration factor for the mass. Additionally, I found that it took some time to reach the threshold value of the actual mass when actually outputting rather than calibrating, this could be a result of the call to the tare-function at the beginning, regardless it eventually reaches the intended mass (with a given tolerance of .1%), then oscillates around the value.

Once the load-cell was calibrated, we moved all of the gear into the previously noted fish-tank (in the most-recent set of tests, this was done in a different fish tank, but the same principles apply). Then, after putting the fish in place, we measured an arbitrarily high desired frequency (such as 4 hZ). Then, we post-processed the servo-positional data (using a find-peaks method in Matlab) to determine the actual frequency for which the motor was moving and put that as the upper-limit, in our case this was 1.5 hZ. Then, it was our idea to do a similar arbitrarily high-test every few runs to make sure the maximum limit was still the same (showing the same tension in the lines during each test segment), however, this was not completely followed as oftentimes the motor would crash due to too much tension, so the lines were felt by hand before each test run to maintain some semblance of a reliable and consistent tension in each test run. As for some brief notes, a frequency of zero threw an error because the script was dividing by zero, this makes sense intuitively because with

no desired frequency the fish-tail wouldn't move. Additionally, it could be of some utility to use close to the same-length string in future testing to have an additional visual measure to see how the tension in the two lines compares (if they were the same length of string, tied at the same point, then they should be the same length and therefore tension at the origin).

Testing Results

Although we were able to get some tethered testing during my time at ETHZ, there were many inadequacies in the test data, strategy, or setup. For example, the tank was either too small and causing edge-effects on the tail; the tail wasn't tensioned with a high-enough degree of certainty, so it favored one side over the other during swimming; the FPS rate fluctuated, so the outputted video couldn't be accurately deemed as "real-time"; or just a combination of many small mishaps. Regardless, the most recent test data is published in separate Dropbox locations:

- 0.1 to 1.0 hZ Desired Frequency Tests
- 1.1 to 1.5 hZ Desired Frequency Tests
- Force vs. Time Graphs for 0.1 to 1.5 hZ
- Head-Tracking CV Video for 0.1 to 1.5 hZ
- Interpolated Force in Direction of Head for 0.1 hZ to 1.5 hZ
- Average Force in Each Force Component Direction for 0.1 hZ to 1.5 hZ

The main issue with these tests was that the tension wasn't entirely consistent between both threads, so it favored movement to one side rather than the other. However, this can quickly be fixed with the following script that moves the motor position to its origin (the side of the circle parallel and nearest to the fish tail): Positional Finder/Setter, as well as more replacement parts (mainly the force-amplification boards).

Autonomous Swimming

Although a lot of preliminary work was completed during the summer, the fish was intended to be a module for which future testing could be easily implemented, helping to empirically develop a new aquatic stratagem. First and foremost is the implementation of autonomous movement. According to Univeral Robots (Universal Robots), autonomous robots are robots that involve little to no human intervention to function. For the fish, this would involve algorithmic swimming and control. In order to implement, first it would involve using my **Camera Calibration Script**, from above, at a reasonable distance from test the camera, in my eyes that would be roughly 5-20 cm away (given the size of the fish tank and the computing capabilities of the Raspberry Pi 5); this calibration could involve either the symmetric or assymetric-chessboard calibrations. Then, using my PyTorch training script

(PyTorch ML Training Script), one can train a model on a very simple dataset (in this case it was the MNIST dataset).

Then, during testing, which the fish-testing setup shown below, the fish would take pictures of its underwater environment every so-often and then using the AI-algorithm's pathway flashed to the Raspberry Pi 5, to both undistort and make a prediction about the actual image underwater (via the following script: PyTorch Image Undistorted and Saved ML Model), it would try to determine the real hand-written number (from the data-set). Then, the fish could invoke a particular action based on the number it had determined, for example if it were to predict the hand-written number underwater was an "8", then it could turn clock-wise, whereas if the number were a "five" it could continue swimming forward. It could also be implemented such that predictions that fall below a certain threshold of accuracy are not followed (i.e., a prediction that is thought to be only 65% accurate would not trigger a movement behavior, this could involve a prediction accuracy in the model or just a comparison across many different time-steps in testing). This would be the ultimate show of autonomous underwater movement. Alternatively, although less impressive, the work done with the TinyCircuits could be implemented and provide an output for a closed control-loop with automation using a computer outside of the water with much more computing power than the Raspberry Pi 5.

