Student names: . . . (please update)

*Please note that this project is **graded**. **The project will be graded in two parts**.*

- *In Part 1(Project 1) you will implement and analyze a wave controller to actuate a zebrafish model to study swimming in water.*
- *Part 2(Project 2) is divided in two subparts:*
  - *In the first part you will implement and analyze a biophysically inspired neural network firing rate controller.*
  - *In the second part you will incorporate stretch feedback neurons in the the firing rate controller and study the role of stretch during swimming.*

The two projects' submissions will be done separately. The code useful to complete the project will be updated on the repository under the folder Project/ (if necessary) after each project part assigment. This file is the Assignment for Project 1.

Deadlines: Project 1: Friday 26/04/2024 23:59, Project 2: Friday 07/06/2024 23:59,

Instructions

- *Update this file (or recreate a similar one, e.g. in Word) to prepare your answers to the questions. Feel free to add text, equations and figures as needed. Hand-written notes, e.g. for the development of equations, can also be included e.g. as pictures (from your cell phone or from a scanner). The code of project 1 should contain exercise 1a to 1c.*
- *Please submit both the source file. (*.doc/*.tex) and a pdf of your document, as well as all the used and updated python code can be submitted in a single zipped file called final_report_name1_name2_name3.zip where name# are the team member's last names. Please submit only one report per team!*

Project overview and motivations - the zebrafish as a case of study

The zebrafish (Fig 1) is a key animal model in biology and neuroscience.

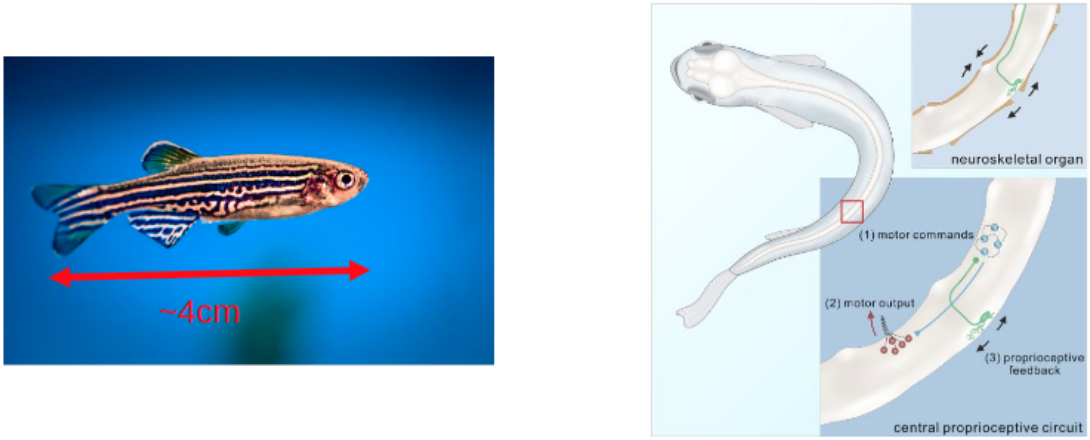

Figure 1: The zebrafish

In Project 1 and 2 you will model and simulate swimming in an adult zebrafish. It is known that zebrafish can generate swimming at three regimes (slow, medium and fast swimming). Here you will implement a controller for generating slow swimming (1-4Hz).

Interestingly, it was recently found that these animals are equipped with proprioceptive sensitive organs that sense the local stretch and use it to sense the environmental changes and update its swimming behavior, but its role is still unclear. This will be the topic of the next part of the project(Project 2).

In these projects you will have the opportunity to explore this puzzling experimental findings by building and testing a full model of the zebrafish locomotion. You will use Python and the MuJoCo physics engine to control a neuromechanical model of the zebrafish, and test some hypothesis of the working principles of the locomotion of these animals.

Simulation environment installation

Before digging into the details of the project, let's first install all the tools needed to run the simulations and test the interactive GUI of the simulation environment.

Prerequisites

To have all the necessary python packages necessary to complete the final project, check that you have installed all the necessary required packages.

