ECE 410 Progress report
Modelling C. Elegans Motion in CoppeliaSim with a Neural Network

Dmitrii Fotin

Introduction

Recent advancements in the field of neural network mapping in living organisms allows some insight into how such organisms perceive and react to external stimuli. The more intelligent an animal is considered, the more complex their neural networks are, gathering data from an increasing number of nerves to compile as complete an image of an external object as possible.
Complex behaviours in mammals operate data from a high number of nervous endings, which can be roughly divided into major senses. Modelling a nervous system of this scale is a challenge. Instead, this report focuses on exploring how to model behaviour of a less complex animal, a worm Caenorhabditis elegans, as an introduction to modular modelling of nervous subsystems, which can be extrapolated and developed further to describe and simulate behaviour of more advanced organisms.
C. elegans is able to perceive its environment through touch, smell, taste, light and temperature. Each nervous system can be further localised to specific areas of the worm to map and model in detail. The primary area of interest for this project is modelling touch perception of the worm, which accounts for over 10% of the 302 neurons C. elegans possesses. This was chosen as the main goal of this project to clearly determine the scope of research and simulation and is not intended to fully model C. elegans behaviour.
The report outlines the major steps to set up a simulation of the worm in CoppeliaSim simulation software using Python to program a neural network for the model to learn to move and react to touch stimuli.

Simulation Design

Creating a Model in CoppeliaSim

CoppeliaSim is a robot simulation platform that uses physics engines to prototype designs in real life conditions. An educational licence is available for free on CoppeliaSim website.

CoppeliaSim offers a wide range of tools to design and program a robot with a specific purpose in mind.
For this project, the initial design is a Breitenberg vehicle that has two proximity sensors pointing at 45 degrees to the left and to the right simulating the “head” of the worm. It has two wheels that rotate in both directions to turn, reverse and avoid collisions.
In order to design a Breitenberg vehicle in CoppeliSim, we need to add objects to the simulation plane by right-clicking anywhere on the plane, selecting “Add” -> “Primitive Shape” and selecting the shape as displayed in **Image 1** below.

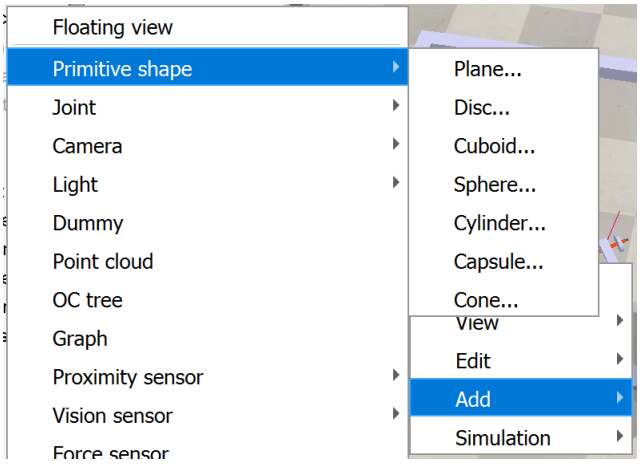


Image 1: Adding a shape in CoppeliaSim
We added a cuboid and set its initial dimension to create the main body of a Breitenberg vehicle.
Next, we need to add a joint, that would connect to the wheels and allow them to rotate by right-clicking anywhere on the plane, selecting “Add” -> “Joint” -> “Revolute” as per **Image 2** below.

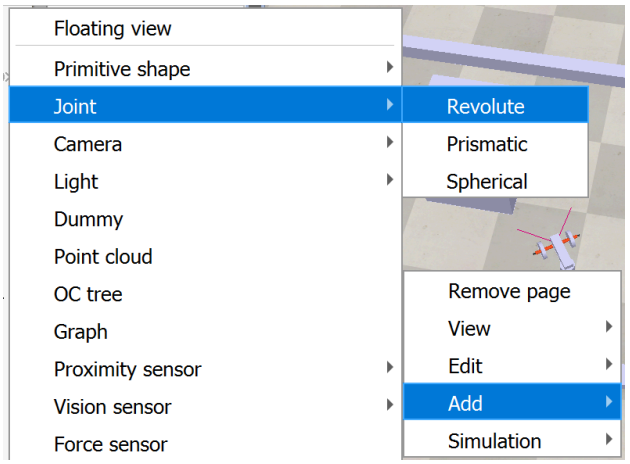


Image 2: Adding a joint in CoppeliSim
Lastly, we add two more primitive shapes for the wheels following the same steps as for the initial body.
All elements then need to be repositioned in the XYZ space to construct the Breitenberg vehicle of the desired configuration using the “Object/item shift” and “Object/item rotate” options in the toolbar at the top of the screen as shown in Image 3 below.

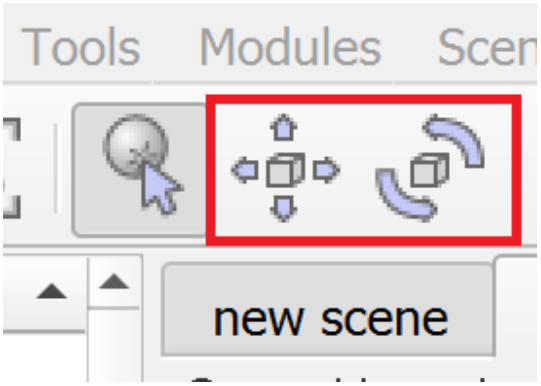


Image 3: Tools in CoppeliaSim to reposition and rotate shapes

Once all shapes and joints are placed where needed, all components need to be selected in the Scene hierarchy on the left and grouped together into one body by right-clicking on the selected items in the menu, selecting “Edit” -> “Shape grouping/merging” -> “group”.

