Bio-Informatics Project

Gilad Freidkin[1] **and Guy Cohen**[1]

[1] *Technion*

Sagi Levy

Contents

1. Introduction

In this project we develop different approaches and control algorithms that will be deployed on a future system being developed, as well as the software framework that will help to develop and assess these algorithms. The system will track and capture *c.elegan* worm neurons without restricting their movement.

The control algorithm will be the 'brains' of the system, receiving real time video stream of the worm, and is tasked with keeping track of the worm and controlling the movement such that the worm neurons will be captured at all times.

1.1. Project Goals

Our main requirements for the framework are:

1. Modular and Expandable - it should be generalizable and adaptable to future requirements as the system being developed
2. Simulation support - since the system is still being developed and many of it's parameters depend on the feasibility and performance of this framework, the framework should be able to simulate the system with different parameters and configurations.
3. Evaluation - the framework should provide accessible and useful tools to evaluate and visualize algorithms on different benchmarks and metrics.

Our most basic requirements for any control algorithm are:

1. Real-Time - the algorithms should be able to run in real time.
2. Reliability - the framework should be predictable as much as possible, with known failure modes and reproducible results.

2. System Overview

2.1. Structure and Main Components

The system mentioned before will be comprised of several components:

1. The arena (also referred to as *platform*) - in which a *c.elegan* worm can move.
2. A camera - The stream from the camera is used as the (only) input for the control algorithm.
3. A microscope - used to capture the neural activity of the worm.
4. Motors - used to adjust the position of the platform so that the worm will be in the field of view (FOV) of the microscope.

Assumptions and structure of the system:

- The arena is made of a clear material (in the desired spectrum) so that the worm is visible to the camera and microscope.
- The centers of the FOVs of the camera and microscope align (as can be seen from Figure 1).
- The entire arena area might not be entirely within the FOV of the camera, meaning that the arena area is larger than the area captured by the camera.
- The camera and microscopes positions are fixed and the arena is moved by the motors.
- A control algorithm receives video footage from the camera, and controls the movement of the platform. This algorithm instructs the motor about position corrections that should be made for the arena.

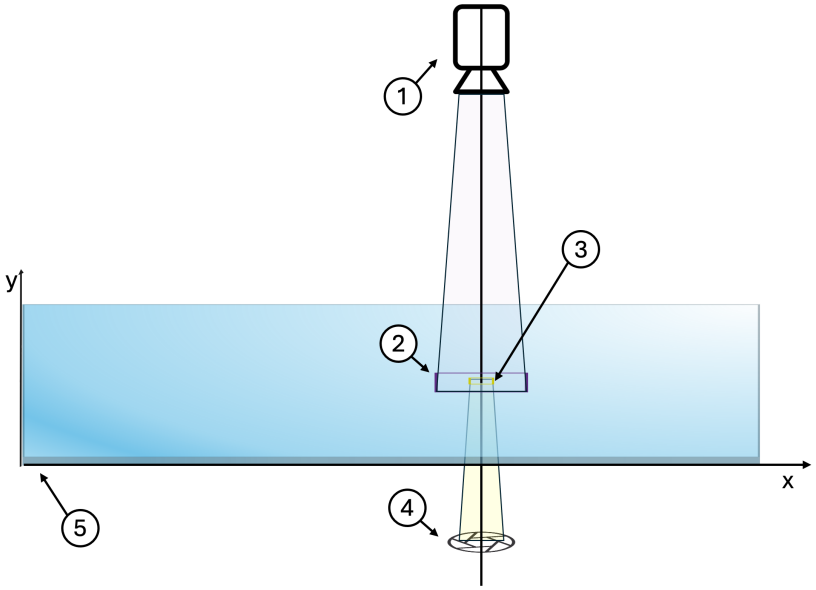

Figure 1. Diagram of the system main components
1. The camera 2. The camera FOV 3. The microscope FOV 4. The microscope 5. The arena

2.2. System Specifications

Here are the ranges of key parameters considered at time of writing:

- arena size - roughly $30mm \times 30mm$
- camera FPS - 60 FPS
- camera pixel size - $\sim 90^2 \frac{pixels}{mm^2}$ or $\sim 0.011^2 \frac{mm^2}{pixel}$ equivalently
- camera FOV - $4mm$
- microscope FOV - $225 - 320\mu m$
  - 60X magnification - $225\mu m$
  - 40X magnification - $320\mu m$

2.3. Control Flow

The system will work in repeating, identical *cycles*. Each *cycle* composed of several *phases* (as shown in Figure 2 below):

1. **Imaging phase** - this phase represents the time during which the microscope captures the neural activity of the worm. During this phase no movement of the arena is allowed.
2. **Prediction phase** - this phase is contained inside the *imaging* phase and represents the time that takes the control algorithm to make a prediction for how to move the arena before the next *imaging* phase begins.
3. **Movement phase** - In this phase the motors move the arena to the position given by the control algorithm at the end of the prediction phase.

Notice that the length of the prediction phase determines what is the last data (frame) that the control algorithm can obtain for it's prediction during the current cycle (during the prediction phase no new inputs are obtained for the control algorithm).

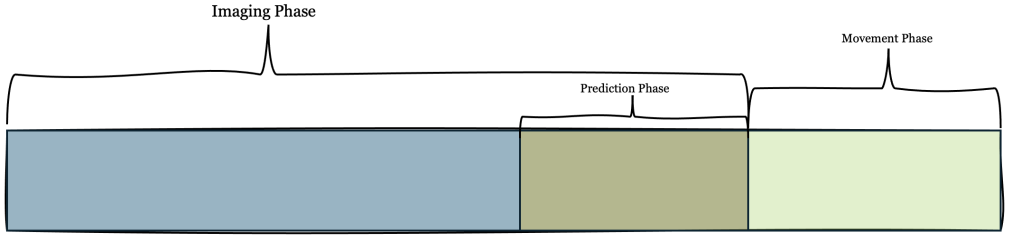

Figure 2. A single Cycle and it's phases

2.4. Control Specifications

Here we present the ranges of key parameters for the control flow considered at time of writing:

- imaging time - 200ms
- prediction time - $10 - 40ms$
- movement time - 50ms

Since the prediction in the case of a microscope FOV of 225μm proved difficult and sometimes impossible (when the worm moves fast and exists the FOV during the imaging phase) we also considered splitting each imaging time into two equal parts of 100ms. This split has no practical effect on the system besides defining the imaging time as 100ms (and the prediction time must be lower than 100ms).

3. Code Overview

This chapter is designed to serve as an overview of the framework we have built. we will go through the framework structure and explaining some design choices, assumptions and logic behind the main parts of the code. Since the framework is large, we will not go over the details of every part or file in it. Instead, we provide all the necessary documentation within the code itself, for specific implementation details one should refer to them. In addition, we created a documentation website which holds the documentation for the entire library.

3.1. Code Structure

The entire codebase is structured hierarchically and is often referred to as a '*package*'. We named our package "wtracker".

This package is divided into several distinct '*modules*', each serving a unique purpose and representing the highest level of abstraction for its respective role. Most of these modules operate independently, each focusing on a specific aspect of the project. Examples for some modules are `sim`, which is used for running system simulations, and `eval`, a module used for evaluating simulation results.

Each module resides in its own folder and consists of multiple '*files*'. Each file is responsible for one task. For instance, modules `sim` and `neural` contain a `config.py` file that manages the configuration settings for various parts of the module.

Usually, each file holds a single '*class*', which implements the functionality that the file is responsible for. In some cases however, several classes are placed within a single file, with their cumulative role being responsible for file's task.

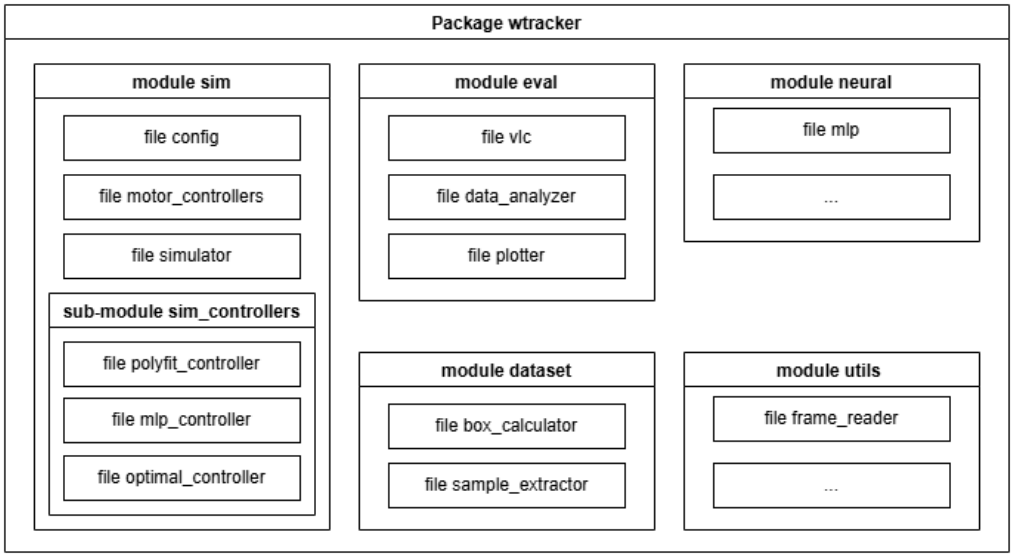

Figure 3. High level overview of the code-structure. Each of the modules is mentioned, along with the main files within each module.

In addition to the wtracker package, there is a folder named "workflows" which includes interactive notebooks (.ipynb files) for each workflow and main task in the project. Each notebook is ready to use with explanations for each step.

3.2. Modules

In this section we elaborate on the main goals of each module, and provide brief descriptions about the main files and classes within each module.

NOTE: usually, the file name and the class name are the same and classes are placed in files with similar names. When talking about specific classes, we will not mention directly in which files they are located if the file name matches the class name.