IMPORTANT: Make sure you have activated and are using your virtual environment and its python interpreter that that you have created for this course.

NOTE: If you are unclear about the basic steps then refer back to Lab 0 documentation

Next, pull the latest version of the exercise repository. Navigate to the directory *Project1/Python* in the terminal and execute the following,

```
>> pip install -r requirements.txt --upgrade
```

Next, download the zip folder *farms_packages.zip* uploaded in Moodle and unzip its content in a desired directory. Copy in this directory the python file *setup_sim_env.py.* and run inside the zipped directory it by simply calling:

```
>> python setup_sim_env.py
```

This will install the *MuJoCo* simulator and the *dm_control* package which are software maintained by Deepmind for running multi-body rigid system simulations. It will also install the *farms_core*, *farms_mujoco*, *farms_amphibious* and *farms_sim* FARMS packages developed at the Biorobotics Laboratory (BioRob). If you are interested in knowing more about *MuJoCo* and FARMS, you can find out more on the official website and the FARMS paper

Examples

You can run a simulation example to get you accustomed to the MuJoCo graphical interface with **example_single.py**. You should see the polymander model floating on the water. Try running:

```
>> python example_single.py
```

Graphical User Interface Interaction

When you run the example script, a Graphical User Interface (GUI) should launch. You can use the left mouse click to move around the scene and right mouse click to rotate the camera. You can also select a part of the robot by double left clicking on a part. Once selected, you can then interact with it by holding the CONTROL key and dragging with left or right click. Try it out for yourself to familiarise with the interface.

There are many keyboard shortcuts also available

- Press SPACE to toggle play/pause
- Press "!" / "^" to change speed factor (between zero (0) and backspace)
- Press w to toggle wireframe
- Press t to toggle transparency
- Press s to toggle shadows
- Press c to show collisions
- Press f to show collisions forces, combine with p to show friction/reaction
- Press b to show external forces (try in water later on)
- And many more...

The mechanical model

The mechanical zebrafish model consists of $n_{links} = 16$ links connected by $n_{joints} = 15$ rotational yaw torques as in 2 that are modeled using an Ekeberg spring-mass-damper muscle model.