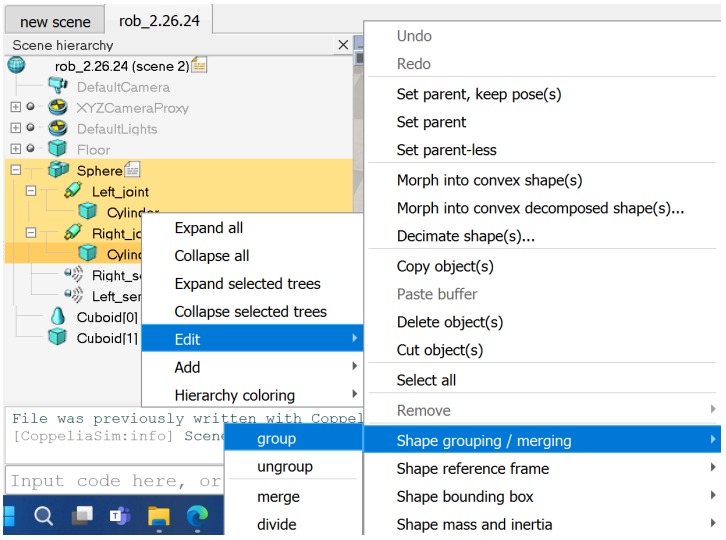


Image 4: Grouping shapes and joints into one body

Setting Up Python on Windows

Before running Python scripts in CoppeliaSim, we need to install the latest version of Python. Then we need to install the zmq and cbor modules vie the command prompt. Open the command prompt, navigate to the folder where our python.exe is and install the modules following the example in Image 5 below.

```
C:\Users\fotin>py -m pip install cbor pyzmq
Collecting cbor
  Downloading cbor-1.0.0.tar.gz (20 kB)
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Installing backend dependencies ... done
  Preparing metadata (pyproject.toml) ... done
Requirement already satisfied: pyzmq in c:\users\foti
)
Building wheels for collected packages: cbor
  Building wheel for cbor (pyproject.toml) ... done
  Created wheel for cbor: filename=cbor-1.0.0-py3-non
1954b914d11707663a6f5cc53
  Stored in directory: c:\users\fotin\appdata\local\p
76e30ff1
Successfully built cbor
Installing collected packages: cbor
Successfully installed cbor-1.0.0
```

Image 5: Installing cbor and pyzmq modules

Once the modules are installed, we need to check where Python is installed on our computer. It is usually located in the path C:\Users\USERNAME\AppData\Local\Programs\Python\Python312\python.exe This is an example path and the actual path might vary depending on the user and python version installed. Once the path of python.exe is record, we need to navigate to C:\Users\USERNAME\AppData\Roaming\CoppeliaSim, open the file **usrset.txt** and change the value for **defaultPython** to the python.exe path we recorded earlier. It's important to note that all backslashes need to be changed to forward slashes as well.

```
additionalLuaPath =  // e.g. d:/myLuaRoutines
additionalPythonPath =  // e.g. d:/myPythonRoutines
defaultPython = c:/Users/fotin/AppData/Local/Programs/Python/Python312/python.exe
```

Image 6: Setting default Python interpreter to the installed python.exe

Once these steps are performed, scripts for specific components in CoppeliaSim can be written in Python.

Adding Custom Programming Scripts

Once the initial model is created, it is possible to write custom behaviour scripts by right-clicking on the main body in the menu on the left, selecting "Add" -> "Associated child script" -> "Non threaded" -> "Python".

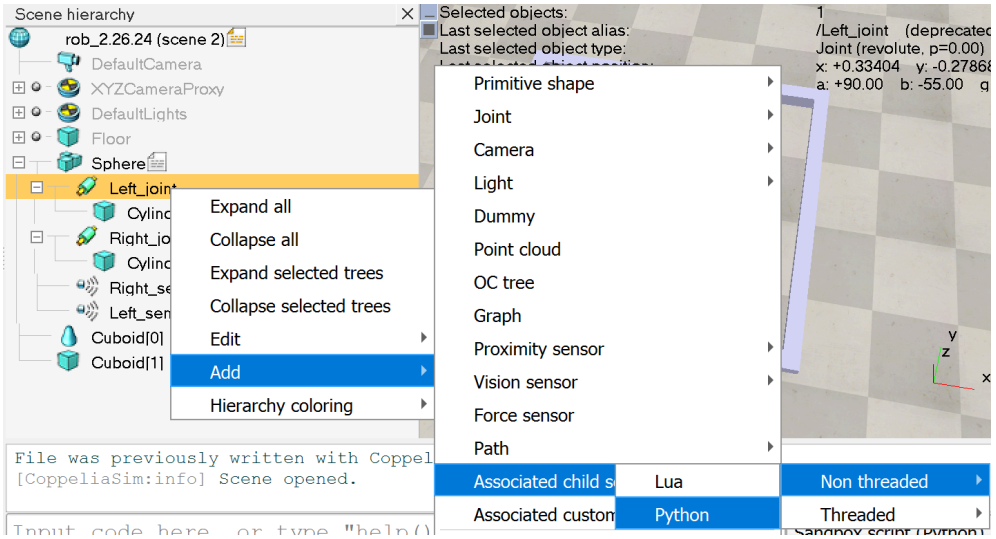


Image 7: Adding a custom script to the model in CoppeliaSim
This would open the script file where Python code can be written.

Addressing Joints and Sensors with Python Scripts

Each component in XYZ space has a handle, which allows us to set specific parameters for that component. In the case of joints, it can be the joint rotation speed. In the case of sensors, we can obtain data from them.
In order to get a component handle, we need to use the function “sim.getObjectHandle” or “sim.getObject” with an argument of the respective component name as we named it in the menu.
In the example below, in line 6 we request a handle for the component “”**Left_joint**”, as it’s named in our simulation on the left, and assign that handle to the variable **leftJointHandle**.