3.2.1. sim

This module is the heart of the project, and contains all the necessary components to simulate the real-world system. These include the platform and its motor, the camera, a microscope and the algorithm that controls the platform movement. The main components of this module are the following:

- `file config` - The classes implemented in this file contain the simulation configurations. They what makes the simulator modular, and allow to easily change the parameters of the entire system. Inside you will find two main classes:
  - `class TimingConfig` - Defines the timings of the different phases of each cycle as well as POVs of both camera and microscope.
  - `class ExperimentConfig` - Holds all the fixed parameters of a given experiment (video) that are relevant for the simulation/system.
- `module sim_controllers` - This sub-module contains different platform controlling algorithms. Each algorithm is placed in a separate, matching file. Notable ones are:
  - `class OptimalController` - Heuristic based controller predicting the 'best' position for the platform with access to data from the future. Best coordinates are calculated as the median of the centers of all worm-head bounding boxes (bboxes), in the incoming imaging phase.
  - `class PolyfitController` - Predicts worm position by fitting a polynomial to positions of the worm acquired in previous frames, and afterwards uses the polynomial to determine worm's future position.

  - `class MLPController` - Uses a neural network to predict the worm position in the future.
- `sim.motor_controllers` - Classes within this file are responsible for simulating the dynamics of the movement of the platform.
- `class Simulator` - This class encompasses all the components of the system and runs the simulation as a whole. The communication between all the different components is implemented using this class, and they are simulated according to the timing configuration discussed earlier.

3.2.2. eval

The role of this module is to evaluate the performance of different control algorithms, as well as gather general information about worm's behavior during experiments. All the components within this module read a log produced by the simulator and extract the relevant information from it. The main components are:

- `class VLC` - The role of this class is visualization of a simulation run. Given a log, this class visually recreates the run of the simulation, with the predictions and FOVs of the camera and microscope along with the ability to rewind, skip and control the flow of the system as well as save it as a video.
- `class DataAnalyzer` - the responsibility of this class is to take raw simulation log and extract important information from it. As result, a new *analyzed* log file is produced containing a lot of auxiliary information.
- `class Plotter` - The goal of this class is to read a previously analyzed log (as produced by `DataAnalyzer`) and plot the relevant results.

3.2.3. dataset

The role of this module is to extract images that contain the worm from experiments. These images are afterwards used to train a model which detects the head of the worm. The main components within this module are:

- `class BoxCalculator` - Given a collection of raw frames, this class is responsible for identifying the worm position within each frame.
- `class SampleExtractor` - This is the main class of the module. This class uses `BoxCalculator` to detect worm positions at different frames. Afterwards, this class crops the original frame image around the detected position to match a specified fixed size (which was provided by the user).

3.2.4. neural

The contents of this module relate to training the neural network that are used to predict worm head position in the future, based on previous observations. Afterwards, this trained neural network is used by a platform control algorithm for more precise position corrections. This module implements several neural network architectures that can be used along with the dataset creation, training, logging and configuration support. Also, this module defines:

- `class mlp.WormPredictor` - This is a general class that represents a neural network that takes the bounding boxes of past frames and predicts the position of the worm in the future.
- `class mlp.IOConfig` - A class that holds the basic configuration for the `WormPredictor` class. Essentially, the data within this class is:
  - Which past time-stamps the `WormPredictor` samples to make a prediction
  - What frames are predictor's output (how many frames into the future the worm position is predicted).

3.2.5. utils

This module contains generic classes and functions (like basic file operations) that are shared by all components of the project. One class worthy of mentioning is:

- `class FremeReader` - This class is the basic data structure that wraps image sequences and allows iterating over them. All components that expect some sort of image sequence as an input expect it to be in the `FrameReader` format (recall that we work with sequences of frames rather than with videos).

3.3. Inner Workings and Assumptions

Here we will explain some of the inner-workings and design decisions made in the code.

3.3.1. The Dataset Module Explained

The role of the dataset module is to extract images of some pre-defined *fixed* size that contain the worm. These images are afterwards used to train a model which detects the head of the worm. The rationale for module is the following: suppose we have a footage of a worm and in this footage the entire arena is visible. On the other hand, the FOV of the camera in the simulated environment does not cover the entire arena, but only a small portion of it. Using this module we detect the worm position at each frame in the original footage and crop images matching the camera FOV size around the detected worms. Afterwards, a head-detection model will be trained on these extracted images, which, as result, are of the exact image size and magnification that the model receives when running the simulation (remember that the only input for the control algorithm is the camera footage). By training the head-detection model on images similar to the ones it's going to encounter during the simulation, we improve its performance drastically.

There are two important notes regarding this module and the components within it. The components of this module rely on the following assumptions, and would not function correctly if they do not hold:

1. In the original footage of the experiment, the background is stationary and always within the field of view. That is, the position of the camera that captures the experiment footage is constant with regards to the arena.
2. At each frame, a single instance of the worm is visible, no less and no more.

3.3.2. Controller Design

We refer to the platform-control algorithms as *controllers*. There are many ways such algorithms can be implemented, and here we elaborate about the general structure of these algorithms.

First, recall that in the System Overview we already made the design choice that any control algorithm's input is the camera video stream (as frames) and its output will be a position to move to during the next movement phase. The algorithm does not receive, nor expect, any other inputs.

Additionally, we made an algorithm design-choice that the algorithms will have 2 very specific steps. The first step is locating the bounding box of the worms head, and the second is predicting the position to move the platform to. In order to make a prediction in the second step the algorithm uses the output of the first step.

It's important to note that for a given system cycle (see section 2.3), a controller algorithm has access to all the frames up to beginning of the prediction phase . All input frames received after the prediction phase began cannot be used for predicting worm position during the current cycle.

Algorithm 1: General Control Algorithm

Input: Camera Video Stream
1 **Step 1:** Detect bounding box of the worm's head in all relevant frames;
2 **Step 2:** Using the bounding boxes from step 1 calculate the position to move the platform to;
Output: (dx, dy) representing the amount to move in each axis

3.3.3. Detecting the Worm Head

For the detection of the worm head and bounding box around it (area of interest) in a way that fulfills our requirements we laid out in the Project Goals (section 1.1) we decided to use a well known, open source Machine-Learning model called YOLO. This model is developed by ultralytics and is known for its ease of use, high performance and with real-time orientation for object detection and segmentation tasks. This model receives raw images as input and returns detected bounding boxes for each image.

The YOLO family of models has many versions with new ones being developed constantly, we used the latest fully supported version 8 although newer version are often completely compatible and should work out of the box. Each version offer several models in several sizes (with respect to number of parameters) that offer a trade-off between accuracy and speed. Currently, we use YOLOv8s, which is the small-size version of the model, which we found fit for the task. The ultralitics library offers an easy to use framework to train, evaluate and deploy the YOLO models which makes them very easy to use.

3.3.4. CSV Controller

The `CSVController` is our baseline control algorithm. This controller detects the worm position at the last possible moment (right before prediction phase begins), and instructs the platform to move towards that position. The bounding box around worm's head is detected by the YOLO model, the center of this bounding box is calculated, and the platform is instructed to move there. This controller does not make any predictions about the future, it simply uses the last observation of the worm and moves the platform there.

3.3.5. Optimal Controller

The `OptimalController` has the information of all the worm bounding boxes in the next imaging phase. Using this information it computes the center of each and calculate their median (the median over each axis is calculated separately). This median is the position it outputs for the motors to move the platform to. This is a simple heuristic that performs very well in practice.

This controller has access to future information by reading a simulation log file (produced by a previous run with another controller) which contains all the bounding box detections for all the frames. By simply accessing the log file at the lines corresponding to the future imaging cycle, this algorithm gets all the data it needs.

We use this heuristic to estimate the optimal performance that any algorithm can possibly achieve. We use the evaluation results of this controller to understand the feasibility of different simulation configurations, to understand for which configurations a good enough [1] algorithm *may exist* in the first place. In addition, by comparing the evaluation results of this controller to others, one can understand how close to the optimum the other control algorithms are.

3.3.6. PolyFit Controller

The `PolyfitController` is using past frames (more precisely the bounding boxes of these frames) to fit a polynomial of degree n through their centers, and using this polynomial it then predict where the worm will be in some future frame. This controller fits two different polynomials, one for the x coordinates and another for the y coordinates. We explain how this controller works for the x coordinates, for y it's similar.

Suppose the Polyfit controller is at some cycle of the simulation. Let $t_1, t_2, ..., t_m$ be the times from which the polyfit controller sample worm positions to make a future prediction (these times are relative to the current cycle). For every such time-stamp t_i, there is a matching center-of-bounding-box coordinate (x_i, y_i), which is provided by the YOLO model. As result, $\{(t_1, x_1), (t_2, x_2), ..., (t_m, x_m)\}$ are the past samples that are used to fit the polynomial. The Polyfit controller finds a polynomial f of degree n such that:

$$f = \arg\min_{\tilde{f} \in Poly(n)} \sum_i (\tilde{f}(t_i) - x_i)^2$$

A weighted version of the above equation, where each time stamp t_i is associated with a weight w_i is formulated as follows:

$$f = \arg\min_{\tilde{f} \in Poly(n)} \sum_i w_i (\tilde{f}(t_i) - x_i)^2$$

[1] by "good enough" algorithm we mean an algorithm with error rate lower than some pre-defined value by the user

There exists a closed formula for the solution of the above minimization problem, with this problem formulated as solving a weighted least-squares system. Therefore, finding such polynomial is very efficient and costs practically no time. Afterwards, to make a prediction about the x coordinate at some future time-stamp t_{fut}, it calculates the following:

$$x_{fut} = f(t_{fut})$$

This controller has multiple parameters. First, the sampling times $t_1, ..., t_m$ are a very important parameter, since the future prediction is performed based on observations from these time-stamps (these times can be specified as offsets, with '0' being the frame at the beginning of the current cycle). Additionally, the degree n of the fitted polynomial has a massive impact on the resulting predictions. Furthermore, the weighting factors w_i which we mentioned previously also impacts the results.