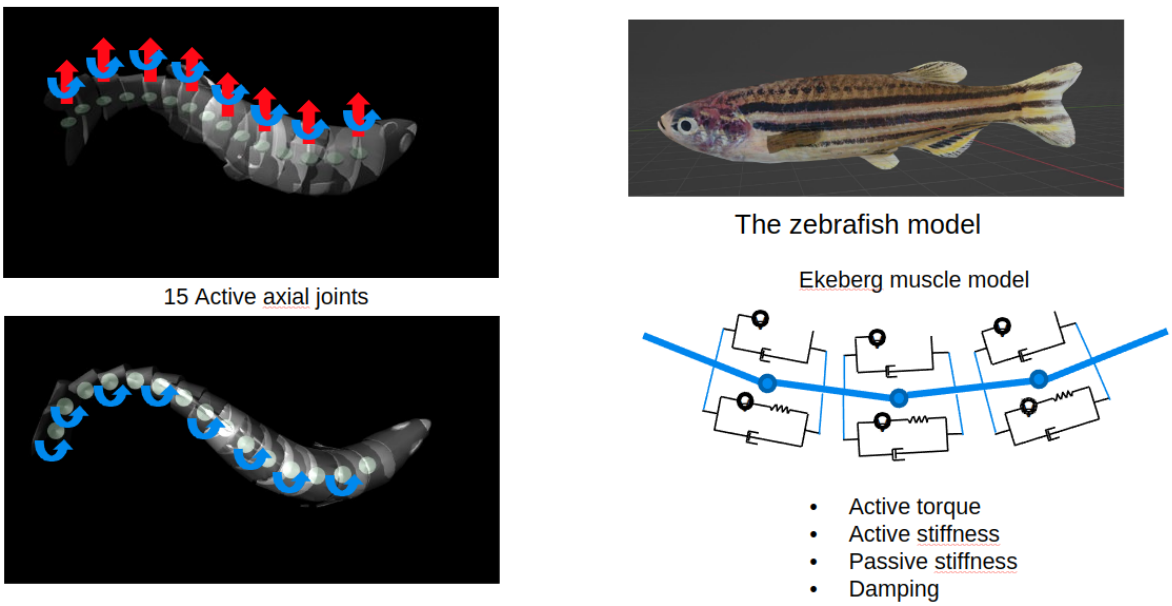

Figure 2: The mechanical model of the zebrafish

The activity of muscle cells on each joint is transduced in output torques for the mechanical model by means of Ekeberg Muscle Models (2).

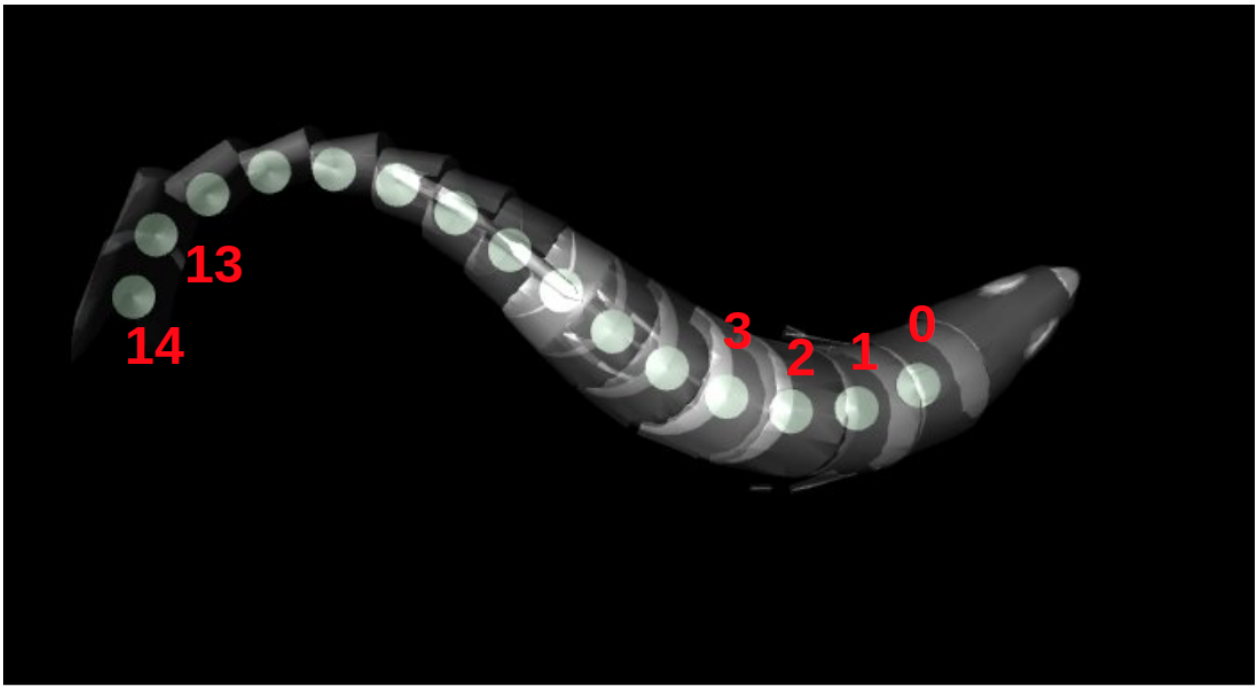

Figure 3: The joint numbering

For each joint $i = 0, ..., 14$, the model receives in input the left (M_{L_i}) and right (M_{R_i}) activations from the muscle cells, and the current joint angle (θ_i) and speed ($\dot{\theta}_i$) to compute the resulting output torque (τ_i) via:

$$\tau_i = \alpha_i M_{diff_i} + \beta_i(\gamma_i + M_{sum_i})\theta_i + \delta_i\dot{\theta}_i \quad (1)$$

$$M_{diff_i} = (M_{L_i} - M_{R_i}) \quad (2)$$

$$M_{sum_i} = (M_{L_i} + M_{R_i}) \quad (3)$$

Where α_i is the active gain, γ_i is the passive stiffness, β_i is the active stiffness, δ_i is the damping coefficient. The Ekeberg muscle model is a rotational spring-damper system with the addition of a variable stiffness term $\beta(M_L + M_R)\theta_i$. The active term of the model acts an external torque $\alpha(M_L - M_R)$.