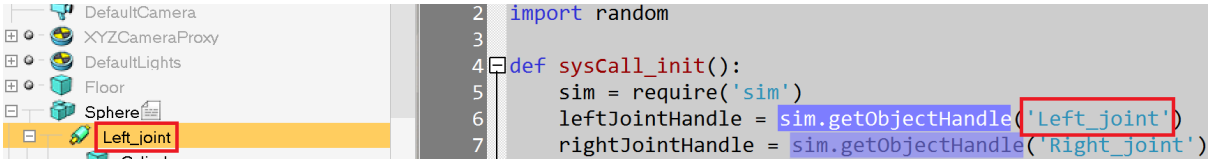


Image 8: Obtaining a handle for a simulation component

Interacting with Joints and Sensors with Python Scripts

Once we obtain the component address, or “handle”, we can then request data from or assign parameters to this component. Each parameter has its own set of functions that can be researched in more detail in CoppeliaSim User Manual.
For the sensors, we can use the function **sim.checkProximitySensor**(sensor_handle,sim.handle_all) with arguments that contain the handle for the sensor as assigned earlier in the script and what data we want to receive from the sensor, where sim.handle_all passes all data. In our examples below we assign all outputs from this function to 5 variables in total:

- First value: 0 (obstacle not detected) or 1 (obstacle detected)
- Second value: distance from the sensor origin to the detected point. None/nil if result is different from 1
- Third value: position of the detected point relative to the sensor origin. None/nil if result is different from 1
- Fourth value: handle of detected object. None/nil if result is different from 1
- Fifth value: normal vector of the surface where the point was detected. Normalized. Relative to the sensor reference frame. None/nil if result is different from 1

For our purposes at this stage, only the distance from the sensor origin to the detected object is used to generate a “shy” Breitenberg vehicle that moves away faster from an object the closer it detects it.
The maximum distance to the object is determined by the range we assign to the sensor in its volume parameters as shown in Image 9 below.

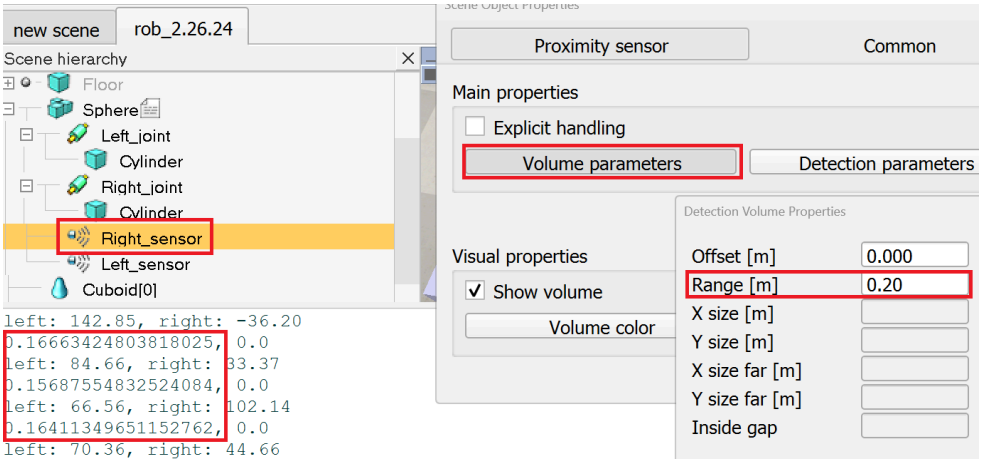


Image 9: Proximity sensor maximum detection range

Knowing the maximum range value, we can map our desired velocity fluctuations to the distance from to the object, using fuzzy logic to increase robot's velocity to move away from another object as it finds itself closer to one.

Once we assign the desired velocity for each wheel, we can pass the it back to the wheels by using the

Next we want to be able to assign target angular velocity to the robot wheels and request proximity range data from the sensors.

Wheel velocity can be assigned to any user-made variable and can be positive or negative for clockwise and counterclockwise rotation. Once the value is assigned, the variable needs to be written back to the simulation component using the function **sim.setJointTargetVelocity**(wheel_handle, wheel_velocity).

```
leftJointHandle = sim.getObjectHandle('Left_joint')
rightJointHandle = sim.getObjectHandle('Right_joint')
leftSensorHandle = sim.getObjectHandle('Left_sensor')
rightSensorHandle = sim.getObjectHandle('Right_sensor')
rl,dl,dPl,obl, nl=sim.checkProximitySensor(leftSensorHandle,sim.handle_all)
rr,dr,dPr,obr, nr=sim.checkProximitySensor(rightSensorHandle,sim.handle_all)
print(dl, dr)
if rl and rr:
    rightTargetVelocity = 1.0 * random.uniform(0.0, 15.0)
    leftTargetVelocity = -1.0 * random.uniform(0.0, 15.0)
    #dir = dir * -1
elif rr:
    rightTargetVelocity = -dr*50.0 * random.uniform(1.0, 15.0)
    leftTargetVelocity = -dr*50.0 * random.uniform(1.0, 15.0)
elif rl:
    rightTargetVelocity = dl*50.0 * random.uniform(1.0, 15.0)
    leftTargetVelocity = dl*50.0 * random.uniform(1.0, 15.0)
else:
    rightTargetVelocity = -10.0 * random.uniform(1.0, 15.0)
    leftTargetVelocity = 10.0 * random.uniform(1.0, 15.0)
sim.setJointTargetVelocity(leftJointHandle, leftTargetVelocity)
sim.setJointTargetVelocity(rightJointHandle, rightTargetVelocity)
```

Image 9: Assigning wheel angular velocity (red) linearly related to proximity to another object (orange)

The initial simulation has a linear relation between proximity to another object and the wheel velocity to avoid it, but it's possible to implement smoother transitions depending on the angle at which another object is detected by using cone-type sensors as opposed to ray-type sensors.

Setting up a C. Elegans Model in CoppeliaSim

After learning how to model a Braitenberg vehicle in CoppeliaSim, it is time to advance to creating a worm model. The overall steps are the same: we need to create solid bodies that are connected to each other with joints, which give the model a certain range of movement. It is best to start with a simplified model and add on to it over time.
For this project, the C. Elegans model went through three stages of evolution, as the structure was becoming more complex due to results obtained in testing.
The first version of the model focused on introducing revolute joints on both sides of each major link of the body, prismatic joints between each two links and two proximity sensors on the “head” of the worm.