This autonomy functions differently than the autonomous control intended with the radio-communication, as this autonomous control would have no human element in the control loop, and it would be entirely self-contained. The other autonomous control doesn't necessarily need to have a human element, however, it does necessitate a smaller range of allowable test environments. The PyTorch methods come with its own issues, such that the Raspberry Pi's computational power is many orders of magnitude less than a PC, so the PyTorch prediction would likely be extremely slow in comparison. The main issue for both at this moment is the hardware is lagging behind the software development. After ample tethered testing, untethered testing can begin and then movement can be coordinates (likely by modifying the ratio of the period spent in one direction vs. the other, to turn). However, the new method would afford a broader range of test-environments as the fish isn't constrained to be nearby a computer and/or radio-module, and it would be easier to replicate with more commonly available products.

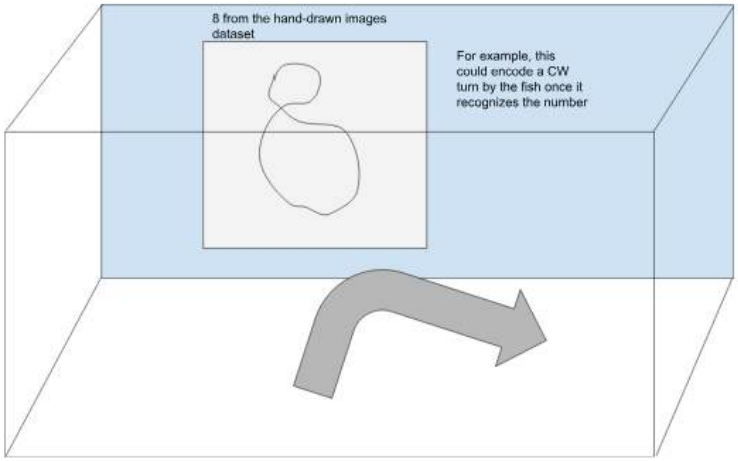

Figure 11: Example of Autonomous Fish Control

Simulation Informed Tail Design

MuJoCo simulation for informed tail design

Other future tests will hopefully include simulation-informed tail design. As noted in the Marine Waters Fact Sheet, there are different uses for different tail designs, for example some fish's primary object is to be more maneuverable, thus they use a continuous tail, whereas, some fish want to be faster and typically use a lunate tail design. The idea in a simulation informed tail-design would be to use each of the five mentioned tail types: Lunate, Forked, Truncate, Rounded, and Continuous, then design a singular test to prioritize one specific aspect of the tail (ie. maneuverability or speed vs. maneuverability, or turn radius), then test in-Vitro to determine the idea tail base-model, then iterate over many tail designs in simulation. The main simulation, as previously noted, was the MuJoCo simulation. The one issue with the simulation is the fluid coefficient can only be applied as an ellipsoidal shape, meaning the true fluid dynamics of the tail and as a result the thrust force and drug force would be inaccurate or a very unrealistic approximation. There were two potential workarounds that weren't explored. First and foremost would be the "Riemann Sum" approach. This approach would involve taking all tail iterations and breaking them into very thin, yet discrete structures, then apply the necessary ellipsoid contact set between the piece and the water. Thus, a tail structure could be assembled to mimic a real-world shape, while still possibly having a realistic drag.