In order to find the optimal parameters for this controller, we created a `WeightEvaluator` class which we use to evaluate the performance of this controller, given a weights vector $(w_1, ..., w_m)$ and a polynomial degree n. By utilizing an optimization algorithm that seeks to minimize this evaluation error, we find the best weights and the best polynomial degree to for our algorithm (more precisely approximations of the best). We use the mealpy [3] [1] [2] library which provides many different optimization algorithms to find the optimal weights, as discussed above.

3.3.7. MLP Controller

The MLP (Multi-Layered Perceptron) is a platform controller, that predicts worm's future positions using a `WormPredictor` neural-network. WormPredictor is the wrapper class for any neural network that accepts several past frames' bboxes as input and predicts the center of the worm head in a future frame(s). The sampled past frames indices, and the future frame for which the prediction is performed is stored in `IOConfig` class.

Similarly to the Polyfit Controller, this controller samples worm positions at past time-stamps $t_1, t_2, ..., t_m$ (again, relative to the current cycle). With each such time-stamp t_i a bounding box (x_i, y_i, w_i, h_i) is associated. This box bounds the worm head at frame of time t_i, and was provided by the YOLO model. Using these input bounding boxes $\{(x_1, y_1, w_1, h_1), ..., (x_m, y_m, w_m, h_m)\}$ the network makes a prediction about future worm position (x_f, y_f) where f denotes some time-stamp in the future.

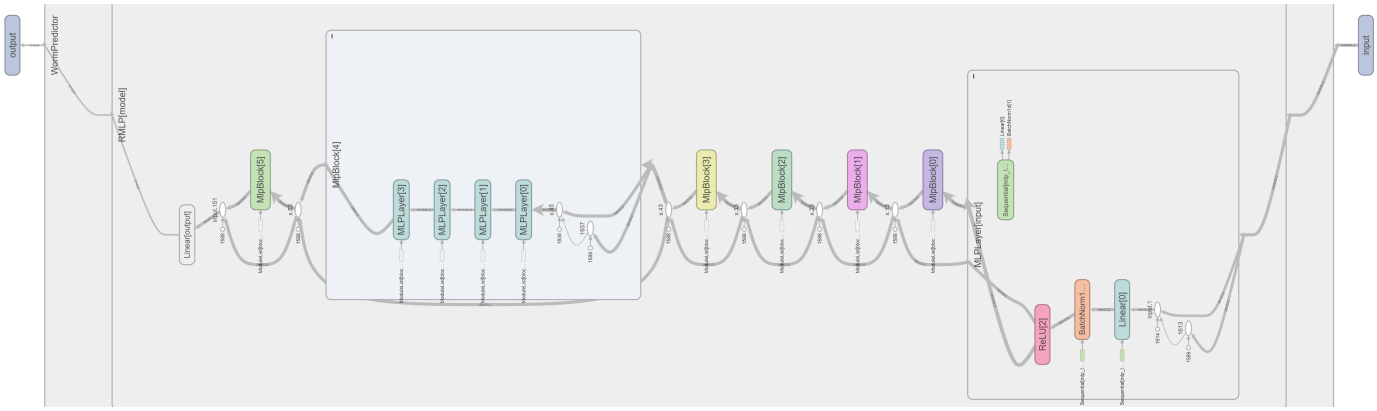

Figure 4. The structure of a `WormPredictor` neural-network we used for the MLP controller. Each `MLPBlock` consists of multiple `MLPLayers` and each `MLPLayer` consists of Linear, BatchNorm and Activation layers. Notice the residual connections between different `MLPBlocks`.

3.3.8. Simulation

The simulation of the system is done by the `Simulator` class found within the the `sim` module. The simulator operates on videos [2] of experiments where the entire arena is visible and the camera is fixed. In addition, the simulator is able to read a log file produced by a previous simulator run, to get worm head positions from the log and run entire system simulation without needing access to the original footage. That's because we only use the video footage of the experiment to detect worm head positions, but these positions can be read directly from the log.

Using the video of an experiment, the configurations of the system and of the experiment, a motor controller and a platform controller algorithm, the `Simulator` logic works as follows:

Algorithm 2: Simulator pseudo-code

Input: (System configuration, Motor controller, Control algorithm, Experiment video / log)
for *each frame of the experiment* **do**
  Determine the current phase of the cycle;
  Pass current frame (cropped to the camera FOV) to the control algorithm;
  if *prediction phase starts during the current frame* **then**
    Request a prediction from the control algorithm;
  if *prediction phase ended during the last frame* **then**
    Read prediction from the control algorithm;
    Pass the prediction to the motor controller;
  if *currently in movement phase* **then**
    Update the platform position using the motor controller

The Simulator is event based, meaning that when an even happens (for example prediction phase ends) then it calls the function that handles that that event.In practice, there are more events than are shown in the pseudo-code above.

[2]The videos are saved as individual images of each frame. More precisely, the simulator expects a `FrameReader` object that wraps the frame sequence of the video.

Since the Simulator is working in discrete time (frames) while the system will work in continuous time, we associate each frame with the time interval that it took place in real time. All actions and state changes will happen at the frame with the corresponding time interval. For logging and visualising purposes, we look at the system state as it will be at the start of the time interval of the current frame, before any changes are made. This means that logs of frame 0 will hold the system state at 0 ms, and logs of a frame that starts at $t = 100$ ms will hold the systems state at 100 ms.

3.4. Workflow

Here we will go over the steps to do some of the main tasks, from training a YOLO model on custom data to running simulations with different configurations. All of the main Workflows have a dedicated, interactive notebook (.ipynb file) ready to use with explanations for each step. All of the workflow notebooks are located in a dedicated folder called "workflows".

We begin by providing a short description of every notebook file, with more details presented later. The general relation between the notebooks, and how to they integrate one with another is described in section 3.4.1 and onwards. In addition, the general flow-chart, depicting all the relations between the notebooks can be seen in figure 5.

`create_yolo_images` - Prepares raw frames of some experiment for the process of training YOLO model on them. This step entails detecting the worm in selected frames and cropping a region of pre-defined size around the worms.

`yolo_training` - Used to train a YOLO model on a given dataset. The training dataset was prepared by annotating [3] the images which were extracted using the notebook `create_yolo_images`. The annotation process can be done with RoboFlow, which is an online dataset creation and annotation tool. For more details see section 3.4.2.

`initialize_experiment` - In order to run system simulations on a *new* experiment, first it's essential to initialize the experiment. The initialization step runs the YOLO predictor on the raw experiment, detects worm's head position in each frame and saves the detection results into a log. That log would be later used for simulating different control algorithms on the experiment. In addition, the background image and worm images are extracted from the raw frames. These can be used later during analysis, to calculate the segmentation based error. This log is useful since in the future the simulator can simply read worm head positions from the log, instead of using YOLO to predict worm's head position in every frame of interest (which is much slower, especially on computers without a dedicated graphics card).

`simulate` - Run a full system simulation on some previously *initialized* experiment. The simulation is ran by reading an experiment log produced by the initialization process - in each frame, worm's head position is retrieved from the log. In this notebook it is possible to simulate the system with any controller and any configuration parameters, not only the ones of used for the initial experiment log. Similar to the initialization process, the simulation produces a log, which would be later used to analyze system's performance and its behavior.

`analysis` - This notebook is used to analyze the performance of a control algorithm (controller). A log which was produced by running `simulate` is read and analyzed, and different plots and statistics are presented. In addition, there is an option to calculate segmentation evaluation-error, by counting how many pixels of the worm are outside of the microscope view. To this end, we use the background and worm images which were extracted during the run of `intialize_experiment` notebook for this experiment.

`visualize` - Given a system log which was produced by `simulate`, this notebook is able to visually recreate the simulator's behavior. At each frame, the position of worm's head is drawn, the position of the microscope FOV, and also the camera FOV. This notebook is used to visually assess the performance and the behavior of the simulator, and to visually investigate what causes the system to misbehave.

`predictor_training` - Used to train a specific simulation control algorithm. The `MLPController` is an algorithm that uses a neural network (NN) to predict worm's future position. Since this algorithm is NN based, it requires training. That script is responsible to train that NN from experiment log files, which were produced by either running `initialize` or `simulate` (doesn't matter).

`polyfit_optimizer` - This notebook is used to tune the parameters of a specific simulation control algorithm. The `PolyfitController` is an algorithm that uses polynomial-fitting to predict worm's future position. A polynomial is fitted from past observations at previous time stamps, and afterwards sampled in the future time to predict worm's position. This notebook is used to determine the optimal degree of the fitted polynomial, and to find the optimal weight of each past sample for the fitting process.

[3] Annotation is the process of labeling each image. For this specific task, labeling means manually placing a bounding box around worm's head for each of the dataset images. The neural-network model learns from the annotated data and learns to do it automatically.