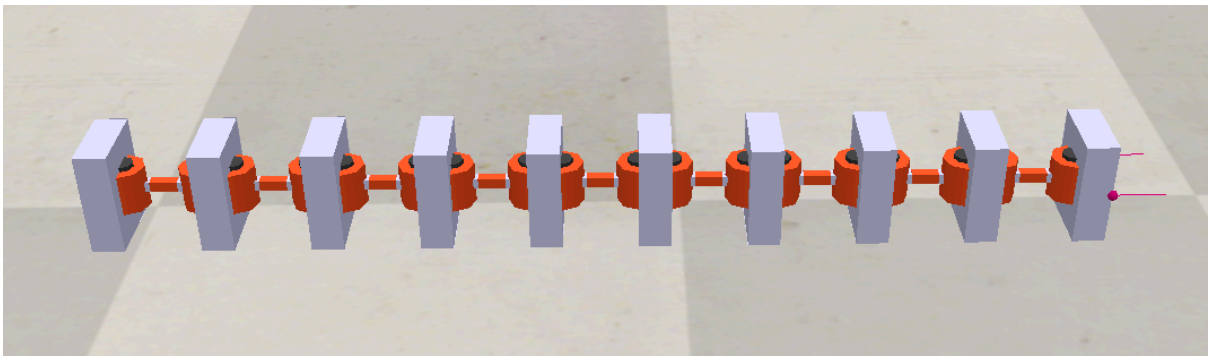


Image 10: C. Elegans model version 1

Version 2 added proximity sensors on the sides of each link and on the back and incorporated a wider base for each link.

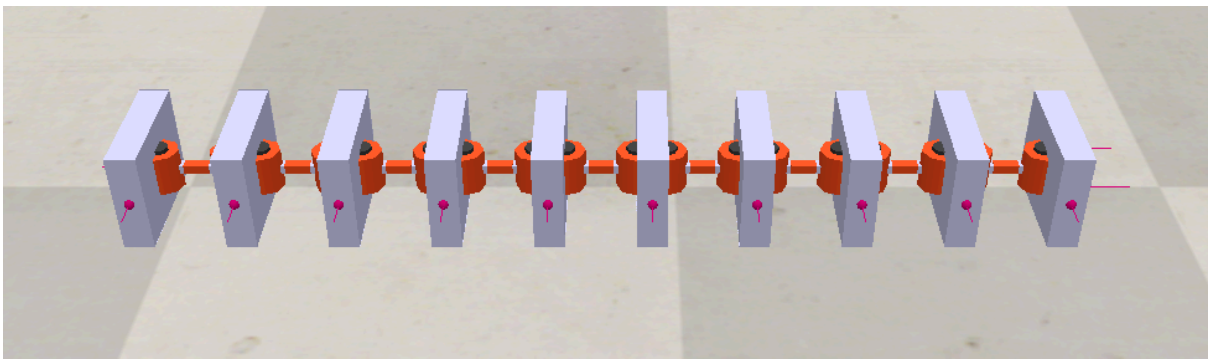
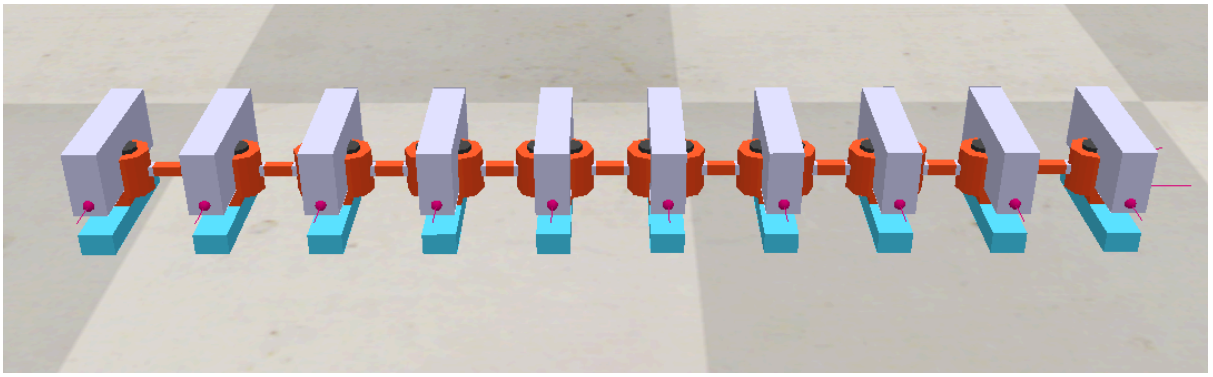


Image 11: C. Elegans model version 2

Version 3 broke out the major links into two three separate components for each link: the top component that connects to other links, another prismatic joint at the bottom of each top component and a “foot” component that goes up and down to simulate worm circular muscles that allow it to decrease friction of the links that need to move. This addition to the design essentially allows the worm to take steps.



Images 12: C. Elegans model version 3

These models can be implemented following the steps outlined in the Creating a Model in CoppeliaSim section of this report.

The latest version of the model incorporates 37 actuators: 18 revolute joints and 19 prismatic joints, as well as 23 proximity sensors.
The revolute joints have a range of 22.5 degrees. As there are two revolute joints per bend, the maximum angle at which each bend can be is 45 degrees. The prismatic joints have a range of 0.02 m. These values and can and should be adjusted with model evolution as more data becomes available about model behavior with a neural network.

How Neural Networks Work

A neural network uses data collected from initially random actions to maximise the reward it gets. Reward is calculated based on factors chosen as desired behavior. For example, a navigation algorithm needs to find a shortest route from point A to point B. It's provided with a map and the shorter the path it chooses, the higher the reward. Over multiple simulations, it learns to find the shortest routes. Adding other factors, like road congestion, can make the training approach more robust and give the model more parameters to optimise.