Dynamical Approach

Secondly, a dynamical approach could be taken. Below are my hand-calculations for the first linkage of a fish-tail with respect to the motor's center. This approach comes with its own drawbacks. Firstly, it takes the motor as a stationary object which would absolutely

not be true for in-Vitro testing, secondly it makes an approximation that the moment on each segment is equivalent for the total angle-of-curvature calculation (the implementation of multiple segments); this can be intuited by the fact that when feeling the wires during testing, the closer wires to the motor (i.e. first segment, second segment, etc.) had less tension, whereas the further segments from the motor had more noticeable tension, as well as the fact that the moment arm (the distance from the force application to the tail itself) decreased with distance from the tail, so a smaller force multiplied by a larger moment arm should be on roughly the same order of magnitude as a larger force and a smaller moment arm. This approximation would need to be tested in order to accurately determine the approximation. Other complications involve realistically adding thrust and drag into the simulation. However, given the dynamical approach's limitations (models a static setup), the thrust and drag aren't necessarily as important as the general motion of the fish-tail, including its amplitude and period.

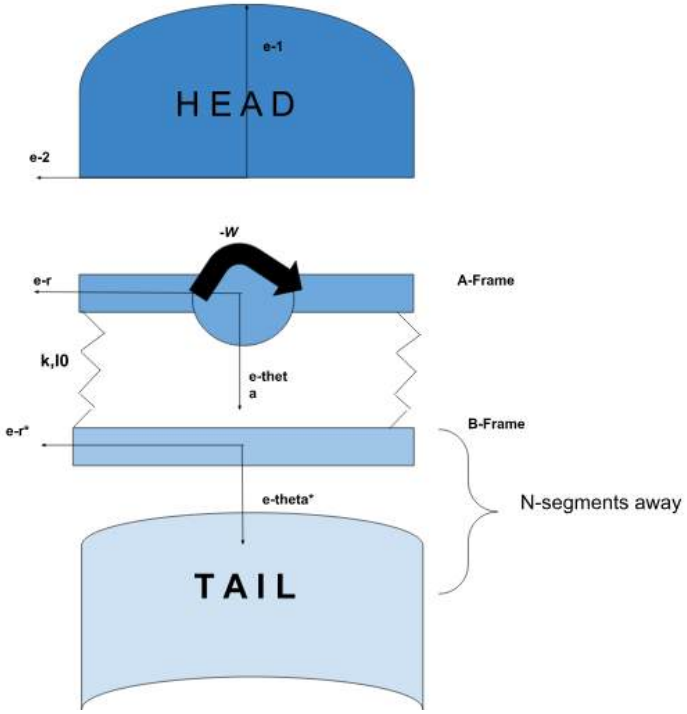

Figure 12: Top-View of the Fish with Labeled Frames for the Dynamical Derivation

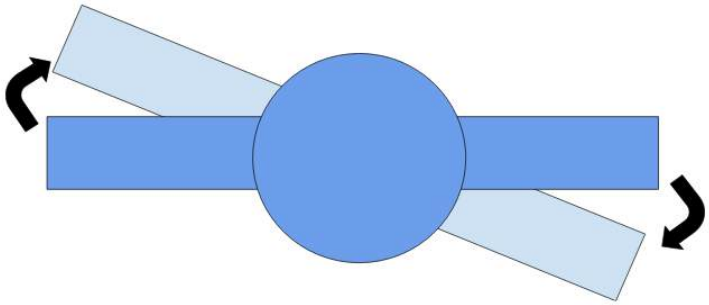

Figure 13: Top-View of the Motor-Movement, showing general approach to how the motor moves in the derivation

Another small assumption I made was that the first linkage was the same length as the motor arm meaning the spring is completely parallel to begin. This isn't an issue for the motor to the first linkage, however, for future linkages this would require the attachment point between the n-1 and n-2 linkage to be at a different radial position than the n to n-1 rod. This is applicable if the linkages are tapering off in length, which is how our real-world setup is initialized. The derivation setup would be like the following, it follows through the math, this just provides a physical intuition for the assumption:

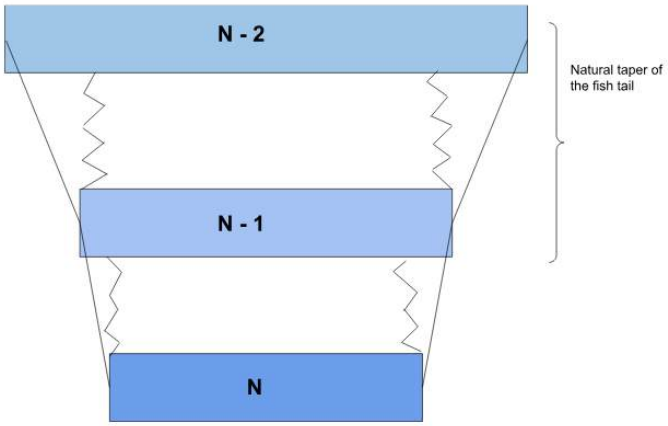

Figure 14: Theoretical Setup for the Springs in Dynamical Approach - No Changes Need to be Made, it Follows Out of Mathematical Setup

Secondly I assumed the height of the linkage and motor arm was negligible so I didn't add a y-component to the position vector. This could be avoided by placing point P at the top of the linkage and the connecting point to the motor at the bottom of the motor arm. Height could also be considered negligible in the moment-of-inertia calculations, which I would consider calculating the moment-of-inertia as a thin rod and in the limit as thickness approaches zero, it matches our model setup.

Dynamical Approach Derivation

This shows the speed and acceleration at which the tail joint moves with respect to the world-frame (the motor). The dynamical approach most resembles the tethered fishing setup.

Setting up all of the vector triangles to represent the points p and q with respect to the origin (o) and with respect to their fixed rigid body (g). Although I set up p and q both, I only follow through with the derivation of p, as q just has a negative switched on all d terms and the final result should produce a position graph that is offset by π-radians.

$$\vec{r}_{\mathrm{m/o}} = le\,\hat{\mathbf{r}}$$

$$\vec{r}_{p/g} = de\,\hat{\mathbf{r}}'$$

$$\vec{r}_{q/g} = -de\,\hat{\mathbf{r}}'$$

$$\vec{r}_{g/o} = he\,\hat{\boldsymbol{\theta}}$$

$$\vec{r}_{p/o} = le\,\hat{\mathbf{r}} + de\,\hat{\mathbf{r}}' + he\,\hat{\boldsymbol{\theta}}$$

$$\vec{r}_{q/o} = le\,\hat{\mathbf{r}} - de\,\hat{\mathbf{r}}' + he\,\hat{\boldsymbol{\theta}}$$

$$\vec{\omega}_a = \dot{\theta}$$

$$\vec{\omega}_a = -\omega e\,\hat{\mathbf{3}}$$

$$\vec{\omega}_b = \dot{\alpha} e\,\hat{\mathbf{3}}$$

Defining the direction-cosine matrices

$${}^{I}\boldsymbol{\omega}^{a}$$

	$\hat{\mathbf{1}}$	$\hat{\mathbf{2}}$	$\hat{\mathbf{3}}$
$\hat{\mathbf{r}}$	$\cos\alpha$	$\sin\alpha$	0
$\hat{\boldsymbol{\theta}}$	$-\sin\alpha$	$\cos\alpha$	0
$\hat{\mathbf{3}}$	0	0	1

$${}^{I}\boldsymbol{\omega}^{b}$$

	$\hat{\mathbf{1}}$	$\hat{\mathbf{2}}$	$\hat{\mathbf{3}}$
$\hat{\mathbf{r}}'$	$\cos\alpha$	$\sin\alpha$	0
$\hat{\boldsymbol{\theta}}'$	$-\sin\alpha$	$\cos\alpha$	0
$\hat{\mathbf{3}}$	0	0	1