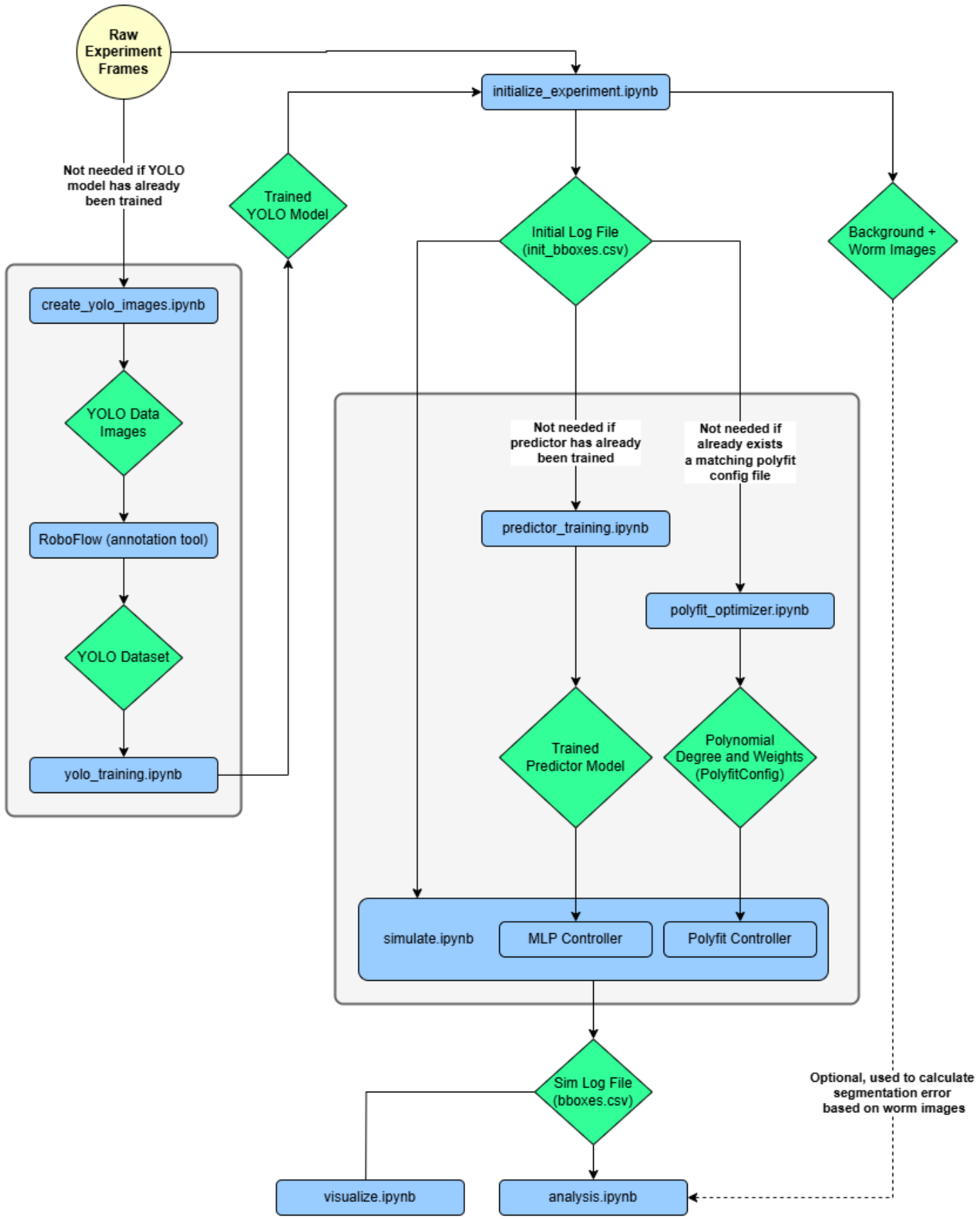

Figure 5. Workflow outline and the dependencies between each notebook files. Blue color (rectangles) denotes an interactive notebook file, green color (diamond) denotes intermediate outputs between different files, and the global input is in yellow color (circle). Dotted line denote optional dependencies.

3.4.1. Conducting an Experiment

Here we explain how to properly capture the footage of an experiment for the simulator.

1. Decide on the frame rate (FPS) of the experiment. There are two distinct scenarios:
   (a) If the sole reason for the experiment footage is to be used for YOLO training, a very low FPS can be used (can be as low as 1 FPS or even lower if possible). Ideally, a single frame would be captured every few seconds.
   (b) If a simulation to be run on the experiment footage, a high FPS should be used, preferably at least 60 FPS.
   Note, that the the chosen frame rate should be the same frame rate on which the platform control algorithms were calibrated.
2. The camera should be completely stationary from which the entire arena is visible.
3. The footage should be captured as distinct images for each frame, not as a continious video. We recommend to use 'bmp' image format, which is lossless, and is able to save single channel if the image is grayscale.
4. Make sure that the distinct frames are saved as images with the frame number appearing in their name, such that it's possible to read them in-order.
5. If you want to run a system simulation on the experiment, follow the steps in the `initialize_experiment` notebook.

3.4.2. YOLO Model Training

Below is the workflow to train a YOLO model to detect worm's head position:

1. Conduct a new experiment and capture the footage, as explained in section 3.4.1.
2. Determine the image size for the YOLO model, this size should match the desired input image size during a simulation run.
   (a) At the time of writing, it is possible to pass images of different sizes to YOLO models, but they are scaled to the closes multiple of 32. This means you should use and train YOLO models on images with sizes that are a multiple of 32 when possible.
   (b) A YOLO model should be trained on images with the same size, it is not expected to work well on images of different sizes without special attention.
   (c) Be careful of a Distribution Shift, this means that the training data is different (not representative) of the real world. For example:
      i. In the training data, are the worms always in a similar position?
      ii. Is the background lighting consistent with the one on the system?
      iii. Is the size of each pixel the same as in the system?
      iv. Are the worms in the dataset representative of the worm population?
3. Create a set of images for annotation - Follow the instructions in the `create_yolo_images` python notebook. Make sure to provide the correct image size, which was determined in the previous step.
4. Annotate the data - The annotation process is the process of marking the actual worm head position in the extracted images. To do so, we recommend using the website Roboflow, which provides easy-to-use tools to annotate the data, and create a dataset for YOLO models.
5. Create a YOLO dataset - If you used Roboflow to create the dataset - on the dataset page you can click on 'export dataset' and then 'show download code'. You can copy the code snippet to the appropriate place in the notebook of step 6 to download the dataset to the computer and use it for training.
6. Follow the instructions in the `yolo_training` notebook and train the YOLO model.

There are two approaches to tackle the challenges mentioned in step 2 (c). The first approach is to carefully train the YOLO model on very similar conditions as of the final system. The resulting model will function well, but if conditions change then models performance will likely degrade.

The other approach is to train the model on wide variety of settings (e.g. different lighting conditions or different magnification levels), leading to a more robust model. The benefit of this approach is that the model is more robust to changes, but a disadvantage is that such models usually require more data, and may perform slightly worse than models carefully trained on some very specific conditions.

3.4.3. Perform System Simulation

Below is the workflow of performing a full system simulation on some experiment, and analyzing the results.

1. If the experiment was not initialized yet, make sure to follow the instructions in the `initialize_experiment` notebook.
2. Decide on the platform control algorithm to be used.
   - If `MLPController` algorithm is chosen: the MLP controller works by a neural network that predicts future positions of the worm. If that network needs training then first run `predictor_training` notebook. Note, that the neural network should be trained only once. Once the network is trained, there is no need to perform this step anymore.
   - If `PolyfitController` algorithm is chosen, and the hyper-parameters of the controller should be tuned then first run the `polyfit_optimizer` notebook. Note, that the hyper-parameters of this controller should be tuned only once. Once they were tuned, there is no need to perform this step anymore.
3. Follow the steps in the `simulate` notebook. The result of running this notebook is a log file containing the full simulation log.
4. To visualize the simulation run `visualize` notebook, and to analyze the performance of the control algorithm, and general statistics of the conducted experiment run the `analyze` notebook. Both of these notebooks analyze the log produced by `simulate`.

3.5. Useful Resources

In this section we provide some useful links to some resources used.

- WTracker GitHub repository (houses all the code of the project) - Code Repository
- Official WTracker Documentation website - WTracker Documentation
- YOLO model used for predicting worm head positions - YOLO Docs
- RoboFlow, the annotation tool used to create the dataset YOLO model was trained on - RoboFlow
- C-Elegan dataset we created to train the YOLO model - C-Elegan Head Dataset

4. Evaluation

Remember, that the goal of the system is to keep worm's head within the microscope view during the imaging phase. Therefore, given a platform control algorithm, we want to evaluate how well it performs this task. There are several different ways to evaluate the error of the system.

We define the error metrics on each frame separately. To define the error per imaging phase one might take the average or the maximum error during the relevant imaging period. We believe that calculating imaging phase error as the maximum of all errors during that phase is preferable, and indeed that's how we measure it.

4.1. Distance Metric

The first, and most intuitive way perhaps is to calculate the distance between the center of the micrososcope field of view and the center of worm's head. Given that the center of worm's head is at (x_1, y_1) and the center of the microscope FOV is at (x_2, y_2), the distance-based error is calculated as:

$$Error = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

4.2. IOU-Based

The main disadvantage of the distance-based error is that even if worm's head is completely within the FOV of the microscope, but their centers are not aligned, then this frame gets penalized. Since what matters is that worm's head is within microscopes FOV, regardless whether the head is centered or not, we define another error which is aimed to resolve this issue.

This error is based on the IOU metric, which measures the proportional area of intersection between two sets. Our error measures the proportional area of the bounding box region around worm's head, which is outside of microscopes field of view.

Let H be the set of points which is within the region of the bounding box detected around worm's head. Let M be the set of points which are within the microscope field of view. The error is calculated as follows:

$$Error = 1 - \frac{|H \cap M|}{|H|}$$

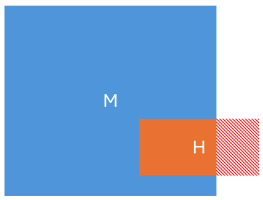

Figure 6. IOU-Based Error depiction. The blue region (M) depicts microscope FOV, the orange region (H) depicts the bounding box around worm's head, and the sub-region of H that has diagonal lines depicts the error area. The error is calculated as the area of diagonal lines divided by the total area of H.

4.3. Segmentation-Based

The disadvantage of the IOU-based error is that it's calculated solely based on the bounding box around worm's head, instead of directly measuring the exact worm's head area outside of the camera FOV. Figure 7 depicts the problem arising from that, and explains the motivation for a segmentation based error. The segmentation based error solves this issue directly, by detecting the exact worm pixels within the head bounding box, and measuring the exact worm area outside of the microscopes FOV. The disadvantage of this loss compared to the previous two losses, is that calculating it requires direct access to each frame image, in order to extract worm's segmentation, while the two previous losses can be calculated directly from the log file produced by a simulation. As result, the calculation of the segmentation-based loss takes considerably longer.