The Ekeberg model parameters have already been optimized (you do not need to change them).

The water is simulated using a simplified drag model.

Code organization

In this exercise you will modify the files `exercise_#.py`, `wave_controller.py`, `simulation_parameters.py` (optional) and `plot_results.py` (optional). We provide below a brief description of these and the other files, for completeness.

- **example_single.py** — Run single simulation example (fixed parameters)
- **example_multiple.py** — Run multiple parallel simulations with different parameters
- **exercise_#.py** — Files for the exercises 1-3
- **wave_controller.py** — This file contains the equations for the left and right muscle activations in a step function, which is internall called at each iteration time step. It also contains the metrics

dictionary.

- **simulation_parameters.py** — This file contains the SimulationParameters class and is provided for convenience to send parameters to the setup of the controller. Please carefully read the parameter list and their documentation in this code.
- **metrics.py** — This file contains the implementation for all the controller/mechanical metrics (see below).
- **plotting_common.py** — Plotting utilities (see below).
- **plot_results.py** — Loading and plotting the simulation results.
- **util folder** — Contains all the utilities for running the simulation (do not modify)
- **logs folder** — Contains all the simulation logs
- **muscle_parameters folder** — Contains the optimized muscle parameters used in the Project (do not modify)
- **models folder** — Contains extra code for the biomechanical and simulation in MuJoCo (do not modify)

Evaluating the performance

In **util.metrics.py**, we provide some basic functions to evaluate the performance of the model simulations, and stored in the dictionary wave_controller.py::WaveController.metrics if the parameter compute_metrics is greater than 0 (check simulation_parameters.py). These metrics are provided to you (you can use them to answer the question, or implement your own). Below we list the metrics and provide a brief description of how these are computed. All metrics are computed after discarting a $sim_fraction$ percentage of transient time point (currently $sim_fraction$ is set to 0.3, i.e. 30% of the total simulation time but you can modify this if needed in **util.metrics.py**). This is the assumed steady state fraction of time points. We distinguish between two different performance metrics:

1. **Controller metrics:** these are metrics related to the controller muscle activation signals $\overrightarrow{M}_{diff} = (M_{diff_i} = M_{L_i} - M_{R_i})_{i=1,...,N_{joints}}$, and assumes that the left and right signals M_{L_i} and M_{R_i} are stored in `wave_controller.py::WaveController().state[:,self.muscle_l]` and `wave_controller.py::WaveController().state[:,self.muscle_r]`, respectively, and converged to a steady state solution. We provide the following metrics:
   - **frequency** - Stored in metrics["frequency"] (value $\in \mathbb{R}$). Computes the average frequency of the vector $\overrightarrow{M}_{diff}$ over the N_{joints} joints from the time difference between zero threshold crossings (threshold crossing method) of each M_{diff_i}.
   - **ptcc** - Stored in metrics["ptcc"] (value $\in [0, 2]$). It's the peak-to-through correlation coefficients based on the difference between the the maximum and the minimum of the autocorrelogram of oscillatory signals. It measures the stability of the signals (value $\sim$ 2=stable oscillations, value $<$ 1.5=unstable oscillations).
   - **ipls** - Stored in metrics["ipls"]. Compute the sum time difference between zero threshold crossings of adjacent muscle M_{diff_i} and $M_{diff_{i+1}}$
   - **wavefrequency** - Stored in metrics["wavefrequency"] (value $\in \mathbb{R}$). Computes the wavefrequency of the muscle difference signals, i.e. metrics["frequency"]*metrics["ipls"].
   - **amplitude** - Stored in metrics["amp"] (value $\in \mathbb{R}$), computes $\max(\overrightarrow{M}_{diff}) - \min(\overrightarrow{M}_{diff})$.
2. **Mechanical metrics:** these are metrics related to the mechanical model. We provide the following metrics:

- **Speed metrics**. We provide two different forward ($\vec{v}_{fwd} \in \mathbb{R}$) and lateral ($\vec{v}_{lat} \in \mathbb{R}$) speed metrics.