Setting up a Neural Network

The neural network library used in this project is TensorFlow. Follow the steps outlined in the Setting Up Python on Windows section of the report to install additional libraries: tensorflow and numpy.
-m pip install tensorflow numpy

Now we are ready to proceed with writing a script for the model in CoppeliaSim, as per the Adding Custom Programming Scripts section of this report. The full code with detailed comments on functionality is provided below.

```
#python
import random #this library allows generating pseudorandom numbers
import tensorflow as tf #this library creates and manages neural network
models
from tensorflow.keras import layers, models #this further imports the
```

```
tools to set up a neural network with a specified number of layers and
compile a learning model
import numpy as np #this library allows access to enhanced calculations
and data managements
import datetime #this library is used to date the model save files
import os #used to manage files on local drive
import time #used to record time when a model file is
created/overwritten

model_path = "worm_gen_3.keras" #the relative path to save the model for
future loading, defaults to the folder where CoppeliaSim is installed,
adjust accordingly to save elsewhere on the drive

#due to the number of inputs and outputs, the model for this project has
three layers: 2 of 128 neurons and 1 of 64 neurons
#it is possible to add more layers and neurons if simulations do not
perform as expected
def create_model(input_size, output_size):
    model = models.Sequential()
    model.add(layers.Dense(128, activation='relu',
input_shape=(input_size,)))
    model.add(layers.Dense(128, activation='relu'))
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dense(output_size, activation='tanh'))
#model.compile determines the optimizer to use for the model, in this
case the Adam optimizer. This optimizer includes a set of features to
enhance model learning and ensure model evolution
    model.compile(optimizer='adam', loss='mse')
    return model

#this function parses through the handles of the joints and the sensors
to get their current values (positions) in order to make predictions for
expected value changes as new values are being written to the joints
def get_sensor_data(joint_handles, sensor_handles):
    sensor_data = []
    for joint in joint_handles:
        sensor_data.append(sim.getJointPosition(joint))
    for proximity_sensor in sensor_handles:
        result, distance, _, _, _ =
sim.readProximitySensor(proximity_sensor)
        sensor_data.append(distance if result else -1.0)
    return np.array(sensor_data)

#this function writes the new position values to the joints
def set_actuator_commands(joint_handles, commands):
    for joint, command in zip(joint_handles, commands):
```

```
        print(f"Setting joint {joint} to position {command}")  # Debug
line
        sim.setJointTargetPosition(joint, command)

#this function contains the rewards system, which informs the model if
what it is doing is desired behavior, for this project currently reward
is only calculated based on worm displacement in an X/Y plane
def compute_reward(previous_position, current_position):
    forward_reward = np.linalg.norm(np.array(current_position[:2]) -
np.array(previous_position[:2]))
    return forward_reward


#this function contains the code that only runs once at the beginning of
a simulation
def sysCall_init():
#start the CoppeliaSim simulation
    sim = require('sim')
#variable declarations
    global model, joint_handles, sensor_handles, robot_handle,
previous_position, log_dir, tensorboard_callback, episode, total_reward,
epsilon

#model logs setup
    log_dir = "logs/fit/" +
datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
#calling this function allows visualising model behavior, for the
purposes of this project, model bahvor is observed within CoppeliaSim,
but using this function allows viewing model outputs outside of
CoppeliaSim
    tensorboard_callback =
tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1)

#Create a new model on the first run of the simulation and save it on
the drive, after that load the model from the drive and overwrite it
with new results obtained from each subsequent simulation
    if os.path.exists(model_path):
        model = tf.keras.models.load_model(model_path)
        print("Model loaded from disk.")
    else:
        input_size = 60  #37 joint positions read and 23 sensor values
read
        output_size = 37  #37 joint positions set based on prediction
        model = create_model(input_size, output_size) #create a neural
network model based on the number of values read (joints and sensors)
and the number of values set (joints)
```

```
        print("New model created.")

#very long chunk of code that records handles to access the sensors and
the joints within CoppeliaSim
    robot_handle = sim.getObject(".")
    joint_handles = [sim.getObject(":/REV11"),
                sim.getObject(':/REV11/Cuboid/PRIS1'),
                sim.getObject(':/REV11/Cuboid/PRIS1/Cuboid/REV12'),

sim.getObject(':/REV11/Cuboid/PRIS1/Cuboid/REV12/Cuboid/REV21'),

sim.getObject(':/REV11/Cuboid/PRIS1/Cuboid/REV12/Cuboid/REV21/Cuboid/PRI
S2'),

sim.getObject(':/REV11/Cuboid/PRIS1/Cuboid/REV12/Cuboid/REV21/Cuboid/PRI
S2/Cuboid/REV22'),

sim.getObject(':/REV11/Cuboid/PRIS1/Cuboid/REV12/Cuboid/REV21/Cuboid/PRI
S2/Cuboid/REV22/Cuboid/REV31'),

sim.getObject(':/REV11/Cuboid/PRIS1/Cuboid/REV12/Cuboid/REV21/Cuboid/PRI
S2/Cuboid/REV22/Cuboid/REV31/Cuboid/PRIS3'),