Let H be the set of points which is within region of the bounding box detected around worm's head. Let M be the set of points which are within the microscope field of view. Let W be the set of segmented worm points. The segmentation error is measured as follows:

$$Error = 1 - \frac{|W \cap H \cap M|}{|W \cap H|}$$

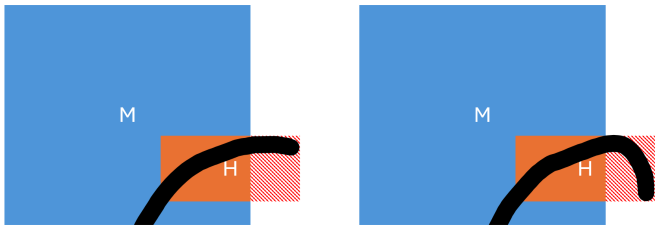

Figure 7. Depiction of the problem with IOU-based error. The IOU error of the left and the right figures are the same, even though the proportional worm area outside of microscope FOV is completely different. On the other hand, the segmentation based error measures the proportion of worm's area outside of the FOV, resulting in the loss of the right figure to be higher, which is indeed correct.

5. Results and Analysis

5.1. Setup

In this section we elaborate on the exact setup of the experiments we conducted.

5.1.1. Experiment Configuration

We conducted 5 different experiments in total, all with similar camera configurations. The camera setup for each of these experiments is the following:

- Camera model: Blackfly S BFS-U3-244S8M
- Camera footage captured in grayscale
- Binning: 2x (both vertical and horizontal)
- Camera distance from the arena such that 90 px / mm
- Exposure Time: 10000 us
- Camera FOV cropped to capture only a single arena
- FPS: 60
- Final resolution of about 1600 x 1400 px
- Back-light brightness was dialled until the maximal image histogram value was about 254. All other lights in the room were off.
- All experiment frames saved in 'bmp' image format, which is lossless (no quality loss).

5.1.2. Simulation Configuration

Each of the experiments were evaluated on 4 different simulator configurations. These configurations differ both in the length of the imaging phase, and in the size of the microscope FOV. Here we present a table summing up these configurations:

	config 1	config 2	config 3	config 4
Imaging Time (ms)	200	100	100	200
Prediction Time (ms)	40	40	40	40
Movement Time (ms)	50	50	50	50
Camera Size (mm)	4	4	4	4
Microscope Size (mm)	0.32	0.32	0.22	0.22
Microscope Magnif.	40	40	60	60

Table 1. System configurations tested

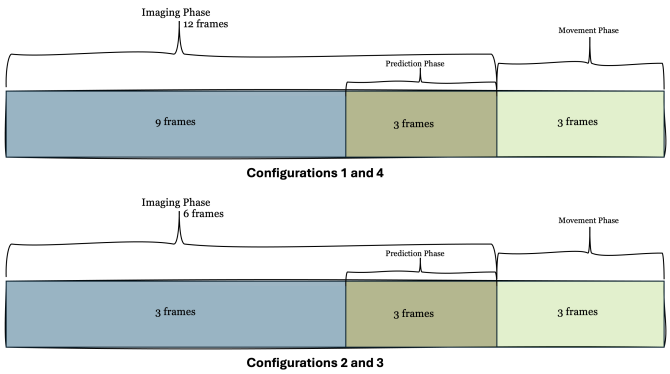

Figure 8. The number of frames in each phase for each configuration.

5.2. Worm Behavior Analysis

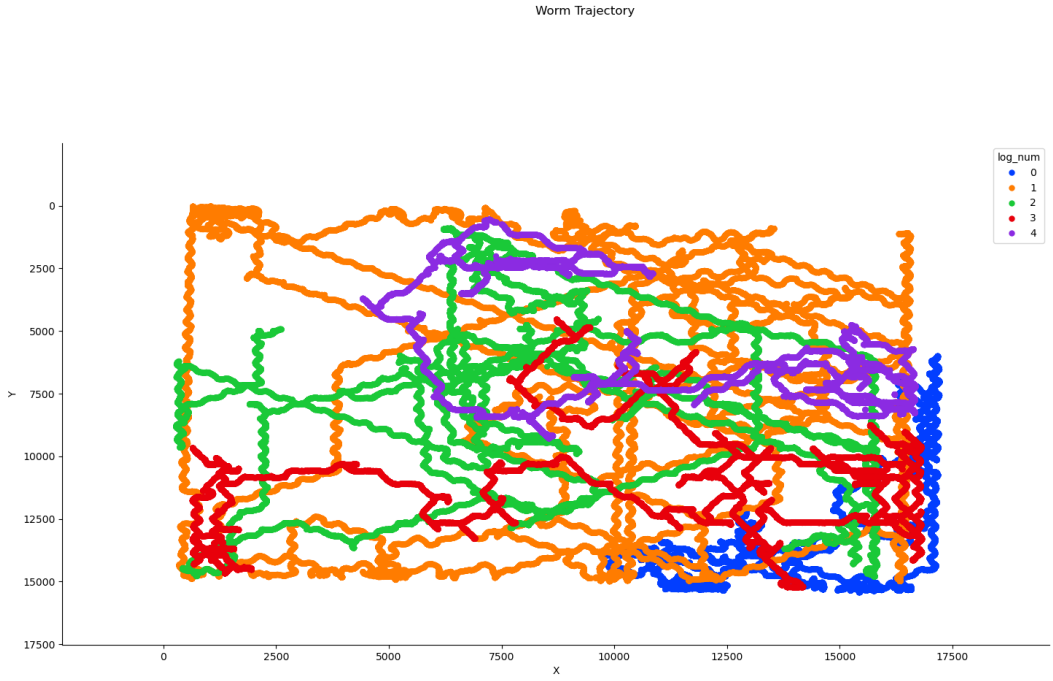

Figure 9. Worm trajectories of each experiment.

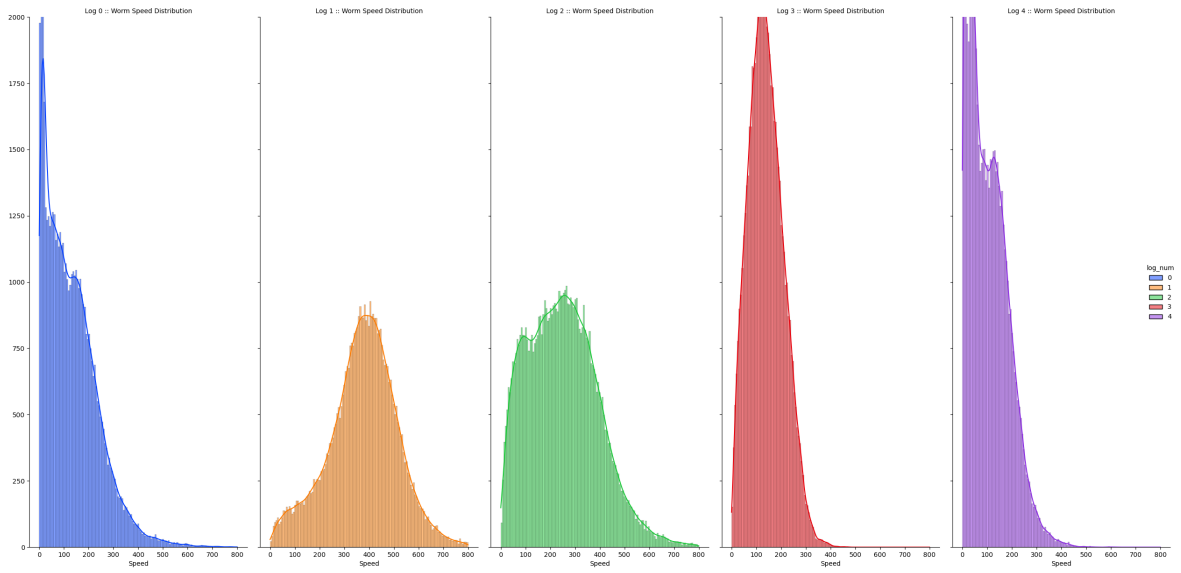

Figure 10. Worm speed ($\mu m/s$) histogram per experiment.

From figure 10 we can see that the speed of the worms varied drastically between the different experiments. Most noticeably, in experiments 1 and 2 (we denote the first experiment as 0 to match the plots), the average speed is much higher than in other experiments. For experiment 3 and 4 the average speed of the worm was less than 150 $\mu m/s$, which is rather slow. Because the worm speed in these last two experiments is so unusually low, we decided to discard them when analyzing different controller algorithms.