  (a) **PCA forward and lateral speeds**. Stored in metrics["fspeed_PCA"] and metrics["lspeed_PCA"] (values $\in \mathbb{R}$), respectively.
  Advantage: compute the instantaneous speeds
  Disadvantage: assumes that the body maintains an elongated position at all times (can produce large errors for large body bendings)
  Explanations: it computes the instantaneous speed of the center of mass ($\vec{v}_{com}(t)$) along the first ($\vec{PC}_1(t)$) and second ($\vec{PC}_2(t)$) principal components of the axial joints' coordinates $X(t) = (x(t), y(t))$ in the x-y dimensions, respectively. These coordinates are called x_{axial} hereafter (total of $N_{njoints} = 15$ axial joints). The projections are computed at each time step and integrated over the simulation to obtain their average values.

$$\vec{PC}_1(t) = argmax_{||a||=1}(\boldsymbol{a}^T Cov(\boldsymbol{X}(t))\boldsymbol{a}) \tag{4}$$

$$\vec{PC}_2(t) = \vec{PC}_1(t) \times \hat{k} \tag{5}$$

$$\vec{x}_{com}(t) = \frac{1}{N_{joints}} \sum_{axial} \vec{x}_{axial}(t) \tag{6}$$

$$\vec{v}_{com}(t) = \frac{1}{dt}(\vec{x}_{com}(t) - \vec{x}_{com}(t - dt)) \tag{7}$$

$$\vec{v}_{fwd} = \frac{1}{T} \sum_{t} < \vec{v}_{com}(t), \vec{PC}_1(t) > dt \tag{8}$$

$$\vec{v}_{lat} = \frac{1}{T} \sum_{t} < \vec{v}_{com}(t), \vec{PC}_2(t) > dt \tag{9}$$

  This methods allows to maintain a notion of positive and negative speed as well as to measure the curvature of the trajectory, two aspects that you will explore during the project.

  (b) **Cycle forward and lateral speeds**. Stored in metrics["fspeed_cycle"] and metrics["lspeed_cycle"], respectively (values $\in \mathbb{R}$).
  Advantage: does not suffer for large body bendings computations
  Disadvantage: compute the speed computed across subsequent cycles (not instantaneous) and assumes oscillatory signals at steady state
  Explanations: Firstly, the algorithm computes the average frequency f of the muscle difference signals M_{diff_i} using the treshold crossing detection method (see controller metrics above). If $f > 0$ (otherwise, the velocities are both zeros) it samples the joints coordinates across cycles and computes the velocity of the displacements of the head coordinates $X_{head}(t) = (x(t), y(t))$ between an initial time instant and a cycle of frequency f before:

$$v(t) = \frac{|X_{head}(t) - X_{head}(t - T)|}{T} \tag{10}$$

Where $T = 1/f \cdot dt$ is the period. The head to tail direction versor $\overrightarrow{ht}$ is then computed from the average angle between the first (head) and the second (neck) links across the considered cycle $\bar{\theta}$ as:

$$ht = (\cos(\bar{\theta}), \sin(\bar{\theta})) \tag{11}$$

The left pointing versor $\overrightarrow{ht}_{left}$ with respect to the $\overrightarrow{ht}$ direction is then computed using the right-hand rule.

Finally, the forward and lateral speeds are respectively computed as the projection of the velocity v along $\overrightarrow{ht}$ and $\overrightarrow{ht_{left}}$.