sim.getObject(':/REV11/Cuboid/PRIS1/Cuboid/REV12/Cuboid/REV21/Cuboid/PRI
S2/Cuboid/REV22/Cuboid/REV31/Cuboid/PRIS3/Cuboid/REV32'),

sim.getObject(':/REV11/Cuboid/PRIS1/Cuboid/REV12/Cuboid/REV21/Cuboid/PRI
S2/Cuboid/REV22/Cuboid/REV31/Cuboid/PRIS3/Cuboid/REV32/Cuboid/REV41'),

sim.getObject(':/REV11/Cuboid/PRIS1/Cuboid/REV12/Cuboid/REV21/Cuboid/PRI
S2/Cuboid/REV22/Cuboid/REV31/Cuboid/PRIS3/Cuboid/REV32/Cuboid/REV41/Cubo
id/PRIS4'),

sim.getObject(':/REV11/Cuboid/PRIS1/Cuboid/REV12/Cuboid/REV21/Cuboid/PRI
S2/Cuboid/REV22/Cuboid/REV31/Cuboid/PRIS3/Cuboid/REV32/Cuboid/REV41/Cubo
id/PRIS4/Cuboid/REV42'),

sim.getObject(':/REV11/Cuboid/PRIS1/Cuboid/REV12/Cuboid/REV21/Cuboid/PRI
S2/Cuboid/REV22/Cuboid/REV31/Cuboid/PRIS3/Cuboid/REV32/Cuboid/REV41/Cubo
id/PRIS4/Cuboid/REV42/Cuboid/REV51'),

sim.getObject(':/REV11/Cuboid/PRIS1/Cuboid/REV12/Cuboid/REV21/Cuboid/PRI
S2/Cuboid/REV22/Cuboid/REV31/Cuboid/PRIS3/Cuboid/REV32/Cuboid/REV41/Cubo
id/PRIS4/Cuboid/REV42/Cuboid/REV51/Cuboid/PRIS5'),

sim.getObject(':/REV11/Cuboid/PRIS1/Cuboid/REV12/Cuboid/REV21/Cuboid/PRI
```

```
S2/Cuboid/REV22/Cuboid/REV31/Cuboid/PRIS3/Cuboid/REV32/Cuboid/REV41/Cubo
id/PRIS4/Cuboid/REV42/Cuboid/REV51/Cuboid/PRIS5/Cuboid/REV52'),

sim.getObject(':/REV11/Cuboid/PRIS1/Cuboid/REV12/Cuboid/REV21/Cuboid/PRI
S2/Cuboid/REV22/Cuboid/REV31/Cuboid/PRIS3/Cuboid/REV32/Cuboid/REV41/Cubo
id/PRIS4/Cuboid/REV42/Cuboid/REV51/Cuboid/PRIS5/Cuboid/REV52/Cuboid/REV6
1'),

sim.getObject(':/REV11/Cuboid/PRIS1/Cuboid/REV12/Cuboid/REV21/Cuboid/PRI
S2/Cuboid/REV22/Cuboid/REV31/Cuboid/PRIS3/Cuboid/REV32/Cuboid/REV41/Cubo
id/PRIS4/Cuboid/REV42/Cuboid/REV51/Cuboid/PRIS5/Cuboid/REV52/Cuboid/REV6
1/Cuboid/PRIS6'),

sim.getObject(':/REV11/Cuboid/PRIS1/Cuboid/REV12/Cuboid/REV21/Cuboid/PRI
S2/Cuboid/REV22/Cuboid/REV31/Cuboid/PRIS3/Cuboid/REV32/Cuboid/REV41/Cubo
id/PRIS4/Cuboid/REV42/Cuboid/REV51/Cuboid/PRIS5/Cuboid/REV52/Cuboid/REV6
1/Cuboid/PRIS6/Cuboid/REV62'),

sim.getObject(':/REV11/Cuboid/PRIS1/Cuboid/REV12/Cuboid/REV21/Cuboid/PRI
S2/Cuboid/REV22/Cuboid/REV31/Cuboid/PRIS3/Cuboid/REV32/Cuboid/REV41/Cubo
id/PRIS4/Cuboid/REV42/Cuboid/REV51/Cuboid/PRIS5/Cuboid/REV52/Cuboid/REV6
1/Cuboid/PRIS6/Cuboid/REV62/Cuboid/REV71'),

sim.getObject(':/REV11/Cuboid/PRIS1/Cuboid/REV12/Cuboid/REV21/Cuboid/PRI
S2/Cuboid/REV22/Cuboid/REV31/Cuboid/PRIS3/Cuboid/REV32/Cuboid/REV41/Cubo
id/PRIS4/Cuboid/REV42/Cuboid/REV51/Cuboid/PRIS5/Cuboid/REV52/Cuboid/REV6
1/Cuboid/PRIS6/Cuboid/REV62/Cuboid/REV71/Cuboid/PRIS7'),

sim.getObject(':/REV11/Cuboid/PRIS1/Cuboid/REV12/Cuboid/REV21/Cuboid/PRI
S2/Cuboid/REV22/Cuboid/REV31/Cuboid/PRIS3/Cuboid/REV32/Cuboid/REV41/Cubo
id/PRIS4/Cuboid/REV42/Cuboid/REV51/Cuboid/PRIS5/Cuboid/REV52/Cuboid/REV6
1/Cuboid/PRIS6/Cuboid/REV62/Cuboid/REV71/Cuboid/PRIS7/Cuboid/REV72'),

sim.getObject(':/REV11/Cuboid/PRIS1/Cuboid/REV12/Cuboid/REV21/Cuboid/PRI
S2/Cuboid/REV22/Cuboid/REV31/Cuboid/PRIS3/Cuboid/REV32/Cuboid/REV41/Cubo
id/PRIS4/Cuboid/REV42/Cuboid/REV51/Cuboid/PRIS5/Cuboid/REV52/Cuboid/REV6
1/Cuboid/PRIS6/Cuboid/REV62/Cuboid/REV71/Cuboid/PRIS7/Cuboid/REV72/Cuboi
d/REV81'),

sim.getObject(':/REV11/Cuboid/PRIS1/Cuboid/REV12/Cuboid/REV21/Cuboid/PRI
S2/Cuboid/REV22/Cuboid/REV31/Cuboid/PRIS3/Cuboid/REV32/Cuboid/REV41/Cubo
id/PRIS4/Cuboid/REV42/Cuboid/REV51/Cuboid/PRIS5/Cuboid/REV52/Cuboid/REV6
1/Cuboid/PRIS6/Cuboid/REV62/Cuboid/REV71/Cuboid/PRIS7/Cuboid/REV72/Cuboi
d/REV81/Cuboid/PRIS8'),

sim.getObject(':/REV11/Cuboid/PRIS1/Cuboid/REV12/Cuboid/REV21/Cuboid/PRI
```

```
S2/Cuboid/REV22/Cuboid/REV31/Cuboid/PRIS3/Cuboid/REV32/Cuboid/REV41/Cubo
id/PRIS4/Cuboid/REV42/Cuboid/REV51/Cuboid/PRIS5/Cuboid/REV52/Cuboid/REV6
1/Cuboid/PRIS6/Cuboid/REV62/Cuboid/REV71/Cuboid/PRIS7/Cuboid/REV72/Cuboi
d/REV81/Cuboid/PRIS8/Cuboid/REV82'),

sim.getObject(':/REV11/Cuboid/PRIS1/Cuboid/REV12/Cuboid/REV21/Cuboid/PRI
S2/Cuboid/REV22/Cuboid/REV31/Cuboid/PRIS3/Cuboid/REV32/Cuboid/REV41/Cubo
id/PRIS4/Cuboid/REV42/Cuboid/REV51/Cuboid/PRIS5/Cuboid/REV52/Cuboid/REV6
1/Cuboid/PRIS6/Cuboid/REV62/Cuboid/REV71/Cuboid/PRIS7/Cuboid/REV72/Cuboi
d/REV81/Cuboid/PRIS8/Cuboid/REV82/Cuboid/REV91'),

sim.getObject(':/REV11/Cuboid/PRIS1/Cuboid/REV12/Cuboid/REV21/Cuboid/PRI
S2/Cuboid/REV22/Cuboid/REV31/Cuboid/PRIS3/Cuboid/REV32/Cuboid/REV41/Cubo
id/PRIS4/Cuboid/REV42/Cuboid/REV51/Cuboid/PRIS5/Cuboid/REV52/Cuboid/REV6
1/Cuboid/PRIS6/Cuboid/REV62/Cuboid/REV71/Cuboid/PRIS7/Cuboid/REV72/Cuboi
d/REV81/Cuboid/PRIS8/Cuboid/REV82/Cuboid/REV91/Cuboid/PRIS9'),

sim.getObject(':/REV11/Cuboid/PRIS1/Cuboid/REV12/Cuboid/REV21/Cuboid/PRI
S2/Cuboid/REV22/Cuboid/REV31/Cuboid/PRIS3/Cuboid/REV32/Cuboid/REV41/Cubo
id/PRIS4/Cuboid/REV42/Cuboid/REV51/Cuboid/PRIS5/Cuboid/REV52/Cuboid/REV6
1/Cuboid/PRIS6/Cuboid/REV62/Cuboid/REV71/Cuboid/PRIS7/Cuboid/REV72/Cuboi
d/REV81/Cuboid/PRIS8/Cuboid/REV82/Cuboid/REV91/Cuboid/PRIS9/Cuboid/REV92
'),
                sim.getObject(':/PRIS12'),

sim.getObject(':/REV11/Cuboid/PRIS1/Cuboid/REV12/Cuboid/PRIS22'),

sim.getObject(':/REV11/Cuboid/PRIS1/Cuboid/REV12/Cuboid/REV21/Cuboid/PRI
S2/Cuboid/REV22/Cuboid/PRIS32'),

sim.getObject(':/REV11/Cuboid/PRIS1/Cuboid/REV12/Cuboid/REV21/Cuboid/PRI
S2/Cuboid/REV22/Cuboid/REV31/Cuboid/PRIS3/Cuboid/REV32/Cuboid/PRIS42'),

sim.getObject(':/REV11/Cuboid/PRIS1/Cuboid/REV12/Cuboid/REV21/Cuboid/PRI
S2/Cuboid/REV22/Cuboid/REV31/Cuboid/PRIS3/Cuboid/REV32/Cuboid/REV41/Cubo
id/PRIS4/Cuboid/REV42/Cuboid/PRIS52'),

sim.getObject(':/REV11/Cuboid/PRIS1/Cuboid/REV12/Cuboid/REV21/Cuboid/PRI
S2/Cuboid/REV22/Cuboid/REV31/Cuboid/PRIS3/Cuboid/REV32/Cuboid/REV41/Cubo
id/PRIS4/Cuboid/REV42/Cuboid/REV51/Cuboid/PRIS5/Cuboid/REV52/Cuboid/PRIS
62'),

sim.getObject(':/REV11/Cuboid/PRIS1/Cuboid/REV12/Cuboid/REV21/Cuboid/PRI
S2/Cuboid/REV22/Cuboid/REV31/Cuboid/PRIS3/Cuboid/REV32/Cuboid/REV41/Cubo
id/PRIS4/Cuboid/REV42/Cuboid/REV51/Cuboid/PRIS5/Cuboid/REV52/Cuboid/REV6
1/Cuboid/PRIS6/Cuboid/REV62/Cuboid/PRIS72'),
```

```
sim.getObject(':/REV11/Cuboid/PRIS1/Cuboid/REV12/Cuboid/REV21/Cuboid/PRI
S2/Cuboid/REV22/Cuboid/REV31/Cuboid/PRIS3/Cuboid/REV32/Cuboid/REV41/Cubo
id/PRIS4/Cuboid/REV42/Cuboid/REV51/Cuboid/PRIS5/Cuboid/REV52/Cuboid/REV6
1/Cuboid/PRIS6/Cuboid/REV62/Cuboid/REV71/Cuboid/PRIS7/Cuboid/REV72/Cuboi
d/PRIS82'),

sim.getObject(':/REV11/Cuboid/PRIS1/Cuboid/REV12/Cuboid/REV21/Cuboid/PRI
S2/Cuboid/REV22/Cuboid/REV31/Cuboid/PRIS3/Cuboid/REV32/Cuboid/REV41/Cubo
id/PRIS4/Cuboid/REV42/Cuboid/REV51/Cuboid/PRIS5/Cuboid/REV52/Cuboid/REV6
1/Cuboid/PRIS6/Cuboid/REV62/Cuboid/REV71/Cuboid/PRIS7/Cuboid/REV72/Cuboi
d/REV81/Cuboid/PRIS8/Cuboid/REV82/Cuboid/PRIS92'),

sim.getObject(':/REV11/Cuboid/PRIS1/Cuboid/REV12/Cuboid/REV21/Cuboid/PRI
S2/Cuboid/REV22/Cuboid/REV31/Cuboid/PRIS3/Cuboid/REV32/Cuboid/REV41/Cubo
id/PRIS4/Cuboid/REV42/Cuboid/REV51/Cuboid/PRIS5/Cuboid/REV52/Cuboid/REV6
1/Cuboid/PRIS6/Cuboid/REV62/Cuboid/REV71/Cuboid/PRIS7/Cuboid/REV72/Cuboi
d/REV81/Cuboid/PRIS8/Cuboid/REV82/Cuboid/REV91/Cuboid/PRIS9/Cuboid/REV92
/Cuboid/PRIS102')]
    sensor_handles = [sim.getObjectHandle('FR'),
                      sim.getObjectHandle('FL'),
                      sim.getObjectHandle('R1'),
                      sim.getObjectHandle('L1'),
                      sim.getObjectHandle('R2'),
                      sim.getObjectHandle('L2'),
                      sim.getObjectHandle('R3'),
                      sim.getObjectHandle('L3'),
                      sim.getObjectHandle('R4'),
                      sim.getObjectHandle('L4'),
                      sim.getObjectHandle('R5'),
                      sim.getObjectHandle('L5'),
                      sim.getObjectHandle('R6'),
                      sim.getObjectHandle('L6'),
                      sim.getObjectHandle('R7'),
                      sim.getObjectHandle('L7'),
                      sim.getObjectHandle('R8'),
                      sim.getObjectHandle('L8'),
                      sim.getObjectHandle('R9'),
                      sim.getObjectHandle('L9'),
                      sim.getObjectHandle('R10'),
                      sim.getObjectHandle('L10'),
                      sim.getObjectHandle('B')]
    previous_position = sim.getObjectPosition(robot_handle, -1) #check
current position of the worm model in X/Y plane in CoppeliaSim
    episode = 0
    total_reward = 0
```

```
    epsilon = 1.0

def sysCall_actuation():
    global previous_position, total_reward, epsilon

    #Get the current sensor values and joint positions
    sensor_data = get_sensor_data(joint_handles, sensor_handles)
    sensor_data = sensor_data.astype(np.float32).reshape(1, -1)
    # Debugging: Print shape and data type of sensor data
    print(f"Sensor data shape: {sensor_data.shape}, data type:
{sensor_data.dtype}")

#initially the model will explore randomly (epsilon is high) but
exploration would decay over time to give way to model predictions based
on the earlier exploration stages
    if random.uniform(0, 1) < epsilon: #write random values to joints
for the “exploration” run
        action = np.random.uniform(low=-1.0, high=1.0,
size=(len(joint_handles),)).astype(np.float32)  # Ensure float32
        print(f"Random action: {action}")
    else:
#after running a series of exploratory actions, start transitioning to
predicting next joint values with the neural net model
        action = model.predict(sensor_data)
        print(f"Predicted action: {action}")

    action = action.reshape(-1) #format values to write into a 1d array
    commands = action
    print(f"Predicted action shape: {action.shape}, data type:
{action.dtype}")
    print(f"Commands: {commands}")
    commands = np.clip(commands, -30.0, 30.0) #not necessary, but clips
the values for joints from -30 to 30 to avoid seeing unnecessarily large
numbers, since joint ranges are defined in Coppelia, not in the code
#Call the function to write the random/predicted joint values to the
joints in CoppeliaSim
    set_actuator_commands(joint_handles, commands)

#Evaluate how much the model has displaced after writing the new
positions for the joints
    current_position = sim.getObjectPosition(robot_handle, -1)
    reward = compute_reward(previous_position, current_position)
    total_reward += reward

#record the new position of the model
    previous_position = current_position
```

```
#this section discounts the previous rewards to encourage further
learning, predicts possible new outputs and what the potential reward
would be and trains the model based on the actions leading to possible
rewards
    try:
        target = reward + 0.99 * np.amax(model.predict(sensor_data))
        target_f = model.predict(sensor_data)
        target_f[0][np.argmax(action)] = target
        model.fit(sensor_data, target_f, epochs=1, verbose=0,
callbacks=[tensorboard_callback])
    except Exception as e:
        print(f"Training error: {e}")  # Debug line

#This section reduces the preference to explore randomly, thus
increasing the need to learn and predict
    if epsilon > 0.01:
        epsilon *= 0.995

def sysCall_sensing():
    pass

def sysCall_cleanup():
    global episode, total_reward

#Save the model on the drive
    model.save(model_path)
    print(f"Model saved after episode {episode}. Total Reward:
{total_reward}")

#Increment episode count and reset total reward
    episode += 1
    total_reward = 0
```

Simulation Results

CoppeliaSim allows fast prototyping of new physical models and new learning models to go through more iterations than building and testing an actual robot.
Over the course of this project, three distinct worm models were designed and tested. The neural network setup was also updated continuously in an attempt to reach desired results. Currently, the learning model continues to overfit outputs for the easiest way to displace: the latest version wiggles to the left or to the right as opposed to developing a more complex algorithm of moving forward.
In order to resolve this issue, the rewards system needs to be adjusted to penalize wiggling and put more weight onto shifting links forward.

Conclusion

This project presented a great opportunity to get introduced to fast prototyping, simulation and testing of robot models in CoppeliaSim as well as to basic neural network setup.
The potential of neural networks is exciting and there are a lot of tools and methods to explore further in order to achieve desired behavior.
In spite of the current version of the C. Elegans model not acting as expected, it is clear what the next possible steps could be to improve the learning approach: define a more detailed rewards system that would encourage the model to advance forward by stepping individual links, as opposed to wiggling left and right.
Overall, this has been a valuable experience and a great introduction to the possibilities simulation tools ike CoppeliaSim and AI algorithms like neural networks can offer when it comes to designing robots.

Resources

1 - C. elegans: A sensible model for sensory biology - PMC (nih.gov)
2 - Lateral Facilitation between Primary Mechanosensory Neurons Controls Nose Touch Perception in C. elegans - ScienceDirect
3 - Robot simulator CoppeliaSim: create, compose, simulate, any robot - Coppelia Robotics
4 - Download Python | Python.org
5 - Error in Importing PyTorch and NumPy libraries - CoppeliaSim forums (coppeliarobotics.com)
6 - Python Scripts not Working as Intended - CoppeliaSim forums (coppeliarobotics.com)
7 - CoppeliaSim User Manual (coppeliarobotics.com)
8 - TensorFlow basics | TensorFlow Core