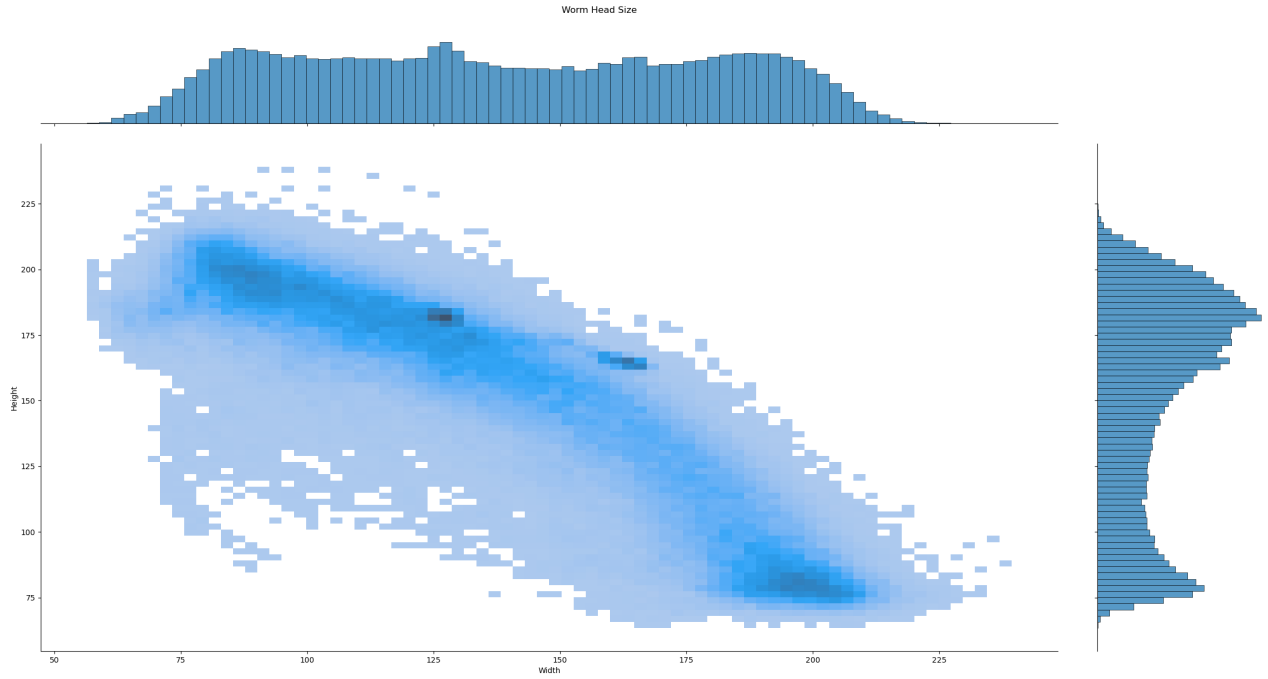

Figure 11. Worm's head size bounding box distribution across all experiments. The predicted bounding boxes are tight, meaning it represents the actual head size of the worm quite accurately.

Figure 11 depicts the head size distribution of the worm among the first three experiments. From this plot we can see that the head width and height often reach up to 210 μm, which is very close to the whole FOV size of the microscope at 60x magnification, hinting that it's perhaps impossible to have no errors in the 60x setup.

5.3. YOLO Analysis

5.3.1. Training

We have trained a YOLO model on data from experiments 0,1 and 2. To create the dataset we have extracted a total of 3,200 uniformly spaced images with resolution of (384, 384) [4] from the experiments (~1,000 per experiment). After manually annotating each image, we have created 2 additional annotated images (for each original image we annotated) by applying a random operation like rotating by 90°, changing the brightness or flipping horizontally/vertically. The resulting dataset has 7,700 images, 87% of which were used for training, 8% for validation and 4% for testing (during the training process additional operations like mentioned above were used as well (on the fly)).

Many different YOLO models and versions are available, we have chosen to train a YOLOv8s model. This is the 'small' model of the 8-th version of YOLO which was the latest at time of training. This model is the 2-nd smallest model, although it was able to achieve very good results comparable with the next model in size (medium) and therefore was chosen as it is faster. Comparison of the different models can be found in this page.

5.3.2. Performance

The model was trained on a Tesla T4 GPU, and achieved a mean inference speed of $\sim 5.5\ ms$. While running on an AMD 5900X CPU each inference took around 20-30 ms. Both numbers are within the range of the prediction time used in all timing configurations, Although it should be noted that these numbers are only for the YOLO model.

The speed of the entire software stack is harder to measure and very much dependant on the computer configuration (on all levels, from hardware choices to OS parameters) and should be measured on a computer specifically built for the system. Although, we have a high confidence that using the right computer configuration and software optimizations it would be possible to achieve prediction times lower than 40 ms from camera to output for the motors.

5.3.3. Failures

Looking at the YOLO predictions on all experiments we have observed several things:

- YOLO was able to correctly detect the worm head the vast majority of the time.
- When detecting the worm head (or tail) the bounding box was precise practically all of the time, even on experiments 3 and 4 which it was not trained on.
- Around 0.788% of frames either have a wrong prediction or no prediction at all. Most of the time these failures happen in several sequential frames.
- Failures can be split into 2 main categories:
  - No prediction - when the model didn't detect a worm head in the image. All the cases on this failure were when the worm was stationary or very slow. Furthermore, they usually happened when the worm was in a circle, in some position that the model has rarely seen or when the worm stuck its head in the channels in the edge of the arena. Some examples are shown in Figure 13.

[4] 384px is close to the camera size ($4mm$) used along with the $\sim 90 \frac{px}{mm}$ pixel sizes of all experiments. Furthermore, it is a multiple of 32 which is ideal for the YOLO model.

Since in almost of these cases the worm speed is very slow, the model was able to detect the worm before it escaped the camera FOV.

– Miss-prediction - when the model detects something else as the worm's head. This mostly happened on the experiments the model was not trained on (the worms there are visibly different). These mis-detections mostly happened when the worm was in a circle, a behavior rarely seen in the first three experiments. In all cases, the wrong predictions we saw were the tail of the worm instead of the head. Some examples are shown in Figure 12.

With this information we believe that the errors happened mainly on cases that the model has not seen or that has a low representation in the dataset (including worms of different sizes), and expect that when trained on a better dataset there will be a very low error rate.

Therefore, we would recommend training YOLO on many (perhaps 10+) experiments in order to get a dataset that represents the real worm population distribution with all the edge/rare cases. With that said we believe that when taking images from many experiments there is no need for a lot of images from each one, and expect that even 200 (or even less) from each would suffice.

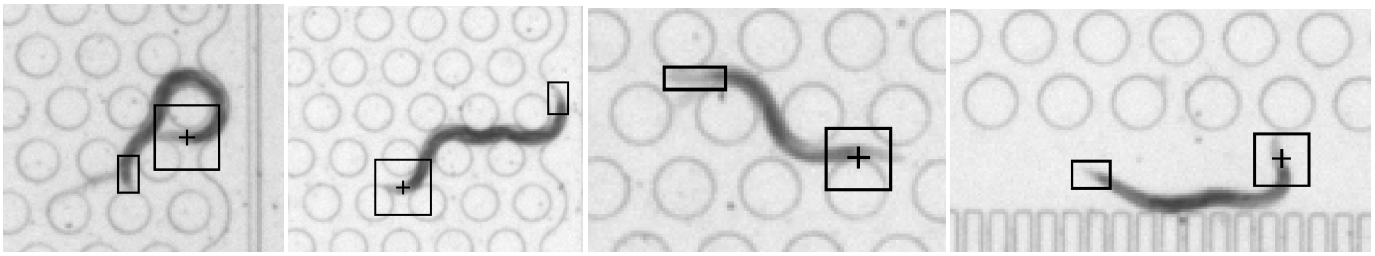

Figure 12. miss-detection of the worm head. In all experiments there were 21 intervals where a miss detection happened (each interval includes some frames with no prediction and good predictions) with total number of 650 frames (out of 300,000+ frames of all experiments).

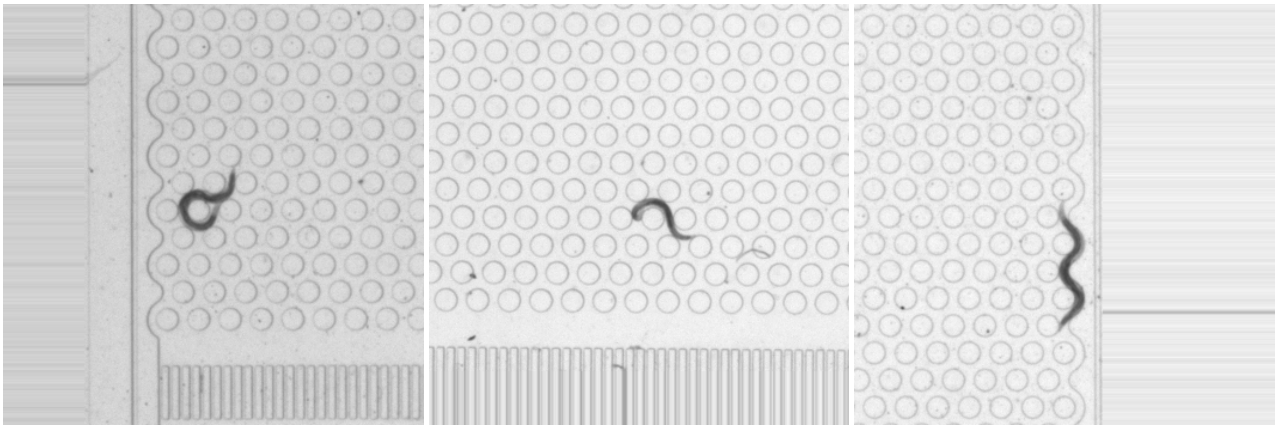

Figure 13. frames where the model didn't detect a worm head

5.4. Controllers Analysis

Here we will present and analyze the results of the different controllers. Recall that in section 3.3.2 we presented a general 2-step pseudo code for all control algorithms. The YOLO Controller discussed above is step 1 of the generic algorithm (detect the worm), and now we will focus on algorithms for step 2 - predict the worm position in the next cycle and move accordingly. This means that all controllers in this section take the worm bboxes (bounding boxes) produced by YOLO as input and not the camera images.

We have trained two versions of `WormPredictor` (also referred to as *ResMLP*, and used by MLPController) and *PolyFit*, one for timing configs 1,4 (with 200 *ms* imaging time) and another for configs 2,3 (100 *ms* imaging time). Both versions were trained/optimized on data from experiment 1.

5.4.1. Evaluation Data

Here we explain which data was used to evaluate different controller algorithms. First of all, as can be seen by plot 10, the speed of the worms in experiments 3 and 4 was unusually slow. As result, we decided to discard these experiments from the evaluation, and the evaluation was done only on experiments 0, 1 and 2.