Taking the inertial frame derivatives of the position of point p with respect to the origin (the center of the motor):

$$ {}^{I}\frac{d}{dt}\vec{r}_{p/o} = l\,\dot{\theta}\,e\,\hat{\mathbf{r}} + d\,\dot{\alpha}\,e\,\hat{\boldsymbol{\theta}} - h\,\dot{\theta}\,e\,\hat{\mathbf{r}} $$

$$ {}^{I}\frac{d}{dt}\left({}^{I}\mathbf{v}_{p/o}\right) = l\ddot{\theta}\,\hat{\mathbf{r}} - l\dot{\theta}^2\,\hat{\boldsymbol{\theta}} + d\ddot{\alpha}\,\hat{\boldsymbol{\theta}} - d\dot{\alpha}^2\,\hat{\mathbf{r}} - h\ddot{\theta}\,\hat{\mathbf{r}} + h\dot{\theta}^2\,\hat{\boldsymbol{\theta}} $$

Given that the motor moves at a constant angular rate, there is no second derivative of θ; therefore, all $\ddot{\theta}$ terms vanish. The result is as follows (and is only applicable to the first link; all future links will include the relevant $\ddot{\theta}$ terms):

$$ {}^{I}\mathbf{a}_{p/o} = -l\,\omega^2\,\hat{\boldsymbol{\theta}} + d\,\ddot{\alpha}\,\hat{\boldsymbol{\theta}} - d\,\dot{\alpha}^2\,\hat{\mathbf{r}} + h\,\omega^2\,\hat{\boldsymbol{\theta}} $$

Using matrix identities, the conversion between b and a frames is given by

$$ {}^{a}\boldsymbol{\omega}^{b} = \left({}^{I}\boldsymbol{\omega}^{a}\right)^{\mathrm{T}} \cdot {}^{I}\boldsymbol{\omega}^{b} $$

Setting up Newton's second law for rotational bodies

$$ \sum \mathbf{M}_p = d\,\hat{\mathbf{e}}_r \times \left(-k(x - l_0)\,\hat{\mathbf{e}}_r/\hat{\mathbf{e}}_\theta\right) $$

After converting all b-frame components into a-frame components using the previously derived matrix a omega b, the final summation is as follows:

$$ \begin{aligned} \sum \mathbf{M}_p = d\Big[&\big(\cos\theta\cos\alpha + \sin\theta\sin\alpha\big)\hat{\mathbf{e}}_r + \big(\cos\theta\sin\alpha - \sin\theta\cos\alpha\big)\hat{\mathbf{e}}_\theta\Big] \\ &\times \Big(-k\left(|h - l\sin\theta|\,\hat{\mathbf{e}}_\theta + |l\cos\theta - l_0|\,\hat{\mathbf{e}}_r\right)\ldots \\ &\cdot\frac{(h - l\sin\theta)\hat{\mathbf{e}}_\theta + (l\cos\theta - l_0)\hat{\mathbf{e}}_r}{\sqrt{(h - l\sin\theta)^2 + (l\cos\theta - l_0)^2}}\Big) \end{aligned} $$

Using $I_p = I_{\mathrm{rod}} + m\left(\frac{d}{2}\right)^2$ from the parallel axis theorem, as well as the fact that:

$$ I_p\ddot{\alpha}_p = \sum \mathbf{M}_p $$

The terms can be reorganized to solve for $\sum \mathbf{M}_p$ and numerically integrated. This is the corresponding Matlab script that integrates over my equation-of-motion for the first tail segments left-most side. The right most-side would produce almost the exact same results, differing only by Pi-radians: Runge-Kutta Numerical Integration). The results of varying spring-constants is shown below:

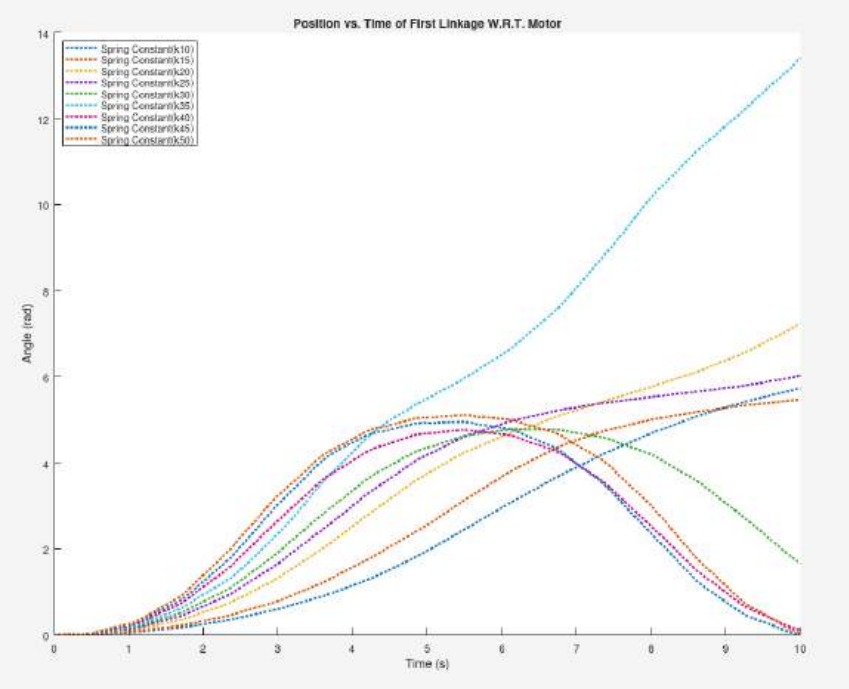

Figure 15: Spring Constant vs. Time for a Runge-Kutta Numerical Integration of Previously Derived Equation of Motion for 1 Degree-of-Freedom Tail Segment

As for adding additional tail segments, the derivation follows almost identically to the steps above. However, as previously noted, the $\ddot{\theta}$ terms do not cancel out because the upstream joints (which we denote as θ) do not necessarily move at a constant rate. To integrate the dynamical model, all equations of motion must be expressed in the inertial frame of reference — in this case, the center of the motor. As a result, the final step, which applies Newton's second law for rotational systems, must account for the influence of all preceding tail angles. Therefore, the added term becomes:

$$\ddot{\alpha}_i = \frac{\sum_{j=1}^{i-1} \frac{M_j}{I_j}}{I_i}$$

Where M_j and I_j represent the moment and moment of inertia of the j-th link, respectively, and I_i is the moment of inertia of the i-th link, i.e., the link for which the dynamic model is being derived. As with the previous derivation, this expression is projected only in the $\hat{\mathbf{e}}_r$ direction to produce the scalar equation of motion. Therefore, any components in other directions are omitted.

$$\ddot{\alpha}_i = d_i \left(\cos\theta_{i-1}\cos\theta_i + \sin\theta_{i-1}\sin\theta_i\right) \cdot \left(- k\left(\left(\left| h - l_{i-1}\sin\theta_{i-1} + l\cos\theta_{i-1} \right| - h_{i/i-1} \right) \right.\right.$$
$$\left.\left. \cdot \frac{h_{i/i-1} - l_{i-1}\sin\theta_{i-1}}{\sqrt{\left(h - l_{i-1}\sin\theta_{i-1}\right)^2 + \left(l - l_{i-1}\cos\theta_{i-1}\right)^2}} \right)\right)$$
$$+ \frac{\sum_{j=1}^{i-1} \frac{M_j}{I_j}}{I_i} \hat{\mathbf{e}}_r$$

One can intuit that as the tail segment number increases, the relative angle with respect to the previous segment will decrease. Consequently, the model produces a decaying function, where the output angle of a given tail segment becomes the input for the next. In the limit as the number of segments approaches infinity, the angular acceleration $\ddot{\alpha}_i$ converges toward zero with respect to the previous term.

This behavior assumes approximately constant moment and moment of inertia across segments—values which can be determined empirically. However, a more accurate model may account for the tapering geometry of the tail by varying the moment of inertia along its length. Future testing could incorporate real-world measurements of inertia and torque, and allow for tuning of the spring constants both globally and locally (i.e., per segment) to better reflect the system dynamics.

Conclusion

I really enjoyed my time working on the project, and with further hardware developments, it is hoped that similar results can be applied to untethered testing, creating a versatile and cheap robotic fish. I am very grateful for the opportunity to have worked in the laboratory, and look forward to see where the provided scripts can take the project.