- **Torque consumption**. Stored in metrics["torque"] (value $\in \mathbb{R}$) and computes the total exerted torques as computed as the integral over time of the sum of all the torques (in absolute values) generated by the joints.

$$T_{tot} = \sum_{t} \sum_{seg} ||\tau_{seg}(t)|| \tag{12}$$

Note that these metrics are to be intended as a reference for the generation of some of the plots but you may implement new metrics that you think could be useful (i.e cost of transport).

Running the simulations

We provided two example files for running the simulations:

- `example_single.py` - in this example you can see how to run a single simulation of the zebrafish (already tries in section). You will explore the GUI, where to store the simulation data (for postprocessing and plotting) and how to use some plotting tools (explained more in the next section). Try to change some parameters of the `siumulation_parameters.py` files to test how it works (try, i.e. to save the video, activate/deactivate the headless mode, etc).
- `example_multiple.py` - in this example you can see how to run multiple simulation (without GUI) in parallel using the processors for different parameter values.

Plotting tools (optional)

We provided some optional plotting tools in `plotting_common.py` (check the documentation for more details):

- `plot_time_histories` - plots time histories of a vector of states on a single plot
- `plot_time_histories_multiple_windows` - plots time histories of a vector of states on multiple subplots
- `plot_left_right` - plotting left and right time histories on separate subplots
- `plot_2d` - plots an array 2d colored plot (uuseful for 2d parameter searches)
- `plot_trajectory` - plots head positions
- `plot_positions` - plots positions of the links

  Examples for how to use these tools can be found in `example_single.py` and `plot_results.py`

Final notes

- Use the exercise files `exercise_X.py`, with X the exercise number to answer the question X, for example, `exercise_0.py` to answer question 0.

- Check the notes on the code. Many explanations are given therein.
- Be concise and precise in the answers in the answers of the exercises.
- You can append videos to explain your reasoning.

Questions

At this point you can now start to work on implementing your exercises.

1. Implement the equations for a sine wave controller

Implement a sine wave swimming controller that generates a periodic traveling wave for the left and right muscle activations (M_{L_i} and M_{R_i}, for $i = 0, ...14$) to actuate the zebrafish according to equation 13.

$$\begin{aligned} M_{L_i}(t) &= 0.5 + \tfrac{A}{2} \cdot \sin\left(2\pi f \cdot \left(t - \epsilon \cdot \tfrac{i}{N_{joints}}\right)\right) \\ M_{L_i}(t) &= 0.5 - \tfrac{A}{2} \cdot \sin\left(2\pi f \cdot \left(t - \epsilon \cdot \tfrac{i}{N_{joints}}\right)\right) \end{aligned} \tag{13}$$

where ϵ is the wavefrequency, A is the amplitude and f is the frequency of the wave. Note that that equation 13 guarantee that the active stiffness component $M_{sum_i} = (M_{L_i} + M_{R_i})$ is equal to 1 at all times, and that the muscle activation difference $M_{diff_i} = (M_{L_i} - M_{R_i})$ has amplitude A.

Test the controller's ability to generate swimming locomotion for fixed values of $\epsilon \in [0, 2]$, $A \in [0, 2]$ and $f \in [1, 5]Hz$ (test different parameter combinations). Show plots of the left and right muscle activations M_{L_i} and M_{R_i}, and of the animal head trajectory in the (x,y) plane. Also, show the evolution of the joint angles. Report the performance metrics in your report. You can also upload a showing that the animal swims correctly.

2. Study the performance of when varying the wavefrequency and amplitude

Run a parameter search of the model when varying the wavefrequency and amplitude of the sine wave controller 13. Use the metrics provided (or your own metrics) to study the performance of the model. What speeds can the model achieve? What is the optimal wavefrequency and amplitude?

3. Implement and study a non-sinusoidal controller

In you implemented a sinusoidal propagating wave to actuate the model. In reality, the left-right muscle activations in a real animal might are less stereotypical than perfect sines. Modify equation 13 to implement a square wave controller, that modifies the signals to make them similar to a square wave. You should implement a gain function similar to the ones used in Lab4, and have a parameter that can control how steep the square wave can be. Test different steepness values and plot the performance of the model.