Additionally, we removed all worm data when the worm was out of the legal bounds of the arena. That means that all frames when the worm was near the channels at the edges of the arena were removed. Also, since the goal of the controllers is to keep the worm in the microscope view *only* during the imaging phase, we removed all frames which do not belong to this phase. Also, frames when the YOLO model made no predictions, miss-predictions and all affected cycles were removed as well, since we're interested to benchmark the Controller performance, and not the performance of YOLO.

It's important to note that both the `MLPController` and the `PolyfitController` were tuned *only* on data from experiment 1, while the evaluation was done on 3 total experiments. Since the tuning (training) was done only on a single experiment, one may expect more robust in results in the future if the controllers to be tuned on more experiments.

5.4.2. Performance

During the simulations with the PolyFit and WormPredictor controllers, the simulator was able to constantly process more than 2000 and 500 cycles per second respectively. Besides the controllers themselves, this includes the logging, simulation logic and other code all running in the background, this means that the controllers themselves run faster than 2 *ms* and probably much faster. Also, the WormPredictor was run on CPU which is usually much slower than a GPU, running both the YOLO and WormPredictor on a GPU should result in very high speeds of prediction.

5.4.3. Cycle-Step Errors

In this chapter we present different plots of error as function of "cycle-step". The *cycle-step* notion refers to the frame number within a cycle. For example, if the system timing configuration is such that imaging phase takes 12 frames, and moving takes 3 frames, then frames 0-11 of cycle-step match the imaging phase, and 12-14 the moving phase.

The error bar at the *n*-th frame of the cycle-step is the error at the *n*-th frame of the cycle, aggregated over all cycles of all evaluation experiments. These plots help in understanding how the error changes during the cycle.

Config 1 (0.32 mm micro size; 200 ms imaging):

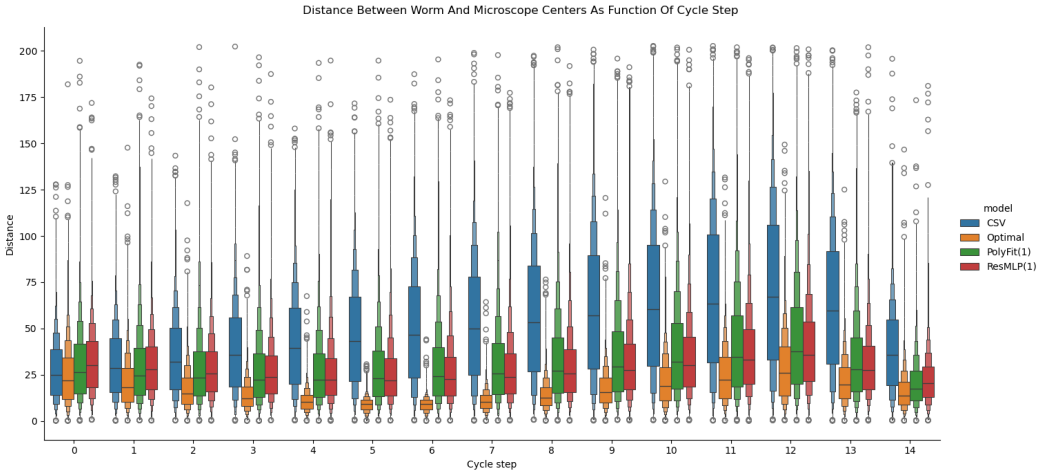

Figure 14. Deviation as function of cycle step of all controllers ran on experiments 0,1 and 2 with timing configuration 1. Frames 0-11 match the imaging phase, and 12-14 the moving phase.

Config 2 (0.32 mm micro size; 100 ms imaging):

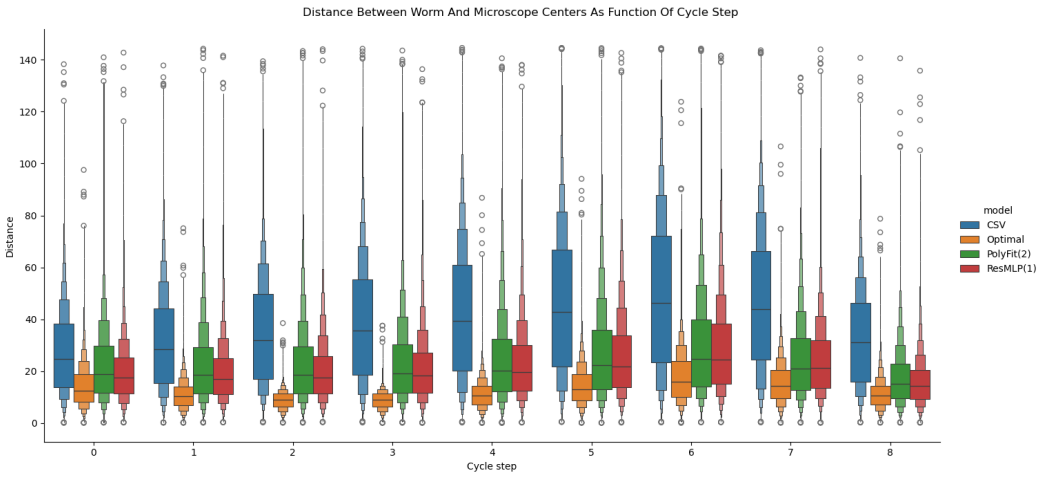

Figure 15. Deviation as function of cycle step of all controllers ran on experiments 0,1 and 2 with timing configuration 2. Frames 0-5 match the imaging phase, and 6-8 the moving phase.

Config 3 (0.22 mm micro size; 100 ms imaging):

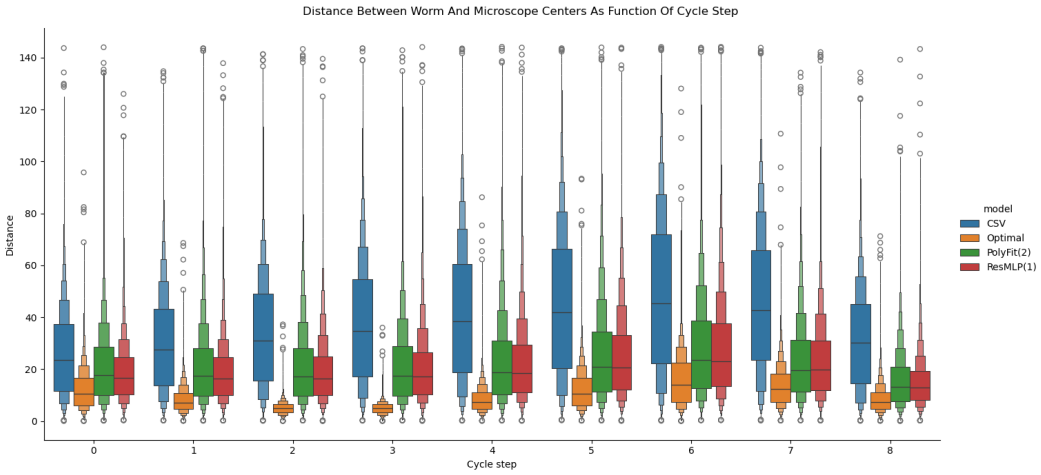

Figure 16. Deviation as function of cycle step of all controllers ran on experiments 0,1 and 2 with timing configuration 3. Frames 0-5 match the imaging phase, and 6-8 the moving phase.

Config 4 (0.22 mm micro size; 200 ms imaging):

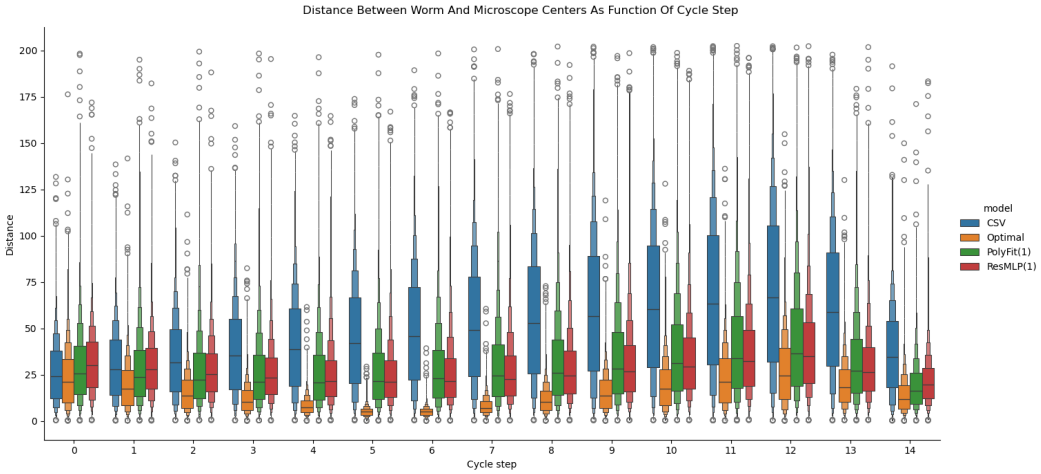

Figure 17. Deviation as function of cycle step of all controllers ran on experiments 0,1 and 2 with timing configuration 4. Frames 0-11 match the imaging phase, and 12-14 the moving phase.

5.4.4. Error Quantiles

In this chapter we present different error quantiles, aggregated over experiments 0, 1 and 2. Values at row q% represent the q-th quantile of the data, which can be achieved by sorting the elements and taking the values of the exact element located at the index matching the q-th quantile. We present various quantiles of the "bbox_error" (IOU-based error) and "worm_deviation" (distance-based error) for different controllers on each experiment configuration.

In the table below containing the results of configuration 1, we can see that only the most basic CSVController suffers a significant bbox error, all the other controllers manage to maintain the worm inside the microscope FOV for the entire time.

Config 1 (0.32 mm micro size; 200 ms imaging)									
	wrm_speed	bbox_error				worm_deviation			
controller		CSV	PolyFit	MLP	Optimal	CSV	PolyFit	MLP	Optimal
mean	245.99	0.024621117	0.005256336	0.003973866	0.000101265	48.16	32.43	30.91	16.18
std	164.16	0.071881746	0.041163754	0.042501621	0.003926081	34.30	47.08	33.87	11.70
min	0.29	0	0	0	0	0.08	0.06	0.08	0.06
25%	109.66	0	0	0	0	20.72	14.52	15.88	8.36
50%	229.53	0	0	0	0	41.59	25.81	26.00	12.65
75%	362.24	0	0	0	0	68.38	42.69	39.47	21.14
80%	391.89	0	0	0	0	75.86	47.71	43.32	24.01
85%	424.30	0.035817	0	0	0	84.75	53.90	48.21	27.62
90%	465.96	0.094546	0	0	0	96.15	62.51	55.21	32.34
95%	525.69	0.17885	0	0	0	112.35	77.29	67.64	39.70
max	4083.33	1	1	1	0.61691	702.88	3529.46	1430.25	181.98

Using configuration 2, we can see that splitting the imaging time in half further improves the results to the point where even CSVController has a very low bbox error rate.

Config 2 (0.32 mm micro size; 100 ms imaging)									
	wrm_speed	bbox_error				worm_deviation			
controller		CSV	PolyFit	MLP	Optimal	CSV	PolyFit	MLP	Optimal
mean	245.99	0.005902724	0.001144292	0.001364378	4.13042E-06	36.84	23.88	22.08	11.42
std	164.16	0.030258013	0.019735751	0.028842918	0.000568007	24.23	23.46	24.64	6.56
min	0.29	0	0	0	0	0.06	0.02	0.04	0.02
25%	109.66	0	0	0	0	17.17	11.80	11.82	7.07
50%	229.53	0	0	0	0	32.66	19.56	18.48	10.41
75%	362.24	0	0	0	0	52.07	31.14	27.68	14.20
80%	391.89	0	0	0	0	57.03	34.77	30.61	15.44
85%	424.30	0	0	0	0	62.73	39.18	34.32	17.16
90%	465.96	0	0	0	0	70.14	45.42	39.28	19.77
95%	525.69	0.035137	0	0	0	81.37	55.92	48.73	23.84
max	4083.33	1	1	1	0.11208	294.93	2066.97	1325.86	97.56

When using the 100 *ms* imaging time as well as the smaller 0.22 *mm* microscope FOV (config 3), we can see that only the OptimalController manages to maintain a low bbox error. We can see that MLP performs similarly to PolyFit and both are significantly better the the naive CSVController.

With that said, the OptimalController manages to get significanly better bbox errors than both MLP and PolyFit controllers. That hints that there might be possible improvements made to the MLP and Polyfit controllers, such that they would achieve lower error.

Config 3 (0.22 mm micro size; 100 ms imaging)									
	wrm_speed	bbox_error				worm_deviation			
controller		CSV	PolyFit	MLP	Optimal	CSV	PolyFit	MLP	Optimal
mean	245.99	0.084611571	0.02364741	0.021038123	0.002929844	35.72	22.57	21.26	8.50
std	164.16	0.111324248	0.063378471	0.060182385	0.012807105	24.66	23.50	24.92	6.32
min	0.29	0	0	0	0	0.04	0.11	0.02	0.02
25%	109.66	0	0	0	0	15.61	10.17	10.52	4.30
50%	229.53	0.030985	0	0	0	31.77	18.17	17.40	6.63
75%	362.24	0.14615	0.0122325	0.0074125	0	51.34	29.82	27.02	10.92
80%	391.89	0.174554	0.02933	0.02369	0	56.32	33.29	29.94	12.51
85%	424.30	0.206301	0.0513455	0.04381	0	62.19	37.73	33.66	14.46
90%	465.96	0.245871	0.081537	0.07225	0	69.53	44.11	39.06	17.08
95%	525.69	0.30613	0.1339	0.122047	0.02179	80.79	54.76	48.67	21.14
max	4083.33	1	1	1	0.35235	295.82	2074.66	1333.29	95.77

Finally with configuration 4 we can see the same trends, where MLP and PolyFit perform similarly, and while significantly better than CSVController they are also significantly worse than OptimalController. Though, there is one thing different in this configuration, this time the optimal controller has a high bbox error as well.

Config 4 (0.22 mm micro size; 200 ms imaging)									
	wrm_speed	bbox_error				worm_deviation			
controller		CSV	PolyFit	MLP	Optimal	CSV	PolyFit	MLP	Optimal
mean	245.90	0.139671874	0.05248736	0.046243934	0.01364601	47.21	31.37	30.16	13.96
std	163.92	0.16909911	0.106713209	0.093176151	0.036315372	34.75	47.13	34.06	11.97
min	0.29	0	0	0	0	0.17	0.05	0.13	0.02
25%	109.49	0	0	0	0	19.44	13.29	15.18	5.33
50%	230.02	0.07705	0	0	0	40.65	24.86	25.25	9.70
75%	362.00	0.231215	0.063475	0.06047	0	67.76	41.68	38.49	19.47
80%	391.02	0.27343	0.08992	0.08175	0.007	75.29	46.60	42.43	22.39
85%	423.67	0.32388	0.123629	0.108199	0.027739	84.31	52.92	47.30	26.06
90%	465.15	0.387886	0.171514	0.144566	0.054112	95.58	61.66	54.22	30.89
95%	525.62	0.482886	0.258476	0.210599	0.093913	112.20	76.31	66.92	38.19
max	4083.33	1	1	1	0.86757	710.10	3521.77	1437.73	176.29

5.5. Conclusions

Given the results above, we believe there are several conclusions that can be made. First and foremost, since the worm head-size is very close to the size of the 60x microscope FOV (0.22 mm), there is practically no way to get no errors while trying keep the worm head within the microscope FOV. Splitting the imaging time into two 100 ms phases seems to help, but still the results are not error free.

Another conclusion we can make is if using the 40x microscope (0.32 mm lens size), then there is no need to split the imaging phase into two 100-ms phases, since even with imaging phase of 200ms practically all algorithms, even the most naive ones, achieve zero-error performance.

Talking about the controller algorithms specifically, we can see that both the PolyFit and the MLP controllers significantly outperform the naive CSV controller. The benefit of these algorithms specifically shines when the tracking is performed in tough environments, such as when using a very small microscope FOV (0.22 mm). As result, if there are plans to use a small microscope lens of 0.22 mm we recommend to use MLP controller, but if the lens is larger then the PolyFit controller might suffice. Even though the PolyFit controller manages to achieve impressive error rates, the MLP controller performs better in all scenarios, something that can be clearly seen from the 'worm_deviation' column in all experiment scenarios.

Regarding the YOLO model, for more robustness we recommend to train the model on a wide variety of experiments. From each experiment there should be taken about 100-200 different images for training. By training on many experiments the head-detection model will become more robust, even though it's already pretty robust given that it was trained on only 3 experiments.

5.6. Future Work and Road-map

Since this is a large system with many components we ere not able to implement all the things necessary for it. Below is a list of features and improvements that should be added in preparation for the real system:

- Due to time constraints we did not fully implement segmentation-based evaluation error (discussed in section 4.3). There are several minor code-tweaks remaining to fully implement this feature, which will be out shortly.
- Implement a controller class for the real system - the need for a new class only for the real system mostly arises from optimization and compatibility (such as communication with camera and motor drivers) concerns.
- Design a computer setup for the project - this includes choosing the right hardware and configuring all the settings in a way that is optimal for the controllers/system. This way it is also possible to configure it once and clone it to a computer on another system without going though the installation and configuration steps all over again.
- Implement more advance neural networks - since the neural network seems to work quite well with only a single experiment for training, more advance and current architectures should be explored. For this goal, developing more training/logging and benchmarking tools is also advised.
- Implement multi-experiment PolyFit optimization - currently the weight optimization is working for a single experiment, generalizing for multiple experiments should be trivial.
- Improve the `VLC` player (briefly described in section 3.2.2) - many quality of life improvements can be added to this player in addition to those already implemented. For example, jumping to frames with no predictions, adding various statistics such as worm speed, bbox error and even user defined metrics.

6. Creative Outlet

In accordance with our centuries long tradition, each great work requires an accompanying great song. With words by Claude and ChatGPT, blood, sweat and tears are by us, we present our greatest masterpiece:

In a lab with minds so keen,
A project like none has ever seen.
To track a worm, oh so small,
Neural networks take the call.

With cameras poised, the stage is set,
Microscopes zoom in, don't forget.
Algorithms dance in real-time haste,
To keep neurons perfectly traced.

Oh, the irony, how grand it is,
Neurons to track neurons, what a biz!
A brain of code to watch and learn,
As the worm's neurons twist and turn.

In fields of view, both big and small,
The platform moves, avoids a fall.
Simulations run, predictions made,
In this bio-tech charade.

The framework was modular, ready to grow,
Adapting to changes and tweaks on the go.
Evaluation tools, oh what a sight,
Making sure every algorithm was tight.

So here's to the code that made it all hum,
And to our worm, the star of the scrum.
With algorithms dancing, they led the advance,
In the tiny, twitchy world of C. elegans.

References

[1] A. N. Ahmed, T. Van Lam, N. D. Hung, N. Van Thieu, O. Kisi, and A. El-Shafie, "A comprehensive comparison of recent developed meta-heuristic algorithms for streamflow time series forecasting problem," *Applied Soft Computing*, vol. 105, p. 107 282, 2021. DOI: 10.1016/j.asoc.2021.107282.

[2] N. Van Thieu, S. D. Barma, T. Van Lam, O. Kisi, and A. Mahesha, "Groundwater level modeling using augmented artificial ecosystem optimization," *Journal of Hydrology*, vol. 617, p. 129 034, 2023. DOI: https://doi.org/10.1016/j.jhydrol.2022.129034.

[3] N. Van Thieu and S. Mirjalili, "Mealpy: An open-source library for latest meta-heuristic algorithms in python," *Journal of Systems Architecture*, 2023. DOI: 10.1016/j.sysarc.2023.